Computation as Logic
COMPUTATION AS LOGIC

René Lallement
Ecole Nationale des Ponts et Chaussées, Paris

Translated by John Plaice
Université Laval, Québec
In memory of
Jean-Bernard Saint
# Contents

1 Introduction  
   1.1 Introduction  
   1.2 Outline  
   1.3 Conventions  
   1.4 Keywords  
   1.5 Symbols  

2 The syntactic landscape  
   2.1 Words  
   2.2 Finite terms  
   2.3 Terms of the $\lambda$-calculus  
   2.4 Formal terms  
      Exercises  
      Bibliographic notes  

3 Reduction  
   3.1 The $\lambda$-calculus  
   3.2 Rewriting systems  
   3.3 Functional programming  
   3.4 Termination and confluence  
      Exercises  
      Bibliographic notes  

4 First-order logic  
   4.1 First-order formulas  
   4.2 Natural deduction  
   4.3 Normal forms  
   4.4 Theories  
      Exercises  
      Bibliographic notes  

5 Models  
   5.1 The semantics of first-order logic  

vii
## Contents

5.2 Models of a theory 159  
5.3 Herbrand's theorem 167  
5.4 The semantics of computation 174  
  Exercises 191  
  Bibliographic notes 194

### 6 Equational logic
6.1 Algebras and equations 195  
6.2 Unification 222  
6.3 Equational proofs 239  
  Exercises 248  
  Bibliographic notes 250

### 7 Resolution
7.1 Logic programs 251  
7.2 Semantics of definite programs 259  
7.3 Control structures 275  
7.4 Negation 286  
7.5 Equality and resolution 295  
  Exercises 299  
  Bibliographic notes 300

### 8 The computable landscape
8.1 Computability 301  
8.2 Decidability 321  
8.3 Metaprogramming 331  
8.4 A classification of problems 342  
8.5 Complexity 350  
  Exercises 360  
  Bibliographic notes 362

Hints to selected exercises 363  

Bibliography 370  

Index 378
Foreword

René Lallement was kind enough to ask me for a preface. I told him that I was incompetent, but he replied that his taste for logic began with a computer algebra seminar on the Knuth–Bendix algorithm that I gave here at the Ecole Polytechnique. Since I required no competence to set off the chain of events which resulted in this book, I was in no position to require any for the preface!

The few lines that follow are those of a non-specialist, a mathematician who has always\(^1\) been fascinated by computer science, but, in keeping with French tradition, very cautious with respect to logic. It is by spending time with the mathematical and logical objects that are computers and programs, and also with computer scientists and logicians, that I have become more reasonable, although I am still irritated by the form of monographs in logic and theoretical computer science.

Mathematicians, particularly French ones, are known for writing important and erudite books which are unreadable by the non-specialist. Each of us has attempted to do otherwise, but very few of us have succeeded in preventing formal rigour from meaning a ‘glaciation’ of form. It is when science is at its most formal and the technical syntax the most strict that the syntax should obscure the explanations the least. What is true for mathematics is even more true for logic, be it considered a part thereof or a younger sibling, where the need for formal rigour is greater. And here begins the irritation I was referring to: we sit in front of books which we feel must be talking about important things, but we cannot grasp the ideas, as they are hidden in perfectly adequate, hence completely unreadable, networks of definitions and systems of notations. One feels that one is missing the ‘folklore’ and that certain relations, probably obvious to those familiar with the field, are unattainable. But, as everyone knows, it is precisely this ‘folklore’ that is the most difficult to pass on.

It should be clear that, in my mind, the basic strength of René Lallement’s work is that it avoids these common problems. It is not the work of someone who ‘knows’ and who has come to tell us what is the best or unique way of writing and

\(^1\)Since the prehistoric age of the CAB 500 and the Gamma 3 (early French machines).
understanding, why we must forget everything that we have already learned and that we must stop asking the questions that we have been asking because they are the wrong questions. Although well structured, it will lend itself to several readings and will, I think, allow each and everyone to complete or update their knowledge without having to forget everything they have previously known.

On the other hand, each reader, according to their curiosity at a given moment, might regret that such and such a particular point is so little discussed, maybe not even at all. However, the detailed bibliographic notes allow the reader to go back to the specialized works to study these questions at greater length.

In summary, I enjoyed this text and recommend its reading and consultation without any reservations. I wish it great success.

Ecole Polytechnique, Paris
April 1990

Michel Demazure
Acknowledgements

This work is the result of teaching Logic and Computer Science to senior students at the Ecole Nationale des Ponts et Chaussées (ENPC) since 1985, as well as to graduate students in Artificial Intelligence and Pattern Recognition at the Université de Paris VI.

The book was written at the CERMA, the Centre of Applied Mathematics at the ENPC, with the encouragement and friendly advice of Nicolas Bouleau, whose taste for logic led him to create this course, which the author oriented towards programming. Many at the CERMA helped with their advice, even reading the preliminary versions of the book.

The author's interest in using logic to study computer science stems from attending enriching seminars given at the various research centers in Paris: the Laboratoire de Recherche en Informatique at Orsay, the Logic group at the Université de Paris VII, the Formal group at the Institut National de Recherche en Informatique et en Automatique (INRIA) and the Ecole Normale Supérieure.

Jean-Bernard Saint, with whom the author taught this course, shared his taste for logic programming with us and partially contributed to the writing of this book. He was a good friend of both of us and will be missed.

The translator would like to thank Catherine and Cyril for putting up with his nonexistence for many nights and weekends.

Ottawa
March 1992

René Lalement

John Plaice
Chapter 1

Introduction

Logic has at least two roles to play in computer science. First, computer science can be studied from the outside, theoretically, and logic can be used to study its foundations, as for mathematics. Second, the methods used in computer science are themselves logical methods.

In its first role, logic acts as meta-computer science: it offers computability and undecidability results such as Church’s, Gödel’s and Rice’s, and the classification of problems according to their complexity.

The second role has to do with programming being a logical activity. To design a program is to prove a proposition in a constructive manner. To execute a functional program is to normalize a proof. To execute a logic program is to construct a proof and extract information from it. To find an expression’s type is to find the proposition of which the expression is a proof. To interpret or optimize a program is to construct a more or less standard model, as if one were evaluating the validity or the consistency of a proposition. Logic’s inference systems and semantics are used extensively. Interpreters, compilers and optimizers are formalized using recursive function theory.

This work deals with two computing paradigms where logic plays a useful and essential role. An attempt will be made to show that logic is not only a language in which to state facts or knowledge accurately, even though it did once play this role, when the teaching of mathematics was overly formal, and still does so today, in the current guise of artificial intelligence.

The two computing paradigms to be studied are reduction and resolution, associated, respectively, with functional programming and logic programming.

Computing can be done by reducing an expression to a simpler one. Let \((2 + 3) \times (1 + 1)\) be a term to compute. A reduction of this term is

\[(2 + 3) \times (1 + 1) \rightarrow 5 \times (1 + 1) \rightarrow 5 \times 2 \rightarrow 10.\]

It stops as there is no rule which applies to 10. Other reductions are possible, as
in

\[(2 + 3) \times (1 + 1) \rightarrow 2 \times (1 + 1) + 3 \times (1 + 1) \rightarrow 2 \times 2 + 3 \times 2 \rightarrow 4 + 6 \rightarrow 10,\]

which ends with the same term. This confluence of all the reductions flowing out of the same term is an essential property of this form of computing, as it allows a choice of ‘reduction strategies’.

Another approach is to consider computing as the resolution of a problem. A typical example is an equation such as

\[x^2 - 4 = 0 \Leftrightarrow x = 2 \text{ or } x = -2.\]

There are two solutions to this equation.

The resolution method must be complete, i.e., it must find all the solutions. Since the resolution of a problem often calls for solving other problems, many solutions must be managed and an order must be found to compute, test, accept, reject or combine these partial solutions.

Computer science will be approached in a formal manner, by first considering the need to manipulate finite objects. Computer science uses numbers, programs and files. Logic uses terms, formulas and proofs. In both cases, words, i.e., character strings, are used to represent these objects. Each of these notations has a concrete syntax, chosen to enhance communication, be it between humans or computers. For example, the notation for formulas, which uses variables, relational and functional symbols, connectives and quantifiers, is quite familiar. The formula

\[(x > 0) \land \forall y \forall z (\neg (y \cdot z = x) \lor y = x \lor z = x)\]

is a word over the alphabet formed of the symbols \((, x, >, 0, \), \(\land, \forall, y, x, \neg, \cdot, = \) and \(\lor\). The surface representation of programs and formulas is similar, and so is their abstract structure, which is stronger than the structure of words. For the preceding formula, \((> 0) \land \forall\) is a subword but not a subformula, while \(x > 0\) is a subformula.

More precisely, there are three fundamental structures by which these objects can be classified:

- Arithmetic applicative-language expressions, first-order terms, proofs in a Hilbert inference system and Prolog values are finite terms.
- Functional language programs, first-order formulas and natural deduction proofs are lambda (\(\lambda\)) terms.
- Sequential control structures and Prolog-II values are rational terms.

It turns out that \(\lambda\)-terms and rational terms can be represented by finite terms, which makes the latter the major structure studied in this book.

To abstract is to simplify: by ignoring the details of concrete syntax, one can study in a general manner these elementary objects which form the basis for the
syntactic landscape of logic and computing; this is the purpose of Chapter 1. The same methods, for definition and proof, will be used to reason about these objects, whether they be programs, formulas or proofs. Structural induction is a powerful method which generalizes induction over the integers for finite terms. More flexible techniques are also presented: monotone and well-founded induction.

A second aim of Chapter 1 is to show the reader the difference between what is elementary and what is not, a difference which will be elaborated in Chapter 7. Since its beginnings, it has been clear that computer science deals with finite objects: finite state automata, finite alphabets and finite programs. Mathematics has long escaped from finite domains, as in real analysis. The use of algebra in mathematics at the end of the nineteenth century reintroduced the concept of ‘finite’, as did the development of computer algebra more recently. Elementary algebraic geometry studies manifolds, such as curves and surfaces, under the form of ideals of polynomials which have a finite number of generators. Algebraic topology studies topological spaces through algebraic structures such as the fundamental group, constructed using generators and relations from a finite description analogous to a graph. The need, still not fully formalized, for finiteness in the description and solving of problems yielded problems which were considered ‘difficult’: for example, the solving of polynomial equations over the integers, or the untying of a knot. One can imagine the surprise experienced by mathematicians when this need, extended by Hilbert to cover reasoning itself—to ensure, he thought, its foundation—brought forth a new degree of difficulty: ‘undecidability’. The meta-mathematical problem of deduction in a theory as simple as that of arithmetic turned out to be undecidable (Church’s theorem). Later, it was discovered that some of the most difficult mathematical problems are in fact undecidable. In computer science, where finiteness is necessary, undecidable problems abound: Rice’s theorem is a remarkable generator of undecidable problems. These questions will be addressed in the last chapter, which examines computability, i.e., general problems of computation.

Functional programming corresponds to ‘computing as reduction’. Two examples of reduction are studied in Chapter 2: the λ-calculus and rewriting systems.

The λ-calculus was created as an attempt to transform logic into a mechanically verifiable calculus. It is a formal method with few elements, the symbol λ, an infinite number of variables and parentheses, and one computing rule, β-reduction, which represents the passing of a parameter to a function. It is remarkable that one can encode all data structures and ‘run’ all computing algorithms with this simple calculus. For example, the Booleans true and false are encoded by the λ-terms

\[ T \equiv \lambda x \lambda y \, x \quad \text{and} \quad F \equiv \lambda x \lambda y \, y, \]

which represent branchings in the sense that the λ-term \( BMN \), which transforms itself by β-reduction to \( M \) if \( B \) is \( T \) and to \( N \) if \( B \) is \( F \), behaves as the \( \text{if } B \text{ then } M \text{ else } N \) of usual languages. The logical \( \text{and} \) is coded by

\[ \text{and} \equiv \lambda x \lambda y \, (x \, y \, F), \]

so that, for example, \( \text{and} \, TF \) reduces to \( F \). It will be shown how integers, pairs, arithmetic operations and recursive definitions of functions can be encoded. One naturally writes ‘higher-order functional programs’,
which take functions as parameters and return functions as results.

Strangely enough, most programming does not venture beyond first-order functions. In mathematics, higher-order functions have been used with success for more than a century, e.g., the functionals and operators of analysis, to implement general methods. Lisp has contributed to this situation: for a long time it was the only language allowing higher-order programming. Its faults were therefore associated with higher-order programming. But Lisp was not a functional language, despite appearances! There now exist real functional languages whose compiled code is of acceptable efficiency. For non-numeric applications dealing with highly structured objects, such as in logic or software engineering, e.g., languages, programs and proofs, functional programming is an ideal programming paradigm. Functional programming is also widely used in artificial intelligence because of its ability to manipulate symbolic data and to define ‘strategies’.

The functional language used in this book is Standard ML (of New Jersey).\(^1\) ML programs are ‘typable’ \(\lambda\)-terms. For example, the Booleans \(T\) and \(F\) can be coded by

\[
\text{val } T = \text{fn } x \Rightarrow (\text{fn } y \Rightarrow x)
\]

and

\[
\text{and } F = \text{fn } x \Rightarrow (\text{fn } y \Rightarrow y)
\]

where \(\text{fn}\) corresponds to \(\lambda\). The compiler computes the types of \(T\) and \(F\),

\[
\text{\smaller \(\bowtie\) val } \text{T} = \text{fn : 'a } \rightarrow \text{ 'b } \rightarrow \text{ 'a}
\]

\[
\text{val } \text{F} = \text{fn : 'a } \rightarrow \text{ 'b } \rightarrow \text{ 'b}
\]

which contain the type variables \('a\) and \('b\), which represent arbitrary types.

A logic consists of a language, an inference system and an interpretative system. It it used to prove and interpret.

Chapter 3 introduces the language of first-order logic and several inference systems: natural deduction systems, the sequent calculus and Hilbert systems. These systems will first be presented in the restricted form of minimal logic, without negation, then in intuitionistic logic and finally in classical logic. The latter is the most familiar, with its rule of reasoning by contradiction. The connectives follow certain algebraic properties which clearly appear in the sequent calculus and allow one to ‘normalize’ formulas.

Here is a proof in natural deduction of the formula

\[
\varphi \equiv ((A \Rightarrow B) \land (A \Rightarrow C)) \Rightarrow (A \Rightarrow (B \land C)),
\]

where \(A, B\) and \(C\) are arbitrary formulas. To prove \(\varphi\), suppose \((A \Rightarrow B) \land (A \Rightarrow C)\). From this assumption, deduce \(A \Rightarrow B\) and \(A \Rightarrow C\). In addition, suppose \(A\). From \(A \Rightarrow B\) and \(A\), conclude \(B\). From \(A \Rightarrow C\) and \(A\), conclude \(C\). Thus \(B\) and \(C\) have been deduced. Therefore \(B \land C\) has been deduced, by supposing \((A \Rightarrow B) \land (A \Rightarrow C)\) and \(A\). By discharging the assumption \(A\), \(A \Rightarrow (B \land C)\) is proven by supposing

\(^1\)The (original) French version of this book used CAML.
only \((A \Rightarrow B) \land (A \Rightarrow C)\). By discharging this last assumption, \(\varphi\) is proven with no assumption. It is simpler to represent this proof as a derivation, a formal object which is drawn as a tree where the deduction proceeds from the assumptions at the top to the conclusion at the bottom:

\[
\begin{array}{c}
(A \Rightarrow B) \land (A \Rightarrow C) \\
\hline
A \Rightarrow B \\
\hline
B \\
\hline
A \Rightarrow C \\
\hline
B \land C \\
\hline
A \Rightarrow (B \land C)
\end{array}
\]

\[
\begin{array}{c}
(A \Rightarrow B) \land (A \Rightarrow C) \\
\hline
A \Rightarrow C \\
\hline
C
\end{array}
\]

The above derivation can be written as a (typed) \(\lambda\)-term. In ML, this becomes

\[
\text{fn } h \Rightarrow (\text{fn } x \Rightarrow (\#1 \ h \ x, \ #2 \ h \ x))
\]

where \#1 returns the first element in a tuple and \#2 returns the second. The type computed by ML is

\[
\text{\_ val it = fn : (\' a \rightarrow \' b ) \ast (\' a \rightarrow \' c ) \rightarrow \' a \rightarrow \' b \ast \' c}
\]

which is precisely the formula \(\varphi\), where implication \(\Rightarrow\) and conjunction \(\land\) are replaced by the functional arrow \(\rightarrow\) and the product \(\ast\) of types, and \(A, B\) and \(C\) by type variables \(' a\), \(' b\) and \(' c\).

This functional interpretation of a proof is based on the Curry–Howard correspondence, which plays an essential role in the relations between logic, \(\lambda\)-calculus and programming. This relation can be defined by the equations:

\[
\text{proof} = \lambda\text{-term} = \text{program},
\]

\[
\text{proposition} = \text{type} = \text{specification}.
\]

A program specification is always a formula in a logical language which states what a program must do. However, proving that a program conforms to a specification requires an \textit{external} logic, typically formed by putting together programs and first-order formulas. Yet, each proof–proposition and term–type pair forms a single entity. The program–specification pair should be understood in as integrated a manner as the other two pairs. In the same way that one never writes a proof without knowing what one is proving, it should not be possible to write programs without knowing what specification a program is implementing. The Curry–Howard correspondence, which deals particularly with functional programming, allows one to consider a program as a proof of its specification, hence programming as looking for a proof. ML’s type system, already powerful for a programming language, is still poor as a specification language. All numeric functions have the same type: they take \texttt{int} to \texttt{int} or \texttt{real} to \texttt{real}. There are also type systems capable of distinguishing types in a finer manner, and one hopes that they will be the basis for future programming languages. These systems correspond to constructive logics: this is true for intuitionistic logic, but not for classical logic.

Here is a well-known example of a non-constructive proof. One must show that there exist two irrational numbers \(a\) and \(b\) such that \(a^b\) is rational. Consider \(\sqrt{3}^{\sqrt{2}}\):
• if $\sqrt{3}^{\sqrt{2}} \in \mathbb{Q}$, then $a = \sqrt{3}$ and $b = \sqrt{2}$;
• if $\sqrt{3}^{\sqrt{2}} \notin \mathbb{Q}$, then $a = \sqrt{3^{\sqrt{2}}}$ and $b = \sqrt{2}$, such that $a^b = 3 \in \mathbb{Q}$.

This proof is correct but it does not yield a value: which of the pairs $(a, b)$ is correct?\(^2\) One of the ideas of logic programming will be to extract a value from a result.

Chapter 4 introduces the other logical approach: semantics. A semantics’s purpose is to examine the possible meaning of an object. In general, this meaning is the result of an evaluation, as is the value of an expression in programming or the truth value of a logical formula.

The computation of this value reduces the operational content of the object: the two expressions $(1 + 1) + 1$ and $2 \times 2$ have the same value in $\mathbb{N}$, yet they have distinct operational meanings. What is lost by equating objects with the same denotation is gained in terms of freedom of interpretation: the two preceding expressions have value 4 in $\mathbb{N}$, as well as 0 in $\mathbb{Z}/2\mathbb{Z}$.

Take the example of arithmetic expressions, which are first-order terms: if $x, y > 0$ and $z < 0$, what is the sign of $xyz$? The axioms of arithmetic could be used to construct a derivation of the formula $xyz < 0$. However, the variables can be interpreted in $\{-1, +1\}$, where $x$ and $y$ become +1 and $z$ becomes −1, and the expression to be evaluated becomes $(+1)(+1)(-1) = -1$. Even if it is known that $x = 3$, $y = 7$ and $z = -2$, it is unnecessary to compute the value of $xyz$ in $\mathbb{Z}$ to determine its sign, the computation in $\{-1, +1\}$ being sufficient. Although $\mathbb{Z}$ is the standard domain of interpretation for arithmetic, there are many ‘non-standard’ domains, such as $\{-1, +1\}$.

In standard mathematical usage, formulas in predicate logic represent set properties. For example, a set with an operation which satisfies the axioms of group theory is a group. To answer the question, ‘can the formula $\forall x \forall y (xy = yx)$ be deduced from the axioms of group theory?’, it suffices to know that there exist non-commutative groups. It is much easier—for a mathematician, not a computer—than to prove that it is impossible to construct a proof of this formula.

Tarski’s concepts of interpretation and model allow such reasoning. The meaning of logical connectives given by the syntactic construction of derivations is the same as the interpreted meaning in a set. Even though syntactic methods are naturally adapted to computer science, semantic methods appear more intelligent, as they are more flexible. To study the semantics of a language is to consider its possible meanings. According to need, special cases can be treated by choosing a particular meaning with the purpose of refuting a proposition or to obtain partial information about an object.

Hence non-standard interpretations of programming languages allow optimizations, e.g. parallelization of code, without creating the standard interpretation of

\(^2\)The Gelfond–Schneider theorem, which solves one of Hilbert’s problems, states that an algebraic number raised to an irrational power is transcendental. Hence the correct pair is $a = \sqrt{3}^{\sqrt{2}}$ and $b = \sqrt{2}$. 

a program. In artificial intelligence, qualitative interpretations of numeric problems allow an estimation of essential phenomena such as growth or stability of measures: this is called qualitative physics. Normal life is led with one or several familiar models in mind. In mathematics, one might manipulate the consequences of the axioms of Hilbert spaces, but one lives in the space $L^2$. In computer science, one writes specifications of programs and of data structures, but the physical implementation is in one's mind. Is this a problem of too much generality?

Not at all! Semantics offers an incredibly rich set of possibilities. There are uncountable models of arithmetic and countable models of set theory! These models are unintuitive, but inevitable. Semantics offers an additional means of abstracting when faced with complex problems. Chapter 4 is concerned more with justifying such methods than with showing their use in the above domains. Logic is probably the domain of application where the methods have been illustrated most brilliantly, particularly for the proof of independence theorems in set theory.

It is easy to use these deductive or semantics methods to construct a formalism with inference rules or semantics. However, the definitions given for minimal, intuitionistic or classical logic are not arbitrary. They respect some important properties such as cut-elimination and normalization, symmetries such as introduction/elimination, and completeness and soundness results. It is not by presenting a formalism as an inference system that one does 'logic'. There are similar risks in mathematics when one chooses axioms liberally: these have little chance of defining anything of mathematical interest.

Chapter 5 deals with equational logic. Equations can be used to state problems to solve algebraic, differential and fixpoint equations as well as to state universal laws ($x + y = y + x$) or relations ($i^2 = -1$, $E = mc^2$) between values which are a priori unequal.

Of course, in predicate logic, this difference in meaning is written using quantifiers. To solve an equation $x^2 + 1 = 0$ proves $\exists x(x^2 + 1 = 0)$, while commutativity is written $\forall xy(x + y = y + x)$. Equational problems can clearly be written using a fragment of first-order logic, but the methods used are not those of the predicate calculus. Furthermore, solving an equation does not entail just proving an existential theorem, since an explicit value is sought in a particular set.i.e., one does not look for a proof of $\exists x\phi(x)$, rather for a value $a$ such that there exists a proof of $\phi(a)$. For example, an algebraic equation does not have the same solutions in $\mathbb{R}$ or in $\mathbb{C}$. The search for solutions will be restricted to sets of terms, where resolution becomes unification.

It turns out that many algebraic structures and many data structures used in algorithms are axiomatized by equations stating such properties as the associativity or commutativity of an operation, and that the semantics of classical logic can be given in Boolean algebras, which can be given an equational axiomatization. Functional programming defines functions in an equational manner, and evaluation of expressions can be seen as a rewriting process which uses the directed rules. The $\lambda$-calculus and combinatory logic are equational formalisms. The type inference
done by the ML compiler consists of solving equations generated by the constraints of the formation of typed λ-terms. As for logic programming, even though the programs themselves are not equations, the interpreter essentially solves equations, and their solutions are returned as results. All of these applications, through different problems and paradigms, show the importance of equational logic.

Logic programming, the purpose of study of Chapter 6, corresponds to ‘computation as resolution’. Unification is already an example of this paradigm, but it is such a simple case that the essential properties of soundness, completeness and switching are hardly visible. To talk of resolution in logic programming is to play with words. A query’s solutions are generated by the resolution of equations, but this process is also a particular case of Robinson’s resolution principle, which is a general method of automatic theorem proving. Logic programs and queries can be read as formulas of a particular form, Horn clauses, for which it is possible to generate constructive proofs. Clause p(M) is proven for a first-order term M, and not just Exp(x). The declarative semantics of logic programs describes what can be computed, and the operational semantics describes how computations must be done. The history of logic programming is shorter than that of functional programming, yet there are already more interpretations of logic programming: theorem proving, constraint solving, communicating processes and more.

The value of different versions of Prolog will be shown. In particular, Prolog–II and its rational terms, and NU–Prolog and its sound negation and quantifiers and its use of coroutines to eliminate useless non-determinism.
Chapter 2

The syntactic landscape

2.1 Words

Words are the basic syntactic objects of computer science. Typically known as character strings, they are used to write the more structured objects described later on in this chapter.

Let \( A \) be a set, called alphabet, whose elements are called symbols. A word over \( A \) is a finite sequence of elements of \( A \). The set of words over \( A \), written \( A^* \), is defined by

\[
A^* = \bigcup_{n \geq 0} A^n.
\]

If \( u \in A^n, |u| = n \) is the length of \( u \). There is a binary operation over \( A^* \), called concatenation, written \( \cdot \), or by simple juxtaposition,

\[
A^p \times A^q \rightarrow A^{p+q},
\]

\[
(u, v) \mapsto u \cdot v \text{ or } uv,
\]

where \( u \cdot v = (u_1, \ldots, u_p, v_1, \ldots, v_q) \) if \( u = (u_1, \ldots, u_p) \) and \( v = (v_1, \ldots, v_q) \). This operation is associative and has as its neutral element the empty word \( \epsilon \), the unique element of \( A^0 \). Thus \( (A^*, \cdot, \epsilon) \) is a monoid, called the free monoid generated by \( A \).

Given these definitions, a useful notation is needed. The single element \( (a) \) is written \( a \), and so \( u = (u_1, \ldots, u_n) \in A^n \) is the result of the concatenation of the symbols that make it up: \( u = u_1 \cdot u_2 \cdots \cdot u_n \), or simply \( u = u_1 \ldots u_n \).

Any subset of \( A^* \) is called a (formal) language over the alphabet \( A \).

Examples

- If \( A = \emptyset \), then \( A^* \) is simply \( \{\epsilon\} \).
- If \( A = \{1\} \), then the monoid \( (A^*, \cdot, \epsilon) \) is isomorphic to \((\mathbb{N}, +, 0)\). There are as many languages over the alphabet \( A \) as there are subsets of \( \mathbb{N} \), i.e., as many as there are real numbers—uncountably many.
If \( A = \{0, 1\} \), then \( A^* \) is the set of binary words.

Programming languages generally use the 'ASCII' set as the alphabet. It has 128 symbols.

The syntactic study of languages is an important branch of computer science. If \( A \neq \emptyset \), there exists an uncountably infinite number of languages. Only a countably infinite number of subsets can be obtained through 'finite means', or recursively. The other subsets will be of little use to computer science. To define languages, completely operational techniques, such as automata and grammars, must be devised. These techniques must at the very least be able to determine, for a given language \( L \), whether a word belongs to \( L \). This problem is called the recognition problem for \( L \). These questions will be addressed in the final chapter of the book.

The following operations over languages are commonly used:

- union: \( L \cup L' \);
- concatenation: \( LL' = \{uu'; u \in L, u' \in L'\} \);
- Kleene closure: \( L^* = \{u_1 \ldots u_n; n \geq 0, u_1, \ldots, u_n \in L\} \).

For \( a \in A \), \( \{a\}L \) will be written \( aL \).

Some order relations will be used over \( A^* \). Let \( u, v \in A^* \). Word \( u \) is called a prefix of \( v \), written \( u \leq v \), if there exists a \( u' \in A^* \) such that \( v = uu' \). This unique \( u' \) is written \( v/u \). The relation \( \leq \) is called the prefix order. Write \( u \perp v \) if \( u \) and \( v \) are incomparable under this order, i.e., if \( u \not\leq v \) and \( v \not\leq u \). The two are then called independent.

Recall a few useful definitions on order relations and variants. A relation is

- a strict order if it is irreflexive and transitive;
- a preorder if it is reflexive and transitive;
- an order if it is reflexive, antisymmetric and transitive.

If \( > \) is a strict order, and if \( \geq \) is defined by \( x \geq y \) if \( x > y \) or \( x = y \), then \( \geq \) is an order. If \( \geq \) is an order, then \( > \), defined by \( x > y \) if \( x \geq y \) and \( x \neq y \), is a strict order. Finally, if \( \geq \) is a preorder, \( x \sim y \) if \( x \geq y \) or \( y \geq x \) is an equivalence relation, and \( x > y \) if \( x \geq y \) and \( y \nless x \) is a strict order.

### 2.2 Finite terms

The structure of words, or rather that of the free monoid, is poor: all symbols are of the same kind. The terms presented in this section are given a structure which differentiates symbols by their usage.
2.2.1 Signatures

For arithmetic expressions, there are operators (+, ×), variables and constants. The structure of these expressions comes from the functional aspect of the symbols, each of which requires a fixed number of arguments, called its arity. In a programming language, one might find

- constant symbols: true, 256, ′,
- unary symbols: not, –;
- binary symbols: or, while, do, <,
- ternary symbols: if, then, else.

where – indicates where an argument should be placed. The introduction of these symbols defines the signature of the language.

Definition 2.1 A signature is a graded set, i.e., a set Σ with a mapping ar : Σ → N. If f ∈ Σ and ar(f) = n, f is of arity n, or f is n-ary. If n ≥ 1, f is a functional symbol, and if n = 0, it is a constant symbol.

T_Σ is the smallest subset E of the free monoid Σ* such that

1. if c ∈ Σ, ar(c) = 0, then c ∈ E;
2. if f ∈ Σ, ar(f) = n ≥ 1, and if M_1, ..., M_n ∈ E, then fM_1...M_n ∈ E.

The elements of T_Σ are called Σ-finite terms, first-order terms or just terms.

Example Let Σ = {0, s, +} with ar(0) = 0, ar(s) = 1, and ar(+) = 2. Then 0, s0, +00, +s00 ∈ T_Σ, but +0, s00 ∉ T_Σ.

So, in attempting to construct objects more structured than words, more words were produced! This is not a problem, since the structure is in the definition itself, not in the objects produced.

In addition, the definition of T_Σ is given using a family of subsets already containing T_Σ: it appears to be a circular definition. The following section clarifies these questions.

2.2.2 Inductive definitions

Inductive reasoning on the integers is well known: to prove a property P over the non-negative integers, one proves it for 0 and then proves that if it is true for n, then it is true for n+1. Intuitively, one proves P for 0, then for 1, then for 2, etc. If this reasoning works for the integers, it is because the integers were defined—using Peano’s definition—so that it would work. The gist of this definition is that a constructive procedure is itself a proof. One can conceive of this procedure, either in a relative manner, if the integers are constructed from already known objects (such as in set theory or in a ‘notation’ system), or in an absolute manner, if one starts with nothing.
Inductive definitions and inference systems

The definition given for terms is relative to the free monoid; the result is that they can be written using words. The relative aspect can be seen in the sentence, 'The smallest subset $E$ of the free monoid $\Sigma^*$ such that ...'. Since $T_\Sigma$ is now defined, a proof procedure using induction over terms can be validated in $\Sigma^*$.

Proposition 2.1 Let $P$ be a property over words. If $P$ is true for each constant symbol in $\Sigma$, and if for each $f \in \Sigma$ of arity $n \geq 1$, one can prove that $P$ is true for the word $fM_1 \ldots M_n$ if it is true for the words $M_1, \ldots, M_n$, then $P$ is true for all terms.

Proof Let $E$ be the subset of $\Sigma^*$ which contains all the words for which $P$ holds. The set $E$ contains the constants, and contains $fM_1 \ldots M_n$ as soon as it contains $M_1, \ldots, M_n$. By the definition of $T_\Sigma$, $T_\Sigma \subseteq E$, i.e., $P$ holds for all terms.

This procedure is called proof by structural induction. It is valid for all objects defined inductively and will often be used in this book. It will also be used for definitions. For example, the height $h(M)$ of a term $M$ is defined using structural induction:

$$h(a) = 0,$$
$$h(fM_1 \ldots M_n) = \max(h(M_1), \ldots, h(M_n)) + 1,$$

for every constant symbol $a$ and every symbol $f$ of arity $n \geq 1$.

An inductive definition can also be presented as an inference system. An inference system is a set of inference rules dealing with judgments that are to be proven. The following judgments are of interest: 'M is a term', 'the formula $\varphi$ is true', 'the type of term $M$ is $\tau$', or 'the value of expression $E$ is $v$'.

An inference rule has the following form:

$$(\text{name}) : \frac{j_1 \ldots j_n}{j},$$

where $j_1, \ldots, j_n$ are judgments, called premises, $j$ is a judgment, called conclusion, name is the name of the rule, and the integer $n \geq 0$ is its arity. The rule presents the inference of the conclusion from its premises. If $n = 0$, the rule asserts the conclusion $j$, which is then called an axiom of the inference system. If such is the case, the horizontal bar is often not written.

The simplest example of an inference system is given by the rules of an inductive definition. For terms, what is of interest is the judgment 'term $M$' stating that $M$ is a term. The signature $\Sigma$ determines the following rules, with one ($n$-ary) rule for each ($n$-ary) symbol:

$$(c) : \frac{}{\text{term } c}$$

$$(f) : \frac{\text{term } M_1 \ldots \text{term } M_n}{\text{term } fM_1 \ldots M_n}$$
where \( c, f \in \Sigma \) and \( \ar(c) = 0, \ar(f) = n \). These rules are completely deterministic. Their premises are uniquely determined by their conclusions. This property is held by all definitions which use structural induction.

Consider for a moment the possibility that the \( M \) in ‘term \( M \)’ might designate any word in \( \Sigma^* \). These rules can be used to prove that a word is a term. The proof is set up in the form of a tree with the root ‘at the bottom’. Consider the word \( +0s0 \). Since 0 is a term, the system can infer that \( s0 \) is a term, and with this intermediate result, that \( +0s0 \) is a term. This reasoning is presented using a new syntactic form, a derivation:

\[
\begin{align*}
(0) : & \quad \text{term} \quad 0 \\
(+): & \quad \text{term} \quad +0s0
\end{align*}
\]

\[
\begin{align*}
(0) : & \quad \text{term} \quad 0 \\
(s): & \quad \text{term} \quad s0
\end{align*}
\]

A derivation is a tree in which every node is labeled by a judgment and by an inference rule (in the current case, by a term and by a symbol from the signature). The judgment at the root is the conclusion of the derivation. The rules in the leaves are of arity zero (here, the introduction of a constant). If \( M \in \Sigma^* \), then \( M \in T_\Sigma \) if and only if there exists a derivation of this inference system whose conclusion is ‘term \( M \)’. In general, the symbol \( \vdash \) (‘turnstile’) is used to note, as in \( \vdash j \), that there exists a derivation of \( j \). Judgment \( j \) is called a theorem of the inference system. A new definition for terms is thus obtained:

\[
T_\Sigma = \{ M \in \Sigma^* ; \vdash \text{term} \; M \}.
\]

It is clear that if \( M \in T_\Sigma \), the judgment ‘term \( M \)’ has a unique derivation whose form is the same as that of \( M \): it is a very particular form of inference system.

Note that if the \( M \) is omitted from ‘term \( M \)’, a derivation for the judgment ‘term’ is obtained. The derivations are of the same form. In fact, the new derivation is another notation, treelike, for that term:

\[
\begin{align*}
(0) : & \quad \text{term} \\
(+): & \quad \text{term} \quad +0s0
\end{align*}
\]

\[
\begin{align*}
(0) : & \quad \text{term} \\
(s): & \quad \text{term} \quad s0
\end{align*}
\]

Derivations of the judgment ‘term’ (which can be read as ‘there exists a term’) are all terms (or rather their notation in tree form). This structure of derivations is taken as the abstract syntax of terms. This syntax will be defined in the next section, and used extensively later on. A derivation of ‘term’ can then be defined to be a term.

Inference systems used from now on will of course be more complex than those associated with a signature.
Well-founded relations
In logic, reasoning must be ‘well-founded’, in a very definite sense. Write \( j > j' \) if the judgment \( j \) is the conclusion of a rule for which \( j' \) is a premise. For terms, term \( M > \) term \( M' \) means that \( M' \) is an immediate subterm of \( M \). The immediate subterms of \( fM_1 \ldots M_n \) are \( M_1, \ldots, M_n \). For example, \( s0 \) is an immediate subterm of \( +0s0 \). A proof of judgment \( j \) would not be well-founded if there were an infinite number of premises \( j > j_1 > j_2 > \ldots \); this is the problem of ‘primary causes’ of philosophy. This situation cannot occur with terms since an immediate subterm of \( M \) can only be written with fewer symbols than \( M \), and there does not exist an infinitely decreasing sequence in \( \mathbb{N} \).

Definition 2.2 A binary relation \( \prec \) over a set \( E \) is well-founded, in short \( \text{wf} \), if every non-empty subset \( A \) of \( E \) contains a minimal element \( a_0 \) in \( A \), i.e., such that \( a \neq a_0 \) for all \( a \in A \).

A \( \text{wf} \) relation is irreflexive (i.e., \( x \not\prec x \) for all \( x \in E \); take \( A = \{ x \} \) in the definition) and, more generally, acyclic (if \( a_1 \prec a_2 \prec \ldots \prec a_n \prec a_1 \), take \( A = \{ a_0, \ldots, a_n \} \)). An order relation can be \( \text{wf} \) only over the empty set. A \( \text{wf} \) order will refer to a strict well-founded order. Even if a \( \text{wf} \) relation is not necessarily transitive, the terminology of order relations such as minimal element and decreasing sequence will be used.

An equivalent description of \( \text{wf} \) relations is: there does not exist an infinitely decreasing sequence \( a_0 \succ a_1 \succ a_2 \succ \ldots \). In fact, if there exists an infinitely decreasing sequence, the set \( A \) has no minimal element. On the other hand, if there exists a subset \( A \) which has no minimal element, take any \( a_0 \in A \). Since \( a_0 \) is not minimal, there exists \( a_1 \in A \) such that \( a_1 \prec a_0 \) and one can iterate to construct an infinite decreasing sequence.

Examples
- The usual strict order is \( \text{wf} \) over \( \mathbb{N} \), but not over \( \mathbb{Z} \) or \( \mathbb{Q} \).
- The relation over \( \mathbb{N} \), ‘\( m \succ n \) if \( m \) is the successor of \( n \)’ is \( \text{wf} \) and is not transitive.
- The relation over \( T_\Sigma \), ‘\( M \prec N \) if \( M \) is an immediate subterm of \( M' \)’ is \( \text{wf} \).
- The relation of divisibility over \( \mathbb{Z} \) is a \( \text{wf} \) preorder.

Wf relations are also called noetherian due to an important algebraic property of which the previous example is only a particular case. A commutative ring is noetherian if every infinite sequence of ideals, increasing over the subset relation \( \subseteq \), is stationary. In the case of \( \mathbb{Z} \), which is principal, an increasing sequence of ideals \( (p_0) \subseteq (p_1) \subseteq (p_2) \subseteq \ldots \) corresponds to a decreasing sequence of their generators: \( p_0 \) is divisible by \( p_1 \), which is divisible by \( p_2 \), etc. Many commonly used rings in algebra and geometry, such as polynomial rings and their quotients, are noetherian. It is a finiteness property: one of Hilbert’s theorems asserts that every algebraic set is the set of zeros of a finite number of polynomials. It is
therefore not surprising that computer science uses this notion to express the finite character of its operations.

Two general constructions of wf orders are useful: products and multisets.

Let \( \succ \), be a well-founded order over a set \( E_i \), \( i = 1, 2 \). Define the lexical order \( \succ \times \) over the product \( E_1 \times E_2 \) by \((x_1, x_2) \succ \times (y_1, y_2)\) if \( x_1 \succ_1 y_1 \) or \( x_1 = y_1 \) and \( x_2 \succ_2 y_2 \). The product \( \succ \times \) is wf if and only if \( \succ_1 \) and \( \succ_2 \) are wf.

A multiset is a set with possibly multiple occurrences of its elements. More precisely, if \( E \) is a set, a multiset over \( E \) is a mapping \( M : E \to \mathbb{N} \). For \( x \in E \), \( M(x) \) is the number of occurrences of \( x \) in \( M \). The elements of a multiset \( M \) are the \( x \in E \) such that \( M(x) \neq 0 \), written \( x \in M \). A multiset \( M \) is finite if it has a finite number of elements. We will use a setlike notation for multisets. For example, \( M = \{a, a, a, c\} \) instead of \( M(a) = 3 \), \( M(c) = 1 \), and \( M(x) = 0 \) for \( x \neq a, c \). The set of finite multisets over \( E \) is written \( \mathcal{M}(E) \). Multiset operations are defined in an analogous manner to the union, difference and intersection of sets. For example, union \( \cup \) is defined by \((M \cup N)(x) = M(x) + N(x)\).

Let \( \succ \) be a strict order relation over \( E \). Extend it to \( \mathcal{M}(E) \) by \( M \gg M' \) if \( M' \) is obtained from \( M \) by replacing an occurrence of an element \( x \) of \( M \) by a multiset of elements, each smaller than \( x \).

**Proposition 2.2** If \( \succ \) is a strict order over \( E \), then \((\mathcal{M}(E), \gg)\) is well-founded if and only if \((E, \succ)\) is well-founded.

*Well-founded induction* This proof method uses induction over a well-founded relation, generalizing induction over the integers and structural induction over terms.

**Definition 2.3** Let \( \prec \) be a relation over a set \( E \). A subset \( P \) of \( E \) is called \( \prec \)-progressive (or hereditary) if it contains an element whenever it contains its predecessors:

\[
\forall x(\forall y(y \prec x \Rightarrow y \in P) \Rightarrow x \in P).
\]

The property \( x \in P \) is also called progressive.

**Remark** A progressive subset \( P \) contains, by definition, all the minimal elements of \( E \) (\( P \) contains the predecessors of a minimal element, since there are none!) \( \square \)

**Proposition 2.3** If \( \prec \) is wf over \( E \) and if \( P \subseteq E \) is \( \prec \)-progressive, then \( P = E \).

**Proof** If \( P \neq E \), then \( A = E \setminus P \) is non-empty and so contains a minimal element \( a_0 \), which means that \( y \prec a_0 \) implies \( y \notin A \). Since \( P \) is progressive, \( a_0 \) must be in \( P \), which contradicts the assumption \( a_0 \in A \). \( \square \)
Examples

- Induction over the relation ‘m is the successor of n’ in \( \mathbb{N} \) is exactly the usual induction.
- Induction over the usual order of \( \mathbb{N} \) is called complete induction; to prove \( P(m) \), suppose \( P(n) \) for all \( n < m \).
- Induction over the relation ‘\( M < N \) if \( N \) is an immediate subterm of \( M \)’ is structural induction over terms.

Well-orders and ordinals A well-order is a well-founded total order. The underlying set is called well-ordered. An equivalent characterization is that every subset has a least element; the set \( \mathbb{N} \), with the usual order, is an example. But there are many other ways to well-order the same set: for example, \( 0 < 2 < 4 < 6 < \cdots < 1 < 3 < 5 < \cdots \) is a well-order for \( \mathbb{N} \). The concept of ordinal classifies all well-orders. An ordinal is a set (in fact, if it is non-empty, it will be a set of ordinals, hence a set of sets) which is transitive and well-ordered by the relation \( \in \). A set \( T \) is transitive if every element of \( T \) is a subset of \( T \). It is easy to construct ordinals. Starting with the empty set, \( \emptyset \), \( \{ \emptyset \} \), \( \{ \emptyset, \{ \emptyset \} \} \) and \( \{ \emptyset, \{ \emptyset \} \} \) are all ordinals. By writing them 0, 1, 2 and 3, we obtain \( 1 = \{ 0 \} \), \( 2 = \{ 0, 1 \} \) and \( 3 = \{ 0, 1, 2 \} \), so \( 0 \in 1 \in 2 \in 3 \). All the finite ordinals can be obtained from \( 0 = \emptyset \) by the successor operation \( \alpha \mapsto \alpha + 1 = \alpha \cup \{ \alpha \} \). The set of all the finite ordinals is an ordinal written \( \omega \)—this is how \( \mathbb{N} \) is constructed in set theory—which is not a successor. It is called a limit ordinal. This construction can be continued: \( \omega + 1 \) is the set of integers along with a greatest element, and \( \omega + \omega \), the set of all \( \omega + n \), is the ordinal corresponding to the well-order ‘even integers < odd integers’ over \( \mathbb{N} \).

Monotone induction

There are two ways to define the set of terms; both are formulations of an inductive definition. From above, as in ‘the smallest subset \( T \) such that ...’, or from below, using an inference system, which is inherently more constructive. These two approaches can be interpreted using an operator over \( \mathcal{P}(\Sigma^\ast) \), the set of languages with alphabet \( \Sigma \). If \( A \) is a language, define

\[
\Phi_\Sigma(A) = \bigcup_{f \in \Sigma} \{ fM_1 \ldots M_{\alpha(f)} ; M_1, \ldots , M_{\alpha(f)} \in A \}.
\]

Note that \( \Phi_\Sigma(A) \) contains the constant symbols of \( \Sigma \). The definition of \( T_\Sigma \) from above is given by

\[
T_\Sigma = \bigcap_{\Phi_\Sigma(E) \subseteq E} E.
\]

Consider a more general situation. Let \( X \) be a set and \( \Phi : \mathcal{P}(X) \rightarrow \mathcal{P}(X) \) a monotone (increasing) operator:

\[
\text{if } A \subseteq B, \text{ then } \Phi(A) \subseteq \Phi(B).
\]
Throughout this discussion, increasing will not imply strictly increasing. We will show that every monotone operator $\Phi$ has a least fixpoint $\mu\Phi$, and a greatest fixpoint $\mu\Phi$, and that there are several ways to define them, as well as to compute them.

Not all monotone operators are defined through structural induction. For example, the set of words over an alphabet $A$ can be obtained by the following rules: a word is empty, a symbol of $A$ or the concatenation of two non-empty words. Structural induction is not used, for a non-empty word may be defined through concatenation in many different ways. This inductive definition can still be translated into a monotone operator $\Phi$, where $\Phi(E) = \emptyset \cup A \cup \{uv; u, v \in E\}$, for $E \subseteq A^*$.

These constructions include the usual induction over the integers, since the latter can be obtained from a monotone induction over the operator $\Phi : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$, defined by $\Phi(A) = \{0\} \cup \{n + 1; n \in A\}$.

A subset $E$ of $X$ is closed under $\Phi$ if $\Phi(E) \subseteq E$. Consider the intersection of the subsets closed under $\Phi$,

$$\text{Ind } \Phi = \bigcap_{\Phi(E) \subseteq E} E,$$

called the set inductively defined by $\Phi$. The following result is a special case of one of Tarski's theorems.

**Proposition 2.4** If $\Phi$ is monotone, $\text{Ind } \Phi$, the set inductively defined by $\Phi$, is the least fixpoint of $\Phi$.

**Proof** Let $I = \text{Ind } \Phi$ and $E$ be the set of $E$ such that $\Phi(E) \subseteq E$. Since $\Phi$ is monotone, $I \subseteq E$ implies $\Phi(I) \subseteq \Phi(E)$, and so if $E \in \mathcal{E}$, then $\Phi(I) \subseteq E$. Hence $\Phi(I) \subseteq \bigcap_{E \in \mathcal{E}} E = I$. Conversely, $\Phi(I) \subseteq I$ implies $\Phi(\Phi(I)) \subseteq \Phi(I)$, i.e., $\Phi(I) \in \mathcal{E}$. By the definition of $I$, it follows that $I \subseteq \Phi(I)$: $I$ is therefore a fixpoint of $\Phi$. If $I'$ is another fixpoint of $\Phi$, then in particular $I' \in \mathcal{E}$, hence $I \subseteq I'$, which proves that $\text{Ind } \Phi = \mu\Phi$. 

Let $A \subseteq X$. Since $\emptyset \subseteq A$, the increasing nature of $\Phi$ implies $\emptyset \subseteq \Phi(\emptyset) \subseteq \Phi(\Phi(\emptyset)) \subseteq \ldots$. An idea familiar in analysis is to iterate towards this fixpoint. An increasing sequence of subsets of $X$ is computed by

$$\Phi^0 = \emptyset,$$
$$\Phi^{n+1} = \Phi(\Phi^n),$$
$$\Phi^\omega = \bigcup_{n \geq 0} \Phi^n.$$

The assumption that $\Phi$ is monotone does not necessarily imply that $\Phi^\omega$ is a fixpoint. The process must continue, starting with $\Phi^{\omega+1} = \Phi(\Phi^\omega)$. However, there is an important property which allows one to stop at $\omega$.

An operator is called *finitary* if its values are uniquely determined by the values taken when applying the operator to finite sets, i.e., if $x \in \Phi(A)$, there exists a
finite $A' \subseteq A$ such that $x \in \Phi(A')$. If $\Phi$ is finitary and increasing, $\Phi(A) = \bigcup \Phi(E)$, $E$ consisting of all the finite subsets of $A$.

If $\Sigma$ is finite, the $\Phi_\Sigma$ operator is finitary, which means that each inference rule has a finite set of premises. It is easy to check, using induction over $n$, that $M \in \Phi_\Sigma^n$ if and only if the judgment ‘term $M$’ has a derivation of height $n$, i.e., the term $M$ is of height $n$. The $\Phi_\Sigma$ operator can be used to generate the theorems of the inference system starting from $\Phi_\Sigma^1$, i.e., from the axioms. This generation is ‘stratified’ by the height of the derivation. Hence

$$ T_\Sigma = \bigcup_{p \geq 0} \Phi_\Sigma^p(\emptyset). $$

Note that $\Phi_\Sigma(\emptyset) = \Sigma_0 = \{ c \in \Sigma; \text{ar}(c) = 0 \}$.

**Definition 2.4** A monotone operator $\Phi$ is continuous if for every increasing sequence $(E_n)_{n \geq 0}$ of subsets of $X$,

$$ \Phi\left( \bigcup_{n \geq 0} E_n \right) = \bigcup_{n \geq 0} \Phi(E_n). $$

**Lemma 2.5** A monotone and finitary operator $\Phi$ is continuous.

**Proof** Since $\Phi$ is monotone, $\bigcup_{n \geq 0} \Phi(E_n) \subseteq \Phi(\bigcup_{n \geq 0} E_n)$. Conversely, let $x \in \Phi(\bigcup_{n \geq 0} E_n)$. Since $\Phi$ is finitary, there exists a finite $A \subseteq \bigcup_{n \geq 0} E_n$ such that $x \in \Phi(A)$. As $A$ is finite, there exists an integer $p$ such that $A \subseteq \bigcup_{0 \leq n \leq p} E_n = E_p$, the sequence $E_n$ being increasing. Hence $x \in \Phi(E_p)$. And so $\Phi(\bigcup_{n \geq 0} E_n) \subseteq \bigcup_{n \geq 0} \Phi(E_n)$. \qed

**Proposition 2.6** If $\Phi$ is continuous, then $\mu \Phi = \text{Ind} \Phi = \Phi^{1\omega}$.

**Proof** Since $\text{Ind} \Phi$ is a fixpoint of $\Phi$, it follows from $\emptyset \subseteq \text{Ind} \Phi$ that $\Phi^{1n} \subseteq \text{Ind} \Phi$, hence that $\Phi^{1\omega} \subseteq \text{Ind} \Phi$. It remains to be shown that $\Phi^{1\omega}$ is a fixpoint of $\Phi$, i.e., that $\Phi^{1\omega+1} = \Phi^{1\omega}$. By the continuity hypothesis,

$$ \Phi^{1\omega+1} = \Phi\left( \bigcup_{n \geq 0} \Phi^{1n} \right) = \bigcup_{n \geq 0} \Phi(\Phi^{1n}) = \bigcup_{n \geq 1} \Phi^{1n} = \Phi^{1\omega}. $$

\qed

In the non-continuous case, the iteration must be continued. $\Phi^{1\alpha}$ can be defined for all ordinals $\alpha$,

$$ \begin{align*}
\Phi^{10} &= \emptyset, \\
\Phi^{1\alpha+1} &= \Phi(\Phi^{1\alpha}), \\
\Phi^{1\lambda} &= \bigcup_{\mu < \lambda} \Phi^{1\mu},
\end{align*} $$


where \( \lambda \) is a limit ordinal, e.g., \( \omega, \omega + \omega, \ldots \).

What can be shown, by a cardinality argument, because \( \Phi^{1\alpha} \subseteq X \), is that there exists a smallest ordinal \( \alpha \) such that \( \text{Ind} \Phi = \Phi^{1\alpha} \). Continuity implies that \( \alpha \leq \omega \).

This transfinite iteration is not necessary in the case of continuous operators, the typical case for the usual inference systems, but it is necessary in the `dual' case of iterative approximations from above, to compute the greatest fixpoint:

\[
\begin{align*}
\Phi^{10} &= X, \\
\Phi^{1\alpha+1} &= \Phi(\Phi^{1\alpha}), \\
\Phi^{1\lambda} &= \bigcap_{\mu<\lambda} \Phi^{1\mu}.
\end{align*}
\]

Continuity is no longer sufficient to ensure that \( \Phi^{1\omega} \) is the greatest fixpoint. What is needed is an ordinal \( \alpha \) such that \( \Phi^{1\alpha} = \mu \Phi \). Concrete examples of this situation can be found in logic programming, related to the proof method of negation as failure.

These results can be formulated in the more general structure of complete lattices. A lattice is an ordered set such that every pair of elements has an upper bound and a lower bound. A lattice is complete if every subset has an upper bound.

When inductive definitions were being presented, it was stated that these can provide proof methods, be it by induction over the integers or structural induction over terms. In the case of a monotone operator \( \Phi \), one proves a property \( P \) over the set \( \text{Ind} \Phi \). If \( E_P \) is the set of \( x \in X \) satisfying \( P \), it suffices to establish that \( E_P \) is closed under \( \Phi \) to deduce that \( \text{Ind} \Phi \subseteq E_P \). One can also reason `from below' by proving that for all \( n \), \( \Phi^{1n} \subseteq E_P \), i.e,

- \( P(x) \) for all \( x \in \Phi^{10} \) (vacuously true);
- if \( P(y) \) for all \( y \in \Phi^{1n} \), then \( P(x) \) for all \( x \in \Phi^{1n+1} \).

The proof has been converted into a proof by induction over the integers. For terms, the proof would be over the height of a term. However, using integers or, more generally, ordinals (transfinite induction) is not necessary when structural or monotone induction is possible.

### 2.2.3 Examples

Terms are the natural syntactic entities to represent the objects described in this book: inference system derivations, data structures, applicative terms, and propositional calculus formulas. For the moment, only the form of these objects will be described: names will be given only cursory meanings. Later on, their semantics, or precise definitions of their meanings, will be given.
Data structures
Terms are the most important objects of symbolic computation, as well as of functional and logic programming: Lisp atoms and pairs, Prolog terms and (first-order) ML concrete types are all terms. Their abstract structure favors a clear and concise ‘algebraic’ programming style.

A type is declared by introducing a signature \( \Sigma \). The concrete values of this type are the elements of \( T_\Sigma \). At least one constant symbol is needed for the type to be non-empty.

Example The signature for integers is given by \( \Sigma = \{0, s\} \), with \( ar(0) = 0 \) and \( ar(s) = 1 \), since \( T_\Sigma = \{0, s0, \ldots, s^00, \ldots\} \cong \mathbb{N} \). The integers are written in a unary form, which, although not convenient for arithmetic computations, does represent the operational character of the integers: consider loop increment (\( i++ \) of C), iteration and recursion. In ML, this type would be declared by

\[
\text{datatype nat = zero | s of nat ;}
\]

and in Prolog by

\[
\begin{align*}
nat&(\text{zero}). \\
nat&(s(X)) :- nat(X).
\end{align*}
\]

Normally one manipulates several different sorts (or types) of objects, for example integers, strings, atoms and lists. An operation of a given arity cannot be applied to any sort of object. After all, what is the successor of a list? The rules for term formation will be restricted through the use of type declarations.

Example Two sort symbols are introduced: ‘atom’ and ‘list’, and typed constant and function symbols:

\[
a, b, c, \ldots : \text{atom}, \\
nil : \text{list}, \\
\text{cons} : \text{atom}, \text{list} \rightarrow \text{list},
\]

The rules for term formation are:

1. \( a, b, c, \ldots \) are terms of sort ‘atom’.
2. \( \text{nil} \) is a term of sort ‘list’.
3. If \( A \) is a term of sort ‘atom’, and if \( L \) is a term of sort ‘list’, then \( \text{cons}(A, L) \) is a term of sort ‘list’.

Hence \( \text{cons}(a, \text{nil}) \) is a term, but not \( \text{cons}(a, b) \). The ML declaration for the type could be

\[
\text{datatype atom = a | b} \\
\text{and list = nil | cons of atom * list ;}
\]
(one can do better in ML, using type variables; we will return to this subject), and
the Prolog one could be

\[
\begin{align*}
\text{atom(a).} \\
\text{atom(b).} \\
\text{list(nil).} \\
\text{list(cons(A,L)) :- atom(A), list(L).}
\end{align*}
\]

Definition 2.5 A many-sorted signature is a pair \((S, \Sigma)\) where

- \(S\) is a set of sort symbols;
- \(\Sigma\) is a set of typed constant and function symbols

\[
\begin{align*}
c & : s, \\
f & : s_1, \ldots, s_n \rightarrow s,
\end{align*}
\]

with \(s_1, \ldots, s_n, s \in S\) and \(n \geq 1\).

The rules for term formation are given by the following inference system, dealing
with judgments of the form ‘term \(M : s\)’, with one \(n\)-ary rule for each \(n\)-ary symbol:

\[
\begin{align*}
(c) : \quad \frac{\text{term } c : s}{\text{term } c : s} \\
(f) : \quad \frac{\text{term } M_1 : s_1 \quad \ldots \quad \text{term } M_n : s_n}{\text{term } f \; M_1 \ldots M_n : s}
\end{align*}
\]

where \(c, f \in \Sigma\) and \(c : s, f : s_1 \ldots s_n \rightarrow s\). \(M\) is a term of sort \(s\) if there exists a
closed derivation of ‘term \(M : s\)’ using this inference system, written \(\vdash \text{term } M : s\).

The set of terms of sort \(s\) is written \(T_{\Sigma, s}\), and \(T_{\Sigma}\) is defined by \(T_{\Sigma} = \bigcup_{s \in S} T_{\Sigma, s}\).

The set of these terms can be implemented in ML as a concrete type, in a
manner similar to lists. The many-sorted signature \((S, \Sigma)\) is associated with the
type definition:

\[
\text{datatype s1 = d1 and s2 = d2 and \ldots and sn = dn ;}
\]

where \(S = \{s_1, s_2, \ldots, s_n\}\) and each \(s_i = d_i\) is of the form

\[
\text{si = a | b | \ldots | f of s}_{p_1} \ast \ldots \ast s_{p_r} | \ldots
\]

where \(a, b, \ldots\) are constant symbols of sort \(s_i\), and the \(f\) are constructors whose
type is \(s_{p_1} \ast \ldots \ast s_{p_r} \rightarrow s\).

Applicative terms

Undoubtedly one of the most common operations is the application of a ‘function’
to an ‘argument’. The meaning of this operation will be given in the next chapter.
From a syntactic point of view, an applicative signature will be used. The only
symbol of arity \(\geq 1\) is the lone binary symbol, \text{App} (also written \@): terms of
this signature are called applicative terms. Application is here noted in an explicit
manner: one would write \text{App cos x} where standard mathematical usage would be
cos(x). A simplified notation will be used most of the time. Write \((M_1M_2)\) instead of \(\text{App} \ M_1M_2\) and omit unnecessary parentheses, assuming that application is left-associative:

\[
(...((M_1M_2)M_3)...M_n) \quad \text{is written} \quad M_1 \ M_2 \ M_3 \ ... \ M_n.
\]

External parentheses are removed.

Applicative terms are a sort of universal syntax for terms. In fact, a mapping \(M \mapsto M^\#\) can be defined from terms constructed over an arbitrary signature \(\Sigma\) into terms over an applicative signature \(\Sigma^\#\) containing all of the elements of \(\Sigma\) as constants, along with \(\text{App}\). A constant is translated into itself and a \(\Sigma\)-term \(fM_1...M_n\) is translated into the \(\Sigma^\#\)-term \(\text{App}...\text{App} \ fM_1^\#M_2^\#...M_n^\#\) or, using the simplified notation omitting the \(\text{App}\), into the \(\Sigma^\#\)-term \(fM_1...M_n\).

**Example** The term \(f(a, g(a, a), h(a))\) is represented in Figure 2.1 as a \(\Sigma\)-term (to the left) and as a \(\Sigma^\#\)-term (to the right).

![Diagram](image)

**Figure 2.1** Representations of a term by a tree

This mapping is clearly not a bijection, for the \(\Sigma^\#\)-term \(\text{App}(a, a)\) cannot be obtained from a \(\Sigma\)-term. The left (respectively right) leaves of \(M^\#\) are labeled by function (respectively constant) symbols of \(\Sigma\). The right nodes of \(M^\#\) correspond to subterms of \(M\). Note that this mapping of a term into an applicative term is different, at the tree level, from the usual transformation in computer science, where a tree is mapped into a binary tree, by linking each node to its ‘eldest child’ and its ‘younger sibling’.

**Formulas of the propositional calculus**

Formulas are terms constructed with connectives and atoms. Begin by introducing a signature \(\Sigma_c\) formed of connectives: here are the common ones. There are binary connectives,

- **implication** \((\Rightarrow, \text{sometimes written } \supset \text{ or } \rightarrow)\),
- **equivalence** \((\Leftrightarrow)\),
- **conjunction** \((\wedge, \text{‘and’})\), and
- **disjunction** \((\lor, \text{‘or’})\);
a unary connective,

- negation ($\neg$, 'not', sometimes written $\sim$);

and constant connectives,

- false ($\bot$) and true ($\top$).

To these connectives is added a set $A$ of atomic propositions, or atoms, or propositional variables, as constant symbols. These are written $p$, $q$, $r$, \ldots. A new signature $\Sigma = \Sigma_0 \cup A$ is generated. A term of this signature is called a proposition, or formula of the propositional calculus. The set of propositions is written $\text{Prop}[A]$.

### 2.2.4 Terms and trees

Let $M$ be a term represented by a tree. Number the arcs emanating from each node from left to right, starting with 1. Consider as an example the term represented by the tree on the left of Figure 2.1. A particular occurrence of a symbol of $M$ can be designated by the word obtained by concatenating the numbers of the arcs of the path leading from the root to the symbol occurrence: $\epsilon$, 1, 2 and 21, respectively, yield $f$, $a$, $g$ and $a$. These words are actually called occurrences. They are elements of $A^*$, where $A$ is the set of integers $\geq 1$. The prefix order will be used between occurrences: $u \leq v$ states that $u$ is an ancestor of $v$. A given occurrence $u$ of $M$ determines a node in a tree, the symbol that labels it and the subterm (subtree) which emanates from it.

For each term $M$, define

- the set $\mathcal{O}(M)$ of occurrences of $M$;
- the symbol $M(u)$, for $u \in \mathcal{O}(M)$;
- the subterm $M/u$, for $u \in \mathcal{O}(M)$;

by

1. if $M = c \in \Sigma$, then $\mathcal{O}(M) = \{\epsilon\}$, $M(\epsilon) = c$, $M/\epsilon = c$;
2. if $M = f M_1 \ldots M_n$, then

$$
\mathcal{O}(M) = \{\epsilon\} \cup \bigcup_{i=1}^n i \cdot \mathcal{O}(M_i)
$$

$$
M(\epsilon) = f \quad M/\epsilon = M
$$

$$
M(i \cdot u) = M_i(u) \quad M/i \cdot u = M_i/u.
$$

The following properties are easy to visualize on the tree representation. They are proven using induction over proof terms. See exercise 6.

**Proposition 2.7** If $M$ is a finite term, then
1. if \( u \leq v \) and \( v \in \mathcal{O}(M) \), then \( u \in \mathcal{O}(M) \);
2. if \( 1 \leq i \leq j \) and \( u \cdot j \in \mathcal{O}(M) \), then \( u \cdot i \in \mathcal{O}(M) \);
3. if \( v \in \mathcal{O}(M) \) and \( \text{ar}(M(u)) = n \), then \( u \cdot 1, \ldots, u \cdot n \in \mathcal{O}(M) \), \( u \cdot (n + 1) \notin \mathcal{O}(M) \);
4. \( \mathcal{O}(M) \) is a finite set.

These properties give a new definition of terms, as trees. The first two properties define trees to be subsets of \( A^* = \{1, 2, 3, \ldots\}^* \), i.e., languages. The third property restricts how trees can be labeled (the degree of a node = the arity of a symbol). The last property is a finiteness condition, although infinite trees and terms can also exist (Prolog II uses infinite terms).

A subset \( \mathcal{O} \) of \( A^* \) is a tree domain, or an occurrence language, if it satisfies properties (1) and (2) (prefix and ‘age’ stability). Tree terminology is used:

- \( \epsilon \) is the root of \( \mathcal{O} \);
- elements of \( \mathcal{O} \) are nodes;
- nodes of degree zero are leaves.

Note that if \( \mathcal{O} \) is a tree domain and that if \( u \in \mathcal{O} \), then \( \mathcal{O}/u = \{v/u ; v \in \mathcal{O} \} \) and \( v \geq u \) is also a tree domain.

**Example** \{\( \epsilon \), \( \epsilon, 1, 2 \), \( \epsilon, 1, 2, 11 \)\} and \( \{1, 2\}^* \) are tree domains. The last one is infinite. \( \square \)

Terms are well-labeled trees:

**Definition 2.6** Let \( \Sigma \) be a signature. A (finite or infinite) \( \Sigma \)-term is a mapping \( M : \mathcal{O} \rightarrow \Sigma \) satisfying properties (1), (2) and (3) of the preceding proposition. The term is a finite \( \Sigma \)-term if (4) is also satisfied.

The domain of \( M \), as a mapping, will be written \( \mathcal{O}(M) \). For the moment only finite terms will be considered. It is then easy to define the subterm \( M \) at occurrence \( u \) by

\[
M/u : \mathcal{O}(M)/u \rightarrow \Sigma,
\]

\[
v \mapsto M(u \cdot v),
\]

and to check that \( M/u \) is a term.

The operation complementary to subterm construction is the graft, whose inductive definition is given here. Let \( M, N \in T_\Sigma \) and \( u \in \mathcal{O}(M) \). If \( M = fM_1 \ldots M_n \), with \( n \geq 0 \), define

\[
M[\epsilon \leftarrow N] = N,
\]

\[
M[i \cdot u \leftarrow N] = fM_1 \ldots M_{i-1}M_i[u \leftarrow N]M_{i+1} \ldots M_n.
\]
$M[u \leftarrow N]$ is a term and (see exercise 7)

$$
M[u \leftarrow N](v) = \begin{cases} 
N(w), & \text{if } v = u \cdot w \text{ and } w \in \mathcal{O}(N) \\
M(v), & \text{if } u \not\preceq v \text{ and } v \in \mathcal{O}(M).
\end{cases}
$$

(2.1)

$M[u \leftarrow N]$ is obtained from grafting $N$ on to $M$ at occurrence $u$. The following properties, given as an exercise, will be used:

- $M[u \leftarrow N]/(uv) = N/v$,
- $M[u \leftarrow N][uv \leftarrow P] = M[u \leftarrow N[v \leftarrow P]]$,
- $M[u \leftarrow N]/u = M/v$ if $u \perp v$,
- $M[u \leftarrow N][v \leftarrow P] = M[v \leftarrow P][u \leftarrow N]$ if $u \perp v$,
- $M[u \leftarrow N]/v = (M/v)[u/v \leftarrow N]$ if $v \ll u$,
- $M[u \leftarrow N][v \leftarrow P] = M[v \leftarrow P]$ if $v \ll u$.

It is useful to introduce the concept of context, which allows one to consider the graft as a binary operation between a context, rather than a term, and an occurrence. A context is a term with a single hole, ready to receive the graft of a term. For example, $f(a, h(a, [ ], b))$ is a context $C$ and $C[N] = f(a, h(a, N, b))$. For a precise definition, see exercise 8.

As an example of the use of occurrences, here is a precise definition of the translation of $\Sigma$-terms into applicative $\Sigma^\oplus$-terms introduced in §2.2.3. If $M$ is a $\Sigma$-term, define the $\Sigma^\oplus$-term $M^\oplus$ by

$$
c^\oplus = c \quad \text{if } c \in \Sigma_0
$$

$$
M^\oplus(1^p) = \text{App} \quad (0 \leq p < \text{ar}(M))
$$

$$
M^\oplus(1^\text{ar}(M)) = M(\varepsilon)
$$

$$
M^\oplus/(1^\text{ar}(M) - p 2) = (M/p)^\oplus \quad (1 \leq p \leq \text{ar}(M)).
$$

### 2.2.5 Concrete syntax

It is just as useful to have a useful syntactic representation as it is to know the abstract syntax of terms. An injective mapping from $T_\Sigma$ to a particular subset of $\Gamma^*$ must be constructed. Its range must be easily recognizable and the mapping must be easily inverted. Such a mapping is called a concrete syntax for terms. A few common examples will be examined.

Prefix notation was used right from the beginning, i.e., this is Lukasiewicz's 'Polish notation':

$$
\text{pref} : T_\Sigma \rightarrow \Sigma^*,
$$

$$
M \mapsto f \cdot \text{pref}(M/1) \ldots \text{pref}(M/n),
$$

where $f = M(\varepsilon)$ and $n = \text{ar}(f)$. 
Example  Let $M$ be the term represented in Figure 2.1.

\[
\text{pref}(M) = f \cdot \text{pref}(M/1) \cdot \text{pref}(M/2) \cdot \text{pref}(M/3) \\
= f \cdot a \cdot g \cdot \text{pref}(M/21) \cdot \text{pref}(M/22) \cdot h \cdot \text{pref}(M/31) \\
= fagaaha.
\]

The problem of syntactic analysis is: given a word $x \in \Sigma^*$, is $x$ the prefix notation of a term, and, if so, of what term? The answer to this problem must be a parsing algorithm. Using exercise 9, a recursive descent (top–down) parser can be generated. On the other hand, suffix notation is well adapted to bottom–up parsing:

\[
\text{suff} : T_\Sigma \rightarrow \Sigma^*, \\
M \mapsto \text{suff}(M/1) \ldots \text{suff}(M/n) \cdot f,
\]

where $f = M(e)$ and $n = \text{ar}(f)$.

Example (cont.) $\text{suff}(M) = aaagahf$. □

Parenthesized notation inserts parentheses and commas in the middle of infix notation. What is added is called ‘syntactic sugar’. It facilitates reading by indicating the arity of symbols and corresponds to mathematical usage:

\[
\text{paren} : T_\Sigma \rightarrow \Sigma^*, \\
M \mapsto f(\text{paren}(M/1), \ldots, \text{paren}(M/n)),
\]

where $f = M(e)$ and $n = \text{ar}(f)$.

Example (cont.) $\text{paren}(M) = f(a, g(a, a). h(a))$. □

Lisp also uses parentheses, in the prefix expressions of the form (\texttt{M1} \ldots \texttt{Mn}). Syntactic sugar is not being used here, as Lisp’s functions are not always of fixed arity. The parentheses are therefore needed.

Binary notation is often written using infix notation: (2 + 3). Parentheses are necessary to avoid ambiguities, but they can be eliminated at the external level. The concrete syntax of each programming language represents terms in its own manner. Here is an example from C: (a-- ? a+a : a++).

2.2.6 Variables

The definition given for terms is not the classical one, as it omits variables. Yet variables can be found everywhere, in numerous forms, in logic and in programming. From a syntactic point of view, a distinction will be made between a free
variable, as in terms, and a bound variable, as in $\lambda$-terms. In programming, each kind of variable is associated with a computing paradigm. For cultural reasons, Pascal's sequential variable is probably the most familiar. Its behavior is very different from that of the simple applicative variable (a name designating a value, with no possibility of the latter being changed), of the functional variable of ML and of the logic variable of Prolog, all of which will be studied, as well as stream variables (designating sequences of values) of dataflow languages and architectures. In mathematics, variables are used to write down equations, in particular, fixpoint equations of the form $x = \phi(x)$. From a logical point of view, two kinds of variable will be studied: the assumption variable, the same as the $\lambda$-calculus variable, from the Curry–Howard correspondence; and the unknown variable, the basis for logic programming.

The introduction of variables will first be justified for finite terms.

Symbols and metasymbols
Since the beginning of this chapter, the notation has included explicitly declared symbols such as 0, s and nil, as well as metasymbols such as $M$ in 'let $M$ be a term'. It would be difficult to write down rules for constructing terms without using such metasymbols. One could not write such things as $+MsN$, which is not a term unless $M, N \in \Sigma$. These are metaterms, which have been used constantly without being mentioned explicitly.

Consider the definition of terms as an inference system. Note first that the writing of the inference rules uses metaterms,

\[
(+) : \frac{\text{term } M_1 \quad \text{term } M_2}{\text{term } +M_1M_2},
\]

which can be replaced by any term in the construction of derivations. Derivations whose judgments contain metaterms can also be constructed: what meaning should be attributed to these derivations?

The difference between term $0s0$ and metaterm $0sM$ can be seen in the derivations:

\[
(0) : \frac{\text{term } 0}{\text{term } 0} \quad (s) : \frac{\text{term } s0}{\text{term } s0} \quad (+) : \frac{\text{term } 0}{\text{term } 0s0} \quad (+) : \frac{\text{term } M}{\text{term } 0sM}.
\]

In the first case, there is a derivation of 'term $0s0$'. In the second case, there is a derivation of 'term $0sM$' which depends upon the assumption 'term $M$'. The symbol $\vdash$'s use is expanded, with the assumptions of the derivation written to its left:

\[
\text{term } M \vdash \text{term } 0sM.
\]
The concept of derivation becomes relative, hence useful, in logic.

Finally, when computing rules are introduced to state, for example, that 0 is the right neutral element for +, term metasymbols can be used to avoid writing an infinite number of rules between terms. Instead of writing

\[ 0 + 0 \rightarrow 0 \quad 1 + 0 \rightarrow 1 \quad 2 + 0 \rightarrow 2 \ldots, \]

write \( M + 0 \rightarrow M \).

The need for metasymbols is not unique to logic. Whatever the case may be, it is important to distinguish metasymbols from the objects they denote when the latter are themselves syntactic objects.

To form *metaobjects*, a set of metasymbols is declared explicitly, along with the initial symbols, those of the signature. Formal language definitions use this method, where the set \( A \) of *terminals*, the symbols of the language, is distinct from the set \( Z \) of *non-terminals*, or syntactic variables. The latter are the metasymbols: examples from a programming language would be *program*, *expression* and *instruction*. If the words of the language are formed over the alphabet \( A \), *syntactic forms* are words formed over \( Z \cup A \).

The same procedure is used for terms. A set \( X \) of *variable* symbols is introduced, the *arity function* \( \text{ar} \) is extended to \( \Sigma \cup X \) by assigning 0 to each element of \( X \) and the set \( T_{\Sigma}[X] = T_{\Sigma \cup X} \) is constructed. Its elements are called *terms*.

For the moment, nothing distinguishes a variable from a constant. However, the way in which variables are going to be used requires a reformulation of the inductive definition of terms:

**Definition 2.7** \( T_{\Sigma}[X] \) is the smallest subset \( E \) of \((\Sigma \cup X)^*\) such that

1. if \( c \in \Sigma \), \( \text{ar}(c) = 0 \), then \( c \in E \);
2. if \( f \in \Sigma \), \( \text{ar}(f) = n \geq 1 \), and if \( M_1, \ldots, M_n \in E \), then \( fM_1 \ldots M_n \in E \);
3. if \( x \in X \), then \( x \in E \).

The first and last cases would be treated differently, for example, to define the set of variables of a term \( M \), written as \( \text{var}(M) \):

1. \( \text{var}(c) = \emptyset \) for \( c \in \Sigma \), \( \text{ar}(c) = 0 \);  
2. \( \text{var}(fM_1 \ldots M_n) = \bigcup_{i=1}^{n} \text{var}(M_i) \) if \( f \in \Sigma \), \( \text{ar}(f) = n \);  
3. \( \text{var}(x) = \{x\} \), if \( x \in X \).

The set \( T_{\Sigma} = T_{\Sigma}[\emptyset] \) is called the set of *closed* terms.

**Substitutions**

So that variables can play the role of term metasymbol, a substitution operation must be introduced.

**Definition 2.8** A *substitution* is any function \( \theta : X \rightarrow T_{\Sigma}[X] \), which is the identity function almost everywhere, i.e., except for a finite subset of \( X \).

The finite set \( \text{dom}(\theta) = \{x \in X \mid \theta(x) \neq x\} \) is called the *domain* of \( \theta \).
If $\text{dom}(\theta) = \{x_1, \ldots, x_n\}$, $x_i$ distinct, then $\theta$ can be represented by the set of variable-term pairs $\{(x_1, \theta(x_1)), \ldots, (x_n, \theta(x_n))\}$, or by the ‘array’ $\begin{bmatrix} x_1 & \cdots & x_n \\ \theta(x_1) & \cdots & \theta(x_n) \end{bmatrix}$.

Substitution $\theta$ operates on $T_\Sigma[X]$ by

1. $\theta c = c$ for $c \in \Sigma$, $\text{ar}(c) = 0$;
2. $\theta(f M_1 \ldots M_n) = f(\theta M_1) \ldots (\theta M_n)$ if $f \in \Sigma$, $\text{ar}(f) = n$;
3. $\theta x = \theta(x)$, if $x \in X$, otherwise $x$.

This action extends $\theta$ to a function from $T_\Sigma[X]$ to itself, also called $\theta$. The third clause replaces each variable by the corresponding term and the first two clauses propagate this transformation in the term. Note also that $\theta M = M[x_1 := \theta(x_1); \ldots; x_n := \theta(x_n)]$. The interpretation of a substitution on a tree is simple: $\theta M$ is obtained by grafting the term $\theta(x_i)$ on to each occurrence of $x_i$ in $M$ (see Figure 2.2).

![Figure 2.2 Substitution](image)

In the case of a variable, $M[x := N] = M[u_1 \leftarrow N, \ldots, u_n \leftarrow N]$, where the $u_i$, $1 \leq i \leq n$, are the (independent) occurrences of $x$ in $M$.

The set of substitutions is written $\mathbb{S}$. A box containing metasymbols can now be considered to be a class of terms obtained by substitutions: for example the metaterm $+MssN$ denotes the set of terms $\theta(+:xssy)$, with $\theta$ covering $\mathbb{S}$.

When variables are introduced into inference systems, derivations can depend on assumptions. When derivations are considered as trees, their leaves can be labeled either by a zero-arity rule and by the judgment that it introduces, or by an arbitrary judgment called an assumption. The set of hypotheses of a derivation is defined in the same way as the set of variables of a term. If a judgment $j$ is an assumption in a derivation, $j$ must be interpreted as an ‘undefined’ derivation, or a ‘variable’ derivation of $j$, which can be substituted with any derivation of $j$. A derivation without assumptions is called closed.

Other kinds of variable will be introduced later: bound variables, those of the $\lambda$-calculus (the functional variable) and of predicate calculus (the logic variable). Furthermore, the variables of programming languages with assignments, such as Pascal, are themselves of a different nature. Terms, generally called expressions, can also be constructed, but the assignment operation prevents substitutions from taking place. For example, one can write $x := x + 1$, but one certainly cannot make the substitution $\begin{bmatrix} x \\ 2 \end{bmatrix}$ which would give $2 := 3$. In fact, a clearer syntax would be to write something like $x := !x + 1$, as in ML, but it is difficult to break tradition.
2.2.7 Shared representations and term implementations

To define a signature as an inference system and terms as trees allows for inductive reasoning over terms as well as for programming in a simple algebraic style. But this representation is not optimal. From an implementation point of view, it is correct to see $O(M)$ as the set of ‘addresses’ where the symbols of $M$ are stored. The definition of the set $O(M)$ of occurrences of a term $M$ is too fine. One can graft distinct terms on to each of the occurrences of variable $x$ in $f(x,x)$, even though the substitution operation, much used below, is incapable of distinguishing the two occurrences. If substitutions only are used, it suffices to have one address per variable.

More generally, subterms of a term can be shared. The address–occurrence set has a directed acyclic graph (dag) structure, instead of a tree structure. This dag can be obtained by a quotient of $M$ as a labeled tree structure, by an equivalence relation $\sim_M$ over $O(M)$ defining the sharing of subterms. This relation must satisfy the property: if $u \sim_M v$ then $M/u = M/v$. In particular,

1. if $u \sim_M v$, then $u_i \in O(M)$ if and only if $v_i \in O(M)$, and then $u_i \sim_M v_i$;
2. if $u \sim_M v$, then $u$ and $v$ are independent under the prefix order.

A uniform method of sharing is needed, valid for all terms. In particular, a sharing relation $\sim_M$ naturally defines a relation $\sim_{M/w}$ for each subterm $M/w$, defined by $u \sim_{M/w} v$ if $wu \sim_M wv$. Several degrees of sharing can be used:

- $u = v$: no sharing;
- $M(u) = M(v) \in X$: sharing of variables;
- $M/u = M/v$: maximal sharing, of all the subterms.

In the latter case, a term $M$ will be represented by a dag whose nodes are the subterms of $M$. For example, $h(h(f(a), f(a)), g(a))$ is represented by the graph

```
   h
  /|
 / |
 h a
```

To implement terms, one must be able to represent trees and graphs and implement operations to compute to $M(u)$, $M/u$ and $M/u_i$. ‘Linked’ structures are used. These can easily be written in a language with pointers, such as C. An occurrence is implemented as a pair formed of a symbol (a pointer to a string of characters) and a pointer to a sequence of occurrences:

```
struct occurrence
{
  char *symbol ;
  struct occurrence **arguments ;
}
```
With the declaration

```
struct occurrence *M ;
```

the value of $M(e)$ is $M\rightarrow\text{symbol}$ and that of $M/i + 1$ is $M\rightarrow\text{arguments}[i]$.

The term $f(a, g(a, x), h(x), h(x))$ would be represented by one of the memory structures of Figure 2.3, depending on the level of sharing: no sharing, sharing of variables or sharing of subterms.

![Diagram](https://via.placeholder.com/150)

**Figure 2.3** Three representations of $f(a, g(a, x), h(x), h(x))$

Another method of representation consists of coding them as applicative terms (§2.2.3); in this case, the constituents of a pair are either a pointer to a pair or a symbol.

### 2.3 Terms of the $\lambda$-calculus

The terms of the $\lambda$-calculus, or $\lambda$-terms, will be used to define functions. The next chapter will show how to compute using these terms. The notation is familiar in computer science: the function $x \mapsto x + 1$ is written

```
(lambda (x) (+ x 1))
```

in Lisp and
The syntactic landscape

\texttt{fn x => x+1}

in ML. It would be written \( \lambda x . (+x1) \) in a \( \lambda \)-calculus extended with the constants + and 1.

### 2.3.1 The \( \lambda \)-notation

The symbols \( \lambda \), \( (\cdot) \) and \( (\cdot) \) are introduced, along with a countably infinite set \( X \), whose elements are called \textit{variables}.

\textit{Definition 2.9} The rules for forming \textit{\( \lambda \)-calculus terms}, or \( \lambda \)-terms, follow:

1. Each variable is a term.
2. If \( x \) is a variable and \( M \) is a term, then \( (\lambda x \ M) \) is a term.
3. If \( M \) and \( N \) are terms, then \( (MN) \) is a term.

This syntax can be described by the productions:

\[
\text{term ::= variable | (\lambda \text{variable term}) | (term term)}.\]

The term \( (\lambda x \ M) \) is an \textit{abstraction} which stands for the ‘function \( x \mapsto M \)’. The term \( (MN) \) stands for the application of the ‘function \( M \)’ to the ‘argument \( N \)’. It is an \textit{application}. The syntactic equality of two terms is written \( = \). The symbol \( = \) will be reserved for semantic equality, in the same way that \( 2 + 2 = 3 + 1 \) while \( 2 + 2 \neq 1 + 3 \), the two expressions being distinct syntactically.

\textit{Example} \( x, (\lambda x x), (\lambda x y), (x(\lambda x y)) \) and \( (\lambda x (y(\lambda z (x z)))) \) are all terms. \( \square \)

Note that the same term can be used to the right (as ‘argument’) or to the left (as ‘function’) of an application.

\textit{Simplified notation} To simplify reading and writing, application is considered to be left-associative and abstraction right-associative. Simplify

\[
(((M_1 M_2) M_3) \ldots M_n) \ \text{into} \ M_1 M_2 M_3 \ldots M_n
\]

and

\[
(\lambda x_1 (\lambda x_2 \ldots (\lambda x_n M))) \ \text{into} \ \lambda x_1 x_2 \ldots x_n . M.
\]

The outer parentheses are removed.

\textit{Curryfication} A ‘function of several variables’ is defined in the \( \lambda \)-calculus by successive abstractions on its variables. Thus one can consider only ‘functions of a single variable’. This can only be done using ‘higher-order functions’, which take functions as arguments or return functions as results. This fact, discovered
Terms of the $\lambda$-calculus

by Schönfinkel in 1924 ([132]), gives the process known as currying, named after H. Curry, of functions in functional programming. For example, the function $h$ of two variables is replaced by the function $h^*$ of one variable, defined by $(h^*(x))(y) = h(x, y)$.

$\lambda$-calculi with constants As for applicative terms, one can define $\lambda$-terms constructed from a given set of constants which play the role of predefined values and functions of programming languages. The constants $0, 1, 2, \ldots, +, \ast, \text{if}, \ldots$, each with specific rules, can be introduced to write terms similar to Lisp expressions, as in $\lambda x. +(*2x)1$. These constants will sometimes be used in examples, particularly in functional programming. However, the 'expressive power' of the calculus is in no way changed by the introduction of the constants, for the pure $\lambda$-calculus, with its impoverished syntax, can represent all imaginable data structures and algorithms.

Examples The following terms are well known. The symbol $\equiv$ is also used to associate a term with a name:

$$
\begin{align*}
1 & \equiv \lambda x.x & \text{identity} \\
0 & \equiv \lambda f.f\cdot x & \text{composition} \\
1 & \equiv \lambda f.f\cdot x & \text{Church's iterators or numerals} \\
2 & \equiv \lambda f.f(\cdot f) \\
\end{align*}
$$

2.3.2 Abstract syntax

Now that the concrete syntax has been given, the manipulation of terms and their implementation in a programming language requires the introduction of an abstract syntax. There are two levels of abstract syntax: the first corresponds to a superficial analysis of a term, the second incorporates an interpretation of abstraction.

Shallow level

By analogy with $\Sigma$-terms, a $\lambda$-term is represented as a tree by considering each $\lambda x$ as a unary symbol, application as a binary symbol written ‘@’ and each variable a symbol of arity zero. Hence $M \equiv \lambda x. x(\lambda y. yx)$ is represented by the tree on the left of Figure 2.4. This representation is convenient and allows one to define the domain $O(M)$ of a term $M$, occurrences of variables (the leaves of the tree), $\lambda$ (unary nodes) and application (binary nodes), subterms $M/u$ and grafts $M[u \leftarrow N]$, for $u \in O(M)$. It is one of the abstract representations of $\lambda$-terms.

Unlike the case for finite terms, where it was necessary to manipulate arbitrary signatures, a specialized alphabet will be used here to designate symbol occurrences
in a tree: \( A = \{ \lambda, 1, 2 \} \). Let \( M \) be a term represented by a tree. Number the arcs emanating from each node: \( \lambda \) for the arc from a \( \lambda \), 1 and 2 for the left and right arcs from an application \( \circ \). A particular symbol of \( M \) is designated by the word obtained by concatenating the numbers of the arcs of the path leading from the root to this symbol.

The definitions given for finite \( \Sigma \)-terms are easily adapted to \( \lambda \)-terms:

1. variable: \( O(x) = \{ \epsilon \} \) and
   \[
   x(\epsilon) = x, \quad x/\epsilon = x;
   \]

2. abstraction: \( O(\lambda x \ M) = \{ \epsilon \} \cup \lambda \cdot O(M) \) and
   \[
   (\lambda x \ M)(\epsilon) = \lambda x, \quad (\lambda x \ M)/\epsilon = \lambda x \ M;
   \]
   \[
   (\lambda x \ M)(\lambda \cdot u) = M(u), \quad (\lambda x \ M)/\lambda \cdot u = M/u;
   \]

3. application: \( O(MN) = \{ \epsilon \} \cup 1 \cdot O(M) \cup 2 \cdot O(N) \) and
   \[
   (MN)(\epsilon) = @, \quad (MN)/\epsilon = MN;
   \]
   \[
   (MN)(1 \cdot u) = M(u), \quad (MN)/1 \cdot u = M/u;
   \]
   \[
   (MN)(2 \cdot u) = N(u), \quad (MN)/2 \cdot u = N/u.
   \]

**Example** If \( M \equiv \lambda x \cdot x(\lambda y \ .yx) \), then

\[
O(M) = \{ \epsilon, \lambda, \lambda 1, \lambda 2, \lambda 2 \lambda, \lambda 2 \lambda 1, \lambda 2 \lambda 2 \}
\]
\[
M(\lambda 2) = \lambda y
\]
\[
M/\lambda 2 = \lambda y \cdot (yx).
\]

**Binders and binding trees**

Although convenient, this representation is far from satisfactory as it does not take into account the role of \( \lambda \), which is to bind a variable. There are many other binders in mathematics and programming:

- \( \int_0^1 t \ dt \)
- \( \{ x \in R \mid x^3 - x + 1 = 0 \} \)
- \( \sup_{i \in I} (a_i) \)
- \( \forall x \ (x \geq 0) \)
- \( \) procedure \( A(x) \) \; begin \( x \) \; end \)

In each case, the binder introduces a variable and binds it. This mechanism is distinguished by
• the possibility of renaming bound variables, as is typical in mathematics,

\[ \int_0^1 t \, dt = \int_0^1 u \, du, \]

with the risk of incorrect renaming (clash)

\[ \int_0^1 (t + u) \, dt \neq \int_0^1 (u + u) \, du; \]

• the concept of scope, well known since the introduction of Algol 60 and its block structure.

Here are definitions for the \( \lambda \)-calculus. Let \( M \) be a term represented by a tree and \( u \in O(M) \) be an occurrence of the variable \( x \). Trace back in the tree from a leaf \( u \) to the root \( \epsilon \). There are two cases:

• If \( \lambda x \) is not encountered on this path, \( u \) is a free occurrence of variable \( x \) in \( M \).
• Otherwise, \( u \) is a bound occurrence of \( x \). The first encountered occurrence of \( \lambda x \) on this path is the binder of \( u \). The number of \( \lambda y \), where \( y \neq x \), which must be crossed before reaching the binder of \( u \) is the binding height of \( u \).

Example  The binding height is written as an exponent on the bound variables of the term:

\[ M \equiv \lambda x . x^0(\lambda y . y^0x^1). \]

An arc to the tree representing \( M \) between each bound occurrence and its binder, then remove the names of the bound occurrences. The result is a binding tree (see Figure 2.4): it is a tree to which arcs attaching a leaf \( u \) with a unary node \( v < u \) have been added. This kind of representation is the second level of abstract syntax for \( \lambda \)-terms. A variant simpler to program will be given by replacing the bindings by integers.

![Figure 2.4 Concrete and abstract syntaxes of \( \lambda \)-terms](image-url)
De Bruijn Terms
Since bindings can be constructed from the binding heights, a purely arborescent representation is obtained by noting its binding height at each occurrence.

Example \( \lambda x \cdot x \) and \( \lambda xy \cdot x \) are represented by \( \lambda 0 \) and \( \lambda (\lambda 1) \).

This form, invented by de Bruijn in 1972 ([35]), can easily implement \( \lambda \)-terms in any programming language which can be used to manipulate trees. It was introduced by de Bruijn to ease the implementation of Automath, a language, or, rather, a family of languages, designed for the writing and proving of mathematical proofs, and based on the typed \( \lambda \)-calculi. Note that the implementation of languages with lexical scoping also uses bindings, represented by integer-valued fields, or lexical addresses, to reach variables which are not local to a procedure. The same holds true for rational control structures (loops with exit).

The free occurrences of a variable could be simply represented by its name, as in \( x(\lambda 0) \) for \( x(\lambda y y) \). However, to have a simple notation which uses only integers, all free variables are numbered, as in

\[ X = \{ v_0, v_1, v_2, \ldots \} , \]

and the free occurrence \( u \) of variable \( v_i \) is represented by the integer \( i + p \), where \( p \) is the abstraction depth of \( u \), i.e., the number of \( \lambda \) encountered in the path leading from \( u \) to the root \( \varepsilon \), ends included.

Example The term \( \lambda x \cdot (\lambda y \cdot yv_4)(v_4x) \) is represented by

\[ \lambda (\lambda (0 6)(5 0)) \] instead of \( \lambda (\lambda (0 v_4)(v_4 0)) \),

the first occurrence of \( v_4 \) being at abstraction depth 2 (\( 6 = 4 + 2 \)), and the second occurrence at depth 1 (\( 5 = 4 + 1 \)): it is the left-hand term in Figure 2.5 on page 38.

An occurrence of integer \( i \) corresponds to a free variable if \( i \) is greater than or equal to its abstraction depth.

\( \lambda \)-terms are therefore represented as finite terms, but over an infinite signature, formed of a binary application symbol, \( \odot \), a unary abstraction symbol, \( \lambda \), and of all the integers as constant symbols. These finite terms are called de Bruijn terms (or trees). Their set is written \( \Lambda \). Here is a concrete syntax, analogous to the one previously given for \( \lambda \)-terms:

\[
\text{term ::= integer} \mid \lambda(\text{term}) \mid (\text{term term}).
\]

This representation is a simple and correct implementation of \( \lambda \)-terms. It also defines the abstract syntax of terms through a translation of terms from a concrete notation, the elements of \((X \cup \{\lambda, (,)\})^*\), to de Bruijn terms. Two terms are \( \alpha \)-convertible if they have the same representation as de Bruijn terms. This relation is
an equivalence relation. So, from now on, terms will be treated up to \( \alpha \)-equivalence, i.e., as elements of \( \Lambda \), but they will be written in the standard manner, e.g. \( \lambda x. x \equiv \lambda y. y \in \Lambda \).

In fact, the relation of \( \alpha \)-convertibility is the first step towards the semantics of the \( \lambda \)-calculus. It gives the precise meaning of the bound variables. That this can be done purely at the syntactic level is impressive.

It remains to be shown how a concrete representation of a term can be produced from a de Bruijn term. A variable is free in a term \( M \) if it has at least one free occurrence in \( M \) (it can also have bound occurrences). The set \( \text{var}(M) \) of free variables of \( M \) can be defined by induction over the terms:

\[
\begin{align*}
\text{var}(x) &= \{x\}, \\
\text{var}(\lambda x M) &= \text{var}(M) \setminus \{x\}, \\
\text{var}(MN) &= \text{var}(M) \cup \text{var}(N).
\end{align*}
\]

A term \( M \) is closed if \( \text{var}(M) = \emptyset \). The term \( M \) is then called a combinator.

A term using concrete syntax can be produced from a binding tree by passing names from the root to the leaves:

- The same variable name is associated with all the leaves linked to the same occurrence of \( \lambda \).
- If a subterm which is an abstraction contains free variables, the head-\( \lambda \) should be associated with a different variable name.

### 2.3.3 Graft and substitution

The main operations over \( \lambda \)-terms are grafting and substitution. Unlike those over \( \Sigma \)-terms, each of these operations is defined for an appropriate abstract representation.

**Graft**

The graft \( M[u \leftarrow P] \) of a term \( P \) on to an occurrence \( u \) of \( M \) is defined for the tree representations of \( M \) and \( P \), as if they were \( \Sigma \)-terms. Define

\[
\begin{align*}
M[e \leftarrow P] &= P, \\
(MN)[1 \cdot u \leftarrow P] &= M[u \leftarrow P] N, \\
(MN)[2 \cdot u \leftarrow P] &= MN[u \leftarrow P], \\
(\lambda x M)[\lambda \cdot u \leftarrow P] &= \lambda x (M[u \leftarrow P]).
\end{align*}
\]

This operation can imply the 'capture' of a free variable of \( P \) by a \( \lambda \) of \( M \).

**Example** Let \( P \equiv K x \), the 'constant function of value \( x \)', and \( M \equiv \lambda x N \), the 'function \( x \mapsto N \)'. Then \( M[\lambda \leftarrow P] \equiv \lambda x (K x) \), the 'function \( x \mapsto K x \), the
constant function of value \( x' \). The free variable \( x \) was bound by the graft, while it was free in \( P \).

In programming, the graft of code is called a \textit{macro}: this is what is done by the C preprocessor when handling \texttt{#define}'s. As for \( \Sigma \)-terms, grafting can also be defined using the concept of context.

\textit{Substitution}

On the other hand, \textit{substitution} of a term \( N \) for the free occurrences of variable \( x \) in \( M \) is done at the level of binding trees representing \( M \) and \( N \). The idea is to graft the binding tree of \( N \) to each of the free occurrences of \( x \) in the binding tree of \( M \); the result is a new binding tree. If one starts with concrete representations of \( M \) and \( N \), constructing their binding trees essentially means renaming variables to avoid capture.

For de Bruijn terms \( M \) and \( N \), let \( u_1, \ldots, u_k \) be the occurrences in \( M \) representing the same free variable, i.e., if \( M(u_i) = n_i \) and if \( p_i \) is the abstraction depth of \( u_i \), then \( n_i - p_i = m \geq 0 \) is independent of \( i \) and its occurrences represent the free variable \( v_m \). Consider the terms \( N^{+p_i} \) obtained by incrementing the occurrences of free variables in \( N \) by \( p_i \). Then

\[
M[v_m := N] = M[u_1 \leftarrow N^{+p_1}] \ldots [u_k \leftarrow N^{+p_k}].
\]

\textit{Example} To the two occurrences of free variable \( v_4 \) are grafted the terms \( 12 = 10^{+2} \) and \( 11 = 10^{+1} \), as seen in Figure 2.5.

\[
\begin{align*}
M & \equiv \lambda \\
M[v_4 := v_{10}] & \equiv \lambda \\
\end{align*}
\]

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{figure2.5.png}
\caption{Substitution over de Bruijn terms}
\end{figure}

While grafts can be considered to be symbolic manipulations of code, it is the mechanism of substitution which is basic to (functional or not) programming to implement parameter passing during a function or procedure call. This mechanism, defined by a relation between terms, is called \( \beta \)-reduction and will be studied in the next chapter. Its basis is the contraction of redexes.
### 2.3.4 Redexes

Two kinds of terms have already been presented: abstractions \((\lambda x \, M)\) and applications \((MN)\). A term of the form \((\lambda x \, M)N\) is called a redex: it is the combination of an abstraction and an application. The role of these terms is important enough for functional languages to have their own notation for them. For example, in Standard ML, two notations are possible:

\[
\begin{align*}
& (\text{fn } x \Rightarrow M) \, N \\
& \text{let } \text{val } x = N \text{ in } M \text{ end}
\end{align*}
\]

The last notation, due to Landin, is quite natural and means that \(\lambda\) can often be avoided.

The basic mechanism of the \(\lambda\)-calculus and of functional programming is the contraction of redexes. The transformation of the redex \((\lambda x \, M)N\) into \(M[x := N]\) is called a \(\beta\)-contraction. The term \(M[x := N]\) is the result of contracting the redex \((\lambda x \, M)N\). This transformation describes the passing of an argument to a function.

**Example**  With \(I \equiv \lambda x. \, x\), the redex \(IM\) contracts to \(M\).

For nested redexes, it will be convenient to do simultaneous contractions (equivalent to successive contractions, as there can be no capture of variables):

\[
(\lambda x_1 \ldots x_p \, M)N_1 \ldots N_p \quad \text{contracts to} \quad M[x_1 := N_1] \ldots [x_p := N_p].
\]

For example, with \(2 \equiv \lambda f \, x. \, f(f \, x)\), \(2PQ\) contracts to \(P(PQ)\).

A term for which no subterm is a redex is called normal. The set of occurrences of subterms of \(M\) which are redexes is written \(\mathcal{O}_{\text{red}}(M)\). Then \(M\) is normal if and only if \(\mathcal{O}_{\text{red}}(M) = \emptyset\).

### 2.4 Formal terms

Infinite terms appear in computer science as naturally as do finite terms. In programming, they appear as data structures: streams in functional programming, trees of Prolog-II. There are certain problems whose solutions, in general, require infinite terms. One can sometimes arrange to consider only finite terms. The situation would be analogous to that in algebra where, to compute the real roots of a second-order polynomial, one would insist that its discriminant be positive, and insist that the other equations are 'impossible'. Why restrict oneself artificially when there is a simple, more general method?

Recall the definition of terms. To avoid confusion, these will be called formal terms, be they finite or infinite, alluding to the formal series of algebra. Tree domains are subsets of the free monoid \(A^*\) stable under prefixes and age, with \(A = \{n \in \mathbb{N}; n \geq 1\}\). The monoid \(A^*\) is itself an example of an infinite tree domain.
Definition 2.10 A formal \( \Sigma \)-term is a mapping \( M : \mathcal{O} \to \Sigma \) such that \( \mathcal{O} \) is a tree domain, and if \( u \in \mathcal{O} \) and \( \text{ar}(M(u)) = n \), then \( u \cdot 1, \ldots, u \cdot n \in \mathcal{O} \) and \( u \cdot (n+1) \notin \mathcal{O} \). Their set is written \( T^\infty_\Sigma \).

As \( T^\infty_\Sigma \) is generally uncountable, this set appears to be of no interest to computer science, since it can deal only with ‘finite’ objects. However, all programming languages use ‘real’ numbers without any problem! There are therefore two solutions: one is to work with ‘floating point numbers’ which are truncated reals and designate intervals; the other is to work with rationals (these are often only decimal fractions, but there are languages such as Lisp and CAML which do exact rational arithmetic). Both of these solutions are used to give a finite representation for an infinite term, ‘exact’ for rational terms or ‘approximated’ in general.

In this book, the most important application of rational terms will be the resolution of equations between terms, the problem of unification. Another application in computer science concerns process algebras. ‘Irrational’ infinite terms appear when expanding arbitrary, recursive set of functions: it suffices to approximate them enough to compute the values of the functions.

### 2.4.1 Approximations

An infinite term can be approximated by finite terms by truncating it. Start by adding a new constant symbol \( \Omega \) to signature \( \Sigma \) and construct a ‘truncating to depth \( n \)’ operator \( \tau_n \). If \( M \) is a formal \( \Sigma \cup \{\Omega\} \)-term, then \( \tau_n(M) \) is a finite \( \Sigma \cup \{\Omega\} \)-term of domain \( \{u \in \mathcal{O}(M) ; |u| \leq n\} \), defined by

\[
(\tau_n(M))(u) = \begin{cases} 
M(u) & \text{if } u \in \mathcal{O}(M) \text{ and } |u| < n \\
\Omega & \text{if } u \in \mathcal{O}(M) \text{ and } |u| = n.
\end{cases}
\]

These \( \Sigma \cup \{\Omega\} \)-terms are called partial terms by analogy with partial functions. The meaning of symbol \( \Omega \) reflects a semantics of the computation: it designates an undefined value, or, rather, a value which has yet to be defined, which could later be substituted with a term at a later stage of the computation, to obtain a better approximation of an infinite term. This is a case where a purely syntactic construction allows one to describe a semantic behavior.

The simplest method of introducing approximation consists of defining a distance, actually an ultrametric, between partial formal terms by

\[
d(M, N) = \begin{cases} 
0 & \text{if } M = N \\
2^{-p} & \text{if } p = \inf \{k; \tau_k(M) \neq \tau_k(N)\}.
\end{cases}
\]

In particular, \( d \) is a distance over the set \( T^\infty_\Sigma \) (and over \( T^\infty_{\Sigma \cup \{\Omega\}} \)), thereby allowing one to reconstruct \( T^\infty_\Sigma \) (and \( T^\infty_{\Sigma \cup \{\Omega\}} \)). One shows in fact that \( T^\infty_{\Sigma \cup \{\Omega\}} \) is the completion of the metric space \( (T^\infty_{\Sigma \cup \{\Omega\}}, d) \).
Another concept of approximation, less artificial, arises from the order relation \( \preceq \Omega \), from which one can also construct \( T^\infty_{\Sigma \cup \{\Omega\}} \), making it into a 'complete' ordered set. Here is its inductive definition over finite terms:

\[
\Omega \preceq \Omega \quad M \quad \text{for all } M \\
\alpha M_1 \ldots M_r \preceq \Omega \quad \alpha N_1 \ldots N_r \quad \text{and } M_i \preceq \Omega \quad N_i \quad \text{for all } i.
\]

Since the constant \( \Omega \) is only an intermediary technique, the infinite terms of interest are the maximal infinite elements of \( T^\infty_{\Sigma \cup \{\Omega\}} \), i.e., those having no occurrence of \( \Omega \).

### 2.4.2 Rational terms

Rational terms are described similarly to rational numbers: in the same way that \( 2/3 \) is the solution of \( 3x = 2 \), the rational term \( f^\omega = fff \ldots \) is the solution of \( f(x) = x \). As process trees, they represent sequential programs. They are the solutions of equations, as programs with terminal recursivity. They can be represented by finite terms over a signature extended with a unary iteration symbol (\texttt{loop}) and an infinite number of 0-ary symbols for leaving loops (\texttt{exit n}).

Consider the set of equations over terms

\[
(E) : x_i = t_i, \quad t_i \in T_\Sigma[x_1, \ldots, x_d], \quad (1 \leq i \leq d).
\]

It is supposed that no \( t_i \) is a variable and that the variables \( x_i \) are all distinct. In Chapter 5, where equations of the form \( s = t \) will be studied, such a system will be referred to as reduced. The solution is an \( n \)-tuple of infinite terms, all elements of \( T^\infty_\Sigma \). By definition, these solutions are rational terms. The existence of a solution is easy to prove. In fact, associate with the system of equations an operator \( \Phi \) over \( (T^\infty_\Sigma)^d \) (or over \( (T^\infty_{\Sigma \cup \{\Omega\}})^d \)) defined by

\[
(M_1, \ldots, M_d) \mapsto (t_1[x_i := M_i], \ldots, t_d[x_i := M_i]).
\]

This operator has a least fixpoint, whose existence comes from general theorems—the fixpoint of a contracting operator in a complete metric space, here \( (T^\infty_\Sigma)^d \), or else the fixpoint of a continuous operator in a complete ordered set, here \( (T^\infty_{\Sigma \cup \{\Omega\}})^d \). In the last case, the fixpoint is the least upper bound of the \( \Phi^n(\Omega) \). Chapter 4 will elaborate on the properties of complete orders.

Solutions can easily be drawn by hand: draw the \( d \) terms \( M_i \) as trees and connect the leaves labeled by variables with the corresponding terms: the result is a graph, a perfectly finite object, but which no longer has the structure of a term. By choosing one of the variables and by making the appropriate substitutions for the graph pointed to by a variable, an infinite term is produced:

A rational term can also be considered to be a formal term whose set of subterms is finite, each subterm being able to appear an infinite number of times: if a term
has a finite number of subterms, a system for which it is a solution can be obtained by associating a variable with each subterm.

To represent the finite graph or the infinite term by a finite term, the signature is extended by defining

\[ \Sigma^* = \Sigma \cup \mathbb{N} \cup \{\ast\}. \]

This countable signature is similar to that defining de Bruijn terms in \( \lambda \)-calculus. For the latter, a constant \( n \) represents a variable bound by a \( \lambda \) of height \( n \), if \( n \) is less than the abstraction depth. In \( \Sigma^* \), the unary symbol \( \ast \) plays the same role as \( \lambda \) with integer constants, called branching indices. The free/bound distinction here becomes exit/iteration, thanks to the following definitions.

Let \( M \in T_{\Sigma^*}[X] \) and \( u \in \mathcal{O}(M) \). The \( \ast \)-depth of \( u \) is the number of occurrences \( v \) of \( M \) such that \( M(v) = \ast \) and \( v < u \). If \( p \) is the \( \ast \)-depth of \( u \) and \( M(u) = n \in \mathbb{N} \), then the difference \( n - p \) is the escape value of \( u \). If \( n - p < 0 \), \( u \) is an iteration leaf and \( n \) is its iteration height, i.e., the number of \( \ast \) between \( u \) and the root \( c \) to reach the \( \ast \) to which \( u \) is bound. Otherwise it is an exit leaf. Constant occurrences other than indices and variable occurrences are also exit leaves. A \( \Sigma^* \)-term is \( \ast \)-closed if all the indices are iteration indices.

**Example** Figure 2.6 represents a rational term in graph form (to the left) and as a finite \( \Sigma^* \)-term (to the right).

![Figure 2.6 Two forms of a rational term](image)

The result of exercise 16 can easily be extended to define a mapping \( M \mapsto M^* \) of \( T^\infty_{\Sigma}[x_1, \ldots, x_n, y] \) to \( T^\infty_{\Sigma}[x_1, \ldots, x_n] \), variables \( x_1, \ldots, x_n \) being considered as constants: \( M^* \) is the limit of sequence \( M, M; M, (M; M); M, \ldots \), where \( M; N = M[y := N] \).
For each $\star$-closed finite $\Sigma^*$-term $M$, an infinite $\Sigma$-term $\delta(M)$ can be defined by induction on the height of $M$. When $M(\epsilon) \in \Sigma$, define

$$\delta(fM_1 \ldots M_n) = f \delta(M_1) \ldots \delta(M_n),$$

for $f \in \Sigma$.

Let $M = \star N \in T_{\Sigma^*}[x_1, \ldots, x_n]$ be a $\star$-closed finite term: the indices of $M$ are all iteration indices, by definition. Let $y$ be a new variable and $N'$ be the term obtained by grafting $y$ on to each of the occurrences of an exit index of $N$ (these are the indices $p$ at $\star$-depth $p$ in $N$): $N'$ is a $\star$-closed of height inferior to that of $M$. Let $\delta(N')$ be constructed by induction, and

$$\delta(M) = \delta(N')^\star,$$

using the operation $M \mapsto M^\star$ defined above (see exercise 16). The formal term $\delta(M)$ is the solution to $y = N'$.

Solving systems of equations whose right-hand sides are in $T_{\Sigma}[X]$ produces systems of equations whose right-hand sides are in $T_{\Sigma^*}[X]$. Solving them will therefore be done in this more general setting, where $M_i \in T_{\Sigma^*}[x_1, \ldots, x_d]$.

Fix a principal variable $x_1$, e.g., in C, the main function, and eliminate the other variables using $x_1$. The same technical problems are met as in the $\lambda$-calculus when a substitution must be made over de Bruijn terms: an index incrementing operator, written $M \mapsto M^+$, is used: it replaces each variable $x$ (a leaf) by $x^+$ and increments each exit index by 1.

An equation $x = M$ is solved by $x = \star M'$, where $M'$ is obtained from $M$ by replacing each occurrence of $x$ by 0, each other variable $y \neq x$ by $y^+$ and by incrementing each of the exit indices.

**Example** Consider the set of equations ($s$ is a constant):

$$x = p(a(x), b(y)), \quad y = q(c(x), d(s)).$$

The second equation is substituted into the first one:

$$x = p(a(x), b(q(c(x), d(s)))),$$

which is solved by

$$x = \star p(a(0), b(q(c(0), d(s)))),$$

By solving first for $x$,

$$x = \star p(a(0), b(y^+)),$$

is obtained. When substituted into $y$'s definition, it gives

$$y = q(c(\star p(a(0), b(y^+))), d(s)),$$

which is solved by (since $0^+ = 1$)

$$y = \star q(c(\star p(a(0), b(1))), d(s)).$$

Note that the solution to a system whose right-hand side is in $T_{\Sigma}[X]$ is a $\star$-closed $\Sigma^*$-term.
2.4.3 Application to sequential processes

Consider a signature formed with binary symbols \( p, q, \ldots \), with unary symbols \( a, b, \ldots \) and a zero-arity symbol \( \text{stop} \). The binary symbols are tests \( p(x, y) \) meaning ‘if \( p \) then \( x \) else \( y \)’: a test is binary because it has two continuations. The unary symbols are atomic actions, which take their continuation as argument. The \( \text{stop} \) symbol designates the end (no continuation). A sequential program defines an infinite term over this signature, each branch designating a possible behavior. For example, the program which computes the greatest common divisor (gcd),

\[
\text{while } x \not< y \text{ do }
\begin{align*}
&\text{if } x \geq y \text{ then } x := x - y \text{ else } y := y - x \text{ end if} \\
&\text{end do ;} \\
&x := x
\end{align*}
\]

corresponds to the infinite term

\[
\begin{array}{c}
p \\
\downarrow \\
q \\
\quad \downarrow \\
a \\
\downarrow \\
b \\
\quad \downarrow \\
\text{stop}
\end{array}
\]

where \( p \) and \( q \) denote the tests \( x <\!\!\!\!\!\!\not< y \) and \( x \geq y \), and \( a, b \) and \( c \) the actions \( x := x - y \), \( y := y - x \) and \( r := x \). The same infinite term would have been obtained for the program

\[
1 : \text{if } x <\!\!\!\!\!\!\not< y \text{ then }
\begin{align*}
&\text{if } x \geq y \text{ then } x := x - y \text{ ; goto 1} \\
&\text{else } y := y - x \text{ ; goto 1} \\
&\text{end if}
\end{align*}
\]

\[
\text{else } r := x
\]

\[
\text{end if}
\]

and it is the same computation process (the same ‘algorithm’) which is executed. These terms describe the computations independently of the form of the programs. The general problem for structures denoting sequential processes is to obtain a finite notation for the infinite terms. The situation is similar for formal languages, which generally denote infinite sets, but which must be defined with finite means, for example with grammars or automata. This is possible if one restricts oneself to certain classes of languages, for example rational and algebraic languages. This is also possible with process terms, which are rational terms.

Adding variables to the signature formed over tests, actions and \( \text{stop} \) allows one to have variables in the leaves instead of just \( \text{stop} \). One can then substitute a variable with any other process, and the latter will be seen as a branch by other processes. To these process variables correspond, in a programming language,
labels, or, equivalently, the names of procedures with no parameters. Their use allows one to write programs as systems of equations: they are recursive definitions of procedures with terminal recursivity. For example, the above gcd program can be written abstractly as follows:

\[ x = p(q(a(x), b(x)), c(\text{stop})). \]

The solution to this equation is an infinite term, whose most natural finite notation is to indicate looping by a link from its leaves \( x \) to the root, which could be written as follows:

\[ x = \text{loop } p(q(a(\text{continue}), b(\text{continue})), c(\text{stop})), \]

by adding symbols loop and continue to the signature.

To have finite terms denote these processes, the signature has to be extended: \( * \) is an interaction, while \text{true} in Pascal or loop in Ada. Placed inside \( p \) iterations, a branching index \( n < p \) means looping on the \((n+1)\)-th \( * \) above \( n \), i.e., escaping \( n \) loops containing this index. If \( n \geq p \), the exit is equivalent to \text{stop}. This is why 0 is equated with \text{stop} and that \text{exit}(n) usually denotes the branching index \( n \) (loops can be named in Ada, which allows one to write \text{exit} \text{identifier}).

Very few languages use these general structures. The \text{while} loop is the most popular iteration structure. It was chosen for reasons of readability (the condition appears at the front of the loop) and ease of compilation. It has even been associated with a model of 'structured programming'. Yet, it is not possible, in general, to transform a program with branching into a program with \text{while} without introducing supplementary Boolean variables. It is therefore only a means of structuring control if one is willing to modify the environment (new variables) and the process (new tests, new assignments). It is a global modification of the \text{algorithm}, which requires new memory and computation resources, and not just a simple transformation of the program.

The formalization of processes just presented, although simplified, isolates the 'control' and distinguishes programs and \text{algorithms}. It is quite remarkable that one can avoid sequencing (;) as one of the instructions of our language, even though it is the characteristic operation of sequential programming!

\textbf{Exercises}\footnote{Hints to the exercises labeled with a \( \star \) can be found at the end of the book.}

1. Let \( \leq \) be the prefix order over \( A^* \). Does it have a smallest element? a greatest element? What is the set of lower bounds of a word? Does each pair of words have a least upper bound? a greatest lower bound?

2.\( \star \) If \( \leq_A \) is a total order over alphabet \( A \), the total (lexical) order \( \leq^* \) over \( A^* \) is defined by: \( u \leq^* v \) if \( u \) is a prefix of \( v \), or if \( u = wiv' \) and \( v = wjv' \) with
i, j ∈ A and i ≤ j (w is the maximal prefix common to u and v). Find a sufficient and necessary condition over A so that the set of lower bounds of a word is finite. Does every word have a successor? Is every word the successor of another one?

3. Show that \( K(\Phi) = \bigcup_{E \subseteq \Phi(E)} E \) is the greatest fixpoint of \( \Phi \), and that if one defines \( \Phi^*(E) = X \setminus \Phi(X \setminus E) \), then \( K(\Phi) = X \setminus \text{Ind} \Phi^* \). Does the continuity of \( \Phi \) ensure that \( K(\Phi) = \text{Ind} \Phi^* \)?

4. Let \( X = E \times E \) and \( R \subseteq X \) be a binary relation over \( E \). Define \( \Phi \) such that \( \text{Ind} \Phi \) is the transitive closure of \( R \). Show that if \( E \) is a finite set, \( \text{Ind} \Phi = \Phi^1 \) for some integer \( n \).

5. Define binary trees with leaves labeled by atoms (the directed pairs of Lisp) and binary trees with nodes labeled by atoms.

6.* Show that \( O(M) \) satisfies the four properties of proposition 2.7.

7. Show that \( M[u ← N] \) satisfies the four properties of proposition 2.7.

8. Since a context \( C \) determines a mapping \( M ⇀ C[M] \), give an inductive definition of contexts as mappings from \( T_{\Sigma}[X] \) to itself. Show also that every context can be designated by a term over a certain signature \( \Sigma_c \) (conversely?). To each of \( M \in T_{\Sigma}[X] \) and \( u \in O(M) \), associate a context \( M_u \). Show how to graft a context on to another context.

9. Let \( x = x_1 \ldots x_n \in \Sigma^* \). Define integers \( r_i, 0 ≤ i ≤ n \), in the following way: after having read the word \( x_1 \ldots x_i \), the integer \( r_i \) is the number of constants which must be concatenated to the right of \( x_1 \ldots x_i \) to obtain the prefix notation of a term; in particular \( r_0 = 1 \). For the word \( fagaaha \), with \( \ar(f) = 3, \ar(g) = 2 \) and \( \ar(h) = 1 \), \( (r_i)_{0 ≤ i ≤ n} = (1, 3, 2, 3, 2, 1, 1, 0) \). Show that \( r_i = r_{i-1} + \ar(x_i) - 1 \) for \( 1 ≤ i ≤ n \), and that \( x \) is the notation for a term if and only if \( r_i > 0 \) for \( 1 ≤ i < n \) and \( r_n = 0 \).

10. Write in C the following functions manipulating terms: creation, head, subterms and equality.

11.* Show that every \( \lambda \)-term \( M \) is in one of the following forms:

\[
\lambda x_1 \ldots x_n . x M_1 \ldots M_m \quad n, m ≥ 0,
\lambda x_1 \ldots x_n . (\lambda x M_0) M_1 \ldots M_m \quad n > 0, \quad m ≥ 1.
\]

12.* Define the set \( \text{var}(M) \subseteq \{v_0, v_1, \ldots \} \) when \( M \) is a de Bruijn term.

13. Find \( T_{\Sigma} \) and \( T_{\Sigma}^\omega \) for the following subsignatures of arithmetic: \( \Sigma_1 = \{0\} \), \( \Sigma_2 = \{s\} \), \( \Sigma_3 = \{0, s\} \) and \( \Sigma_4 = \{+\} \).
14. Show that $\mathcal{T}_\Sigma^\infty$ is uncountable if and only if $\Sigma$ contains at least two symbols whose sum of arities is $\geq 2$. Find similar conditions for $\mathcal{T}_\Sigma^\infty$ to be countable, finite, empty or equal to $T_\Sigma$.

15. Show that $M \leq_{\Omega} N$ if and only if for every occurrence $u$ of $M$, $M(u) = N(u)$ or $M(u) = \Omega$. Deduce that $M \leq_{\Omega} N$ is equivalent to the existence of $u_1, \ldots, u_k \in \mathcal{O}(M)$ and of terms $P_1, \ldots, P_k$ such that $M(u_i) = \Omega$ and $N = M[u_1 \leftarrow P_1, \ldots, u_k \leftarrow P_k]$.

16. Show that the set $\mathcal{T}_\Sigma^\infty[y]$, along with the operation $\cdot$ defined by $M; N = M[y := N]$, is a monoid. Let $M^0 = y$ and $M^{n+1} = M^n; M$. Show that the sequence $(M^n)_{n \geq 0}$ is a Cauchy sequence. Let $M^* = \lim_{n \to \infty} M^n$. Show that $M^* \in \mathcal{T}_\Sigma^\infty$.

**Bibliographic notes**

Formal languages are studied in [65]. For a deep study of infinite terms, see B. Courcelle’s article [27]. The representation of rational terms by finite terms is due to G. Cousineau [28], who saw their usefulness for studying control structures in sequential programming.
Chapter 3

Reduction

3.1 The $\lambda$-calculus

The concept of function used in contemporary mathematics, as a particular case of a relation—a set of pairs—is inadequate for computer science. Apart from the special case of a function defined over a finite set, for which a finite representation of its extension is possible, it is necessary to represent functions by their intensions, i.e., by computation rules. A return is thus made to the concepts of the nineteenth century. Based on this idea, the $\lambda$-calculus was introduced by A. Church in 1933 [14], and developed by B. Rosser and S. Kleene.

The $\lambda$-calculus has today become the paradigm of functional programming, after having been, first, an attempt to mechanize logic, then a general theory of functions, and finally, in the hands of computer scientists, a low-level programming language.

It is also used regularly to experiment with new computing paradigms, where certain properties, such as termination and confluence, play a crucial role comparable to that of the fundamental principles of physics.

3.1.1 $\beta$-reduction

Recall that a $\lambda$-calculus term is either a variable, an application or an abstraction. A redex is an application whose left term is an abstraction. The result of contracting redex $(\lambda x \, M)[N]$ is the term $M[x := N]$.

To explain the functional character of $\lambda$-calculus terms, a new relation, called $\beta$-reduction, based on the contraction of redexes, is introduced. In fact, several binary relations will be introduced: immediate $\beta$-reduction ($\rightarrow_\beta$), $\beta$-reduction ($\ast_\beta$) and $\beta$-conversion ($\equiv_\beta$). Each of these definitions is given using an inference system containing the same judgments $M \triangleright N$, where $M$ and $N$ are terms and $\triangleright$ is a new symbol. This judgment means 'M reduces to N': for $\rightarrow_\beta$, it is a reduction in one
step, for $\rightarrow_\beta$, it is a reduction in several steps, and for $\rightarrow_\beta$ it is an inter-reduction.

The inference system for immediate $\beta$-reduction contains the contraction rule, with no premises, and three unary rules for context passing.

**Definition 3.1** Immediate $\beta$-reduction is a binary relation, written $\rightarrow_\beta$, or simply $\rightarrow$: $M \rightarrow_\beta N$ if and only if there exists a closed derivation, i.e., without assumption, of the judgment $M \triangleright N$ by the inference system:

\[
\begin{align*}
(\lambda) & : M \triangleright M' \quad \lambda x M \triangleright \lambda x M' \\
(1) & : \frac{M \triangleright M'}{M \triangleright M'} \\
(2) & : \frac{N M \triangleright N M'}{N M \triangleright N M'}
\end{align*}
\]

If $M \rightarrow_\beta N$, $M$ reduces immediately to $N$.

**Example** Let $R$ be a redex which contracts to $C$, and $M$ be an arbitrary term. Here is a derivation of $\lambda x. R M \triangleright \lambda x. C M$ (recall that $\lambda x. R M$ is a simplified notation for $\lambda x (R M)$):

\[
\begin{align*}
(\text{red}) & : R \triangleright C \\
(1) & : \frac{R M \triangleright C M}{\lambda x. R M \triangleright \lambda x. C M}
\end{align*}
\]

Recall that a term $M$ is normal (§2.2.4) if it has no redex subterms, i.e., if $O_{\text{red}}(M) = \varnothing$.

It is not a coincidence that the context-passing rules are designated by the names $\lambda$, 1 and 2, which form the alphabet of occurrences for $\lambda$-terms (see §2.3.2). A ternary relation is defined between two terms $M$, $N$ and an occurrence $u$ by $M \overset{u}{\rightarrow} N$ if $u \in O_{\text{red}}(M)$ and $M/u \equiv (\lambda x P) Q$, $C \equiv P[x := Q]$ is the result of contracting $M/u$, and $N \equiv M[u \leftarrow C]$.

**Proposition 3.1** $M \rightarrow_\beta N$ if and only if there exists $u \in O_{\text{red}}(M)$ such that $M \overset{u}{\rightarrow} N$.

**Proof** Reason by induction over the derivation of $M \triangleright N$. Recall that all the rules are of arity 0 or 1.

If $M$ is a redex and $N$ the result of contracting $M$, define $u = \epsilon$.

Otherwise, $M \triangleright N$ is inferred by one of the unary rules. Suppose that it is the first one, $(\lambda)$, with $M \equiv \lambda x M_1$, $N \equiv \lambda x N_1$ and $M_1 \rightarrow_\beta N_1$. By the inductive hypothesis, there exists $u_1 \in O_{\text{red}}(M_1)$ such that $M_1 \overset{u_1}{\rightarrow} N_1$. Deduce that $u = \lambda \cdot u_1 \in O_{\text{red}}(M)$ and that $M \overset{u}{\rightarrow} N$. The second and third cases are treated similarly.
Conversely, if \( M \vdash N \), construct a derivation of \( M \triangleright N \) by starting with the contraction rule of \( M/u \) and successively applying the unary rules by reading \( u \) from right to left. In the preceding example, the rede \( R \) of \( \lambda x. RM \) is at occurrence \( u = \lambda 1 \), and rules (red), (1) and (\( \lambda \)) are applied.

Context passing can also be written concisely, using the single rule,

\[ (\text{cont}) : \frac{P \triangleright Q}{C[P] \triangleright C[Q]}, \]

\( C \) being an arbitrary term context. Proposition 3.1 shows that it is sufficient to apply this rule when \( P \) is a rede.

An attempt will be made to reduce a term by successive immediate \( \beta \)-reductions, hoping that a normal form will be reached. To do this means to compute the reflexive, transitive closure of immediate \( \beta \)-reduction, called \( \beta \)-reduction.

**Definition 3.2** \( \beta \)-reduction is a binary relation, written \( \triangleright \), or simply \( \triangleright \): \( M \triangleright N \) if and only if there exists a closed derivation of judgment \( M \triangleright N \) by the following inference system:

\[ \begin{array}{c}
(\beta) : \frac{M \triangleright M'}{M \triangleright M'} \quad \text{if} \quad M \triangleright \beta M' \\
(R) : \frac{M \triangleright M}{M \triangleright M} \\
(T) : \frac{M \triangleright M' \quad M' \triangleright M''}{M \triangleright M''}
\end{array} \]

**Proposition 3.2** \( M \triangleright M' \) if and only if there exist terms \( M_0, M_1, \ldots, M_n, n > 0 \), such that \( M_0 \equiv M, M_n \equiv M' \), and \( M_i \triangleright \beta M_{i+1} \) or \( M_i \equiv M_{i+1} \), for \( 0 \leq i < n \).

**Proof** By induction over the derivation of \( M \triangleright M' \) proving \( M \triangleright \beta M' \).

- If \( M \triangleright M' \) is derived from \((\beta)\) or from \((R)\), then \( M_0 \equiv M \) and \( M_1 \equiv M' \).
- If \( M \triangleright M'' \) is derived by applying \((T)\), by the inductive hypothesis, there are two sequences,

  \[ M \equiv M_0, \ M_1, \ldots, M_n \equiv M' \]

for \( M \triangleright M' \) and

  \[ M' \equiv M'_0, \ M'_1, \ldots, M'_n \equiv M'' \]

for \( M' \triangleright M'' \). The required sequence is

\[ M \equiv M_0, \ M_1, \ldots, M_n, \ M'_1, \ldots, M'_n \equiv M''. \]

Conversely, one can easily construct a derivation of \( M \triangleright M' \) from a sequence of immediate \( \beta \)-reductions. \( \square \)
The \( \rightarrow \) relation expresses the concept of computation as reduction. Also of interest is the transitive closure of \( \rightarrow^\beta \), written \( \uparrow \) and defined analogously to \( \rightarrow \), but without rule (R). Finally, the equivalence relation induced by \( \rightarrow^\beta \) is written \( =^\beta \), or simply \( = \), or even \( =^*^\beta \): it is called \( \beta \)-conversion. In programming, it corresponds to equivalence between programs. As the reflexive, symmetric and transitive closure of \( \rightarrow^\beta \), it is defined by adding the symmetry rule

\[
\begin{array}{c}
(M \triangleright N) \\
(N \triangleright M)
\end{array}
\]

(S)

to the inference system defining \( \rightarrow \).

3.1.2 Fundamental properties

Note that \( \beta \)-reduction does not always reduce the size of terms, and that it is not always possible to reach a normal form.

**Definition 3.3** A term \( M \) is *normalizable* if there exists a \( \beta \)-reduction \( M \rightarrow N \), where \( N \) is normal.

**Examples** Normal terms are normalizable. The term \( \Omega \equiv \Delta \Delta \), where \( \Delta \equiv (\lambda x . xx) \), is not normalizable, since the only possible immediate \( \beta \)-reduction of \( \Omega \) is \( \Omega \rightarrow^\beta \Omega \rightarrow^\beta \Omega \rightarrow^\beta \ldots \). Its *reduction graph* has one vertex, \( \Omega \), and a loop.

The following result, due to Church, indicates the level of difficulty of the \( \lambda \)-calculus; it will be proven in Chapter 7, where decidability will be defined:

**Theorem 3.3** The normalization problem is undecidable.

The fundamental property of \( \beta \)-reduction is confluence. To define it, the transitive, reflexive closure of a binary relation \( \rightarrow \) is written \( \rightarrow^* \).

**Definition 3.4** Let \( x \in E \). The binary relation \( \rightarrow \) over \( E \) is

- *confluent* in \( x \), if for every \( x_1, x_2, \) if \( x \rightarrow x_1 \) and \( x \rightarrow x_2 \), there exists \( x' \) such that \( x_1 \rightarrow^* x' \) and \( x_2 \rightarrow^* x' \);

- *locally confluent* in \( x \), if for every \( x_1, x_2, \) if \( x \rightarrow x_1 \) and \( x \rightarrow x_2 \), there exists \( x' \) such that \( x_1 \rightarrow^* x' \) and \( x_2 \rightarrow^* x' \);

- *strongly confluent* in \( x \), if for every \( x_1, x_2, \) if \( x \rightarrow x_1 \) and \( x \rightarrow x_2 \), there exists \( x' \) such that \( x_1 \rightarrow x' \) and \( x_2 \rightarrow x' \).

These properties can be expressed by the following diagrams, where a complete line indicates a hypothesis ('if ... ') and a dotted line indicates a conclusion ('there exists ... '):
A binary relation $\rightarrow$ over a set $E$ is confluent (respectively locally confluent, strongly confluent) if it is in every element of $E$.

Confluence is a fundamental property of every model using reduction for computation: it means the independence of the result of a computation from the reduction taken. The conversion relation turns out to mean reducing to the same term: this is called the Church–Rosser property.

**Proposition 3.4** If relation $\rightarrow$ is confluent, then $x \xrightarrow{*} y$ is equivalent to the existence of $z$ such that $x \xrightarrow{*} z$ and $y \xrightarrow{*} z$.

**Proof** By induction on the derivation of $x \xrightarrow{*} y$. If $x \xrightarrow{*} y$ results from

- $x \rightarrow y$, take $z \equiv y$;
- the reflexivity rule (with $x \equiv y$), take $z \equiv x$;
- $y \xrightarrow{*} x$, by the symmetry rule, the inductive hypothesis applied to $y \xrightarrow{*} x$ directly yields $z$;
- $x \xrightarrow{*} t$ and $t \xrightarrow{*} y$ by the transitivity rule, the inductive hypothesis applied to $x \xrightarrow{*} t$ and to $t \xrightarrow{*} y$, respectively, provides $x_1$ and $y_1$ such that $x \xrightarrow{*} x_1$, $t \xrightarrow{*} y_1$, and $y \xrightarrow{*} y_1$. By applying the confluence property to the two reductions emanating from $t$, $z$ is obtained such that $x_1 \xrightarrow{*} z$ and $y_1 \xrightarrow{*} z$.

This proposition states that confluence implies the consistency of a reduction relation, i.e., that it is impossible to prove $M \xrightarrow{*} N$ for all terms. It suffices to take two normal terms $M$ and $N$ which are syntactically different, for example two distinct variables: if $M \xrightarrow{*} N$, then there exists $P$ such that $M \xrightarrow{*} P$ and $N \xrightarrow{*} P$, hence $M \equiv P$ and $N \equiv P$, since $M$ and $N$ are normal, which contradicts $M \not\equiv N$.

**Theorem 3.5 (Church–Rosser)** Immediate $\beta$-reduction $\rightarrow_{\beta}$ is confluent.

This theorem will be proven when the reduction mechanism is more finely presented, using the concept of residue.

**Corollary 3.6** If $M$ is normalizable, there exists a unique normal term, written $\overline{M}$, such that $M \xrightarrow{*} \overline{M}$. 
Proof If \( M \) is normalizable, there exists a normal \( M_1 \) such that \( M \rightarrow^* M_1 \). If \( M \rightarrow^* M_2 \), \( M_2 \) normal, then by confluence there exists \( M_3 \) such that \( M_1 \rightarrow^* M_3 \) and \( M_2 \rightarrow^* M_3 \). If \( M_1 \) and \( M_2 \) are normal, then \( M_1 \equiv M_3 \) and \( M_2 \equiv M_3 \), hence \( M_1 \equiv M_2 \). □

\( \bar{M} \) is called the normal form of \( M \). This uniqueness allows one to look for the normal form in a non-deterministic manner.

One must not conclude that any reduction of a normalizable term leads to a normal term: the successive choice of redexes to be contracted plays a crucial role. For example, \( M \equiv (\lambda x y)\Omega \) is normalizable. Its normal form is obtained by contracting \( M \). If \( \Omega \) were always contracted, the infinite reduction \( M \rightarrow M \rightarrow M \rightarrow \ldots \) would be obtained.

Definition 3.5 A term \( M \) is strongly normalizable if it does not have an infinite reduction \( M \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots \).

There are normalizable terms which are not strongly normalizable. This is the case for \( (\lambda x y)\Omega \). This example also shows that a term can be normalizable even if not all of its subterms are, and that precautions must be taken when reducing.

If the \( \lambda \)-calculus is to be useful, as a method of computing the normal form of a term, a reduction strategy must be fixed: for each term, the redex to be contracted must be chosen. For example, the leftmost strategy consists of contracting the leftmost redex (whose \( \lambda \) is leftmost). By making the search for the normal form deterministic, the choice of strategy determines for each \( M \) a unique reduction which is either finite, with \( M \rightarrow^* M_n \) and \( M_n \) normal, or infinite. A strategy is normalizing if the reduction that it determines for every normalizable term is finite. The following theorem is proven in [8, theorem 13.2.2].

Theorem 3.7 The leftmost reduction strategy is normalizing.

Corollary 3.8 If the leftmost reduction of a term is infinite, then the term is not normalizable.

Corollary 3.9 The normalization problem is semi-decidable.

Proof If \( M \) is normalizable, the leftmost reduction gives the normal form of \( M \) in a finite time. If \( M \) is not normalizable, one can always wait \( \ldots \) □

3.1.3 Programming in \( \lambda \)-calculus

Although it is easy to write interpreters for the \( \lambda \)-calculus that apply a reduction strategy, it is very difficult to program in it, i.e., to write terms defining the usual objects and algorithms of computer science. In general, it is the operational aspect of the objects that yields their \( \lambda \)-calculus representation. For example, integers are ‘function iterators’ and Booleans are ‘branchings’.
Booleans  The combinators T and F representing the Booleans \texttt{true} and \texttt{false} must serve as branchings: \( T \, x \, y \rightarrow^* x \) and \( F \, x \, y \rightarrow^* y \). Thus
\[
\begin{align*}
T & \equiv \lambda xy. \, x, \\
F & \equiv \lambda xy. \, y.
\end{align*}
\]
A conditional expression, analogous to \((b \ ? \ x : y)\) in C, is simply represented by the application of the Boolean condition to its two branches:
\[
\text{if } \equiv \lambda bxy. \, bxy,
\]
so that if \( T \, x \, y \rightarrow^* x \) and if \( F \, x \, y \rightarrow^* y \). The Boolean operations can be defined by the combinators
\[
\begin{align*}
\text{and } & \equiv \lambda xy. \, xyF, \\
\text{or } & \equiv \lambda xy. \, xTy, \\
\text{not } & \equiv \lambda bxy. \, byx.
\end{align*}
\]
So, \( \text{and} \, Ty = TyF = y \) and \( \text{and} \, Fy = FyF = F \) (note, however, that \( \text{and} \, xF = xFF \) does not reduce to \( F \); the two arguments do not play the same role), and \( \text{or} \, Ty = T \) and \( \text{or} \, Fy = y \).

Pairs  Booleans allow one to define pairs as objects responding to the messages \( T \) and \( F \), in the object-oriented programming sense, by giving the appropriate component: \((\text{cons} \, xy)T \rightarrow^* x \) and \((\text{cons} \, xy)F \rightarrow^* y \). Hence the projections:
\[
\pi^1 \equiv \lambda c. \, cT, \quad \pi^2 \equiv \lambda c. \, cF,
\]
and the pairing constructor:
\[
\text{cons } \equiv \lambda xym. \, mxy.
\]
A purely functional language, i.e., based on the \( \lambda \)-calculus, is therefore adapted to the `message-passing' style of programming. The necessarily functional nature of objects allows them to respond to messages: it suffices to pass those messages as arguments.

Union  The structure dual to pairs is union, constructed using injections: \( i^1 M \) receiving messages \( f \) and \( g \) must apply \( f \) to \( M \) while \( i^2 M \) applies \( g \) to \( M \): \((i^1 x)fg \rightarrow^* fx \) and \((i^2 x)fg \rightarrow^* gx \). This gives the two constructors
\[
\begin{align*}
i^1 & \equiv \lambda xfg. \, fx, \\
i^2 & \equiv \lambda xfg. \, gx.
\end{align*}
\]

Integers  A natural method, due to Church, is to represent the integers as iterators of functions: the integer \( n \) is coded by the combinator (Church numeral) \( n \equiv \lambda fx. \, f^n x \). In particular,
\[
\begin{align*}
0 & \equiv \lambda fx. \, x \equiv F, \\
1 & \equiv \lambda fx. \, fx.
\end{align*}
\]
The two operations essential for them to be considered as integers are the successor function and the zero equality test:

\[
\text{succ } \equiv \lambda nf . f((nf)x), \\
\text{zero } \equiv \lambda nxy . n(Tx)y,
\]

where \( K \equiv T \equiv \lambda xy . x \). So, \( \text{succ } n \rightarrow p \) if \( p = n + 1 \), and

\[
\text{zero } nxy = \begin{cases} 
  x & \text{if } n \neq 0 \\
  y & \text{if } n = 0.
\end{cases}
\]

The other arithmetic operations are easy to program by successive iterations of successor:

\[
\begin{align*}
\text{add } & \equiv \lambda mngz . (mg)((ng)z), \\
\text{mult } & \equiv \lambda mng . m(ng), \\
\text{exp } & \equiv \lambda mn . nm.
\end{align*}
\]

**Fixpoint combinators** A closed term \( Y \) is called a fixpoint combinator if for every term \( M \), \( YM =_\beta M(YM) \). Here are two, due to Church (\( Y_C \)) and to Turing (\( Y_T \)):

\[
Y_C \equiv \lambda f . VV \quad \text{where} \quad V \equiv \lambda x . f(xx), \\
Y_T \equiv ZZ \quad \text{where} \quad Z \equiv \lambda zf . f(zzf).
\]

For each term \( M \),

\[
\begin{align*}
Y_T M & \equiv (\lambda zf . f(zzf))ZM \\
& \rightarrow f(zzf)[f := M, z := Z] \\
& \equiv M(ZZM) \\
& \equiv M(Y_T M).
\end{align*}
\]

For \( Y_C \), there is only a \( \beta \)-conversion, not a beta-reduction.

These combinators allow one to solve *equations*. Let \( M \) be an arbitrary term and \( n \geq 0 \). To solve for \( x \) in equation

\[
x y_1 \ldots y_n = M
\]

is to find a term \( F \) such that \( F y_1 \ldots y_n =_\beta M[x := F], \) where \( x, y_1, \ldots, y_n \) are all variables of the \( \lambda \)-calculus. There is always a solution, which is

\[
F \equiv Y(\lambda xy_1 \ldots y_n . M),
\]

where \( Y \) is a fixpoint combinator.

In programming, one can write recursive programs with the help of fixpoint combinators. For example, suppose the factorial function must be programmed. It is the solution of the equation

\[
fact(x) = \text{if } (x = 0) \text{ then } 1 \text{ else } x \cdot \text{fact(ped}(x)),
\]
so \textit{fact} is a fixpoint of the ‘functional’

\[
\lambda f x.\ \text{if } (x = 0) \text{ then } 1 \text{ else } x \cdot f(\text{pred}(x)).
\]

One can therefore define

\[
\text{fact} \equiv Y(\lambda f x.\ \text{if } (x = 0) \text{ then } 1 \text{ else } x \cdot f(\text{pred}(x))),
\]

where \( Y \) is a fixpoint combinator. To make this term explicit, define

\[
K \equiv \lambda x y.\ x, \\
\text{pred} \equiv \lambda x.\ (n(\lambda u.\ u(\text{succ}(p\ T))(p\ T))(\lambda u.\ u\ 0\ 0))F.
\]

The combinator \text{pred} represents the predecessor, where if \( n \) is a Church numeral,

\[
\text{pred } n = \begin{cases} 
  m & \text{if } n = m + 1 \\
  0 & \text{if } n = 0.
\end{cases}
\]

See exercise 4.

A \( \lambda \)-term for the factorial is computed as follows:

\[
\text{fact} \equiv Y(\lambda f x.\ \text{zero } x 1(\text{mult } x(f(\text{pred } x)))).
\]

The use of abbreviations (giving names to certain combinators) and of syntactic sugar (\textit{let} \ldots \textit{in} \ldots \textit{end}) already greatly simplifies terms, but it is mostly the introduction of a sufficiently expressive system of types which allows for real programming. The pure \( \lambda \)-calculus should be considered to be a low-level language, a kind of ‘machine language’. This assumption allows one to study language mechanisms and semantics uniformly. This path was first followed in the 1960s by Landin [93].

3.1.4 Typed \( \lambda \)-calculus

In mathematics, a mapping’s definition includes the domain and range (co-domain). This aspect was not treated by \( \lambda \)-calculus terms. So the set-theoretic meaning of terms such as \( \lambda x.\ xx \) is unclear.

\textit{Typed terms}

The first solution consists of introducing simple types.

\textit{Definition 3.6} Given a set \( A \) of atomic type symbols, the closed terms over a signature \( A \cup \{ \rightarrow \} \), \( \text{ar}(\rightarrow) = 2 \) and \( \text{ar}(a) = 0, \ a \in A \), are called simple types.

To do arithmetic, one could define \( A = \{ \text{nat} \} \). Objects of type \text{nat} are then interpreted as integers, objects of type \text{nat} \rightarrow \text{nat} as functions from \( \mathbb{N} \) into \( \mathbb{N} \),
those of type \((\text{nat} \to \text{nat}) \to (\text{nat} \to \text{nat})\) transform functions into functions, and so on. To program, \(\text{bool}, \text{string}\) and others would also have to be included among the atomic types.

Write types with \(\sigma, \tau\) and abbreviate \(\tau_1 \to (\tau_2 \to (\ldots \to (\tau_{n-1} \to \tau_n) \ldots))\) by \(\tau_1 \to \tau_2 \to \ldots \to \tau_{n-1} \to \tau_n\), or even \(\tau_1, \tau_2, \ldots, \tau_{n-1} \to \tau_n\) by noticing that the type can be interpreted by 'decuringly' as that of functions of \(n-1\) variables, respectively, in \(\tau_1, \ldots, \tau_{n-1}\), and values in \(\tau_n\).

Definition 3.7 Let \(X^\sigma\) be a countable infinite set of type variables, which are written \(x^\sigma, y^\sigma, \ldots\) for each type \(\sigma\). Typed terms are defined by:

1. Each variable \(x^\sigma\) is a typed term of type \(\sigma\).
2. If \(M^{\sigma \to \tau}\) and \(N^\sigma\) are typed terms of respective types \(\sigma \to \tau\) and \(\sigma\), then \((M^{\sigma \to \tau} N^\sigma)^\tau\) is a typed term of type \(\tau\).
3. If \(x^\sigma\) is a variable of type \(\sigma\), and if \(M^\tau\) is a typed term of type \(\tau\), then \((\lambda x^\sigma. M^\tau)^{\sigma \to \tau}\) is a typed term of type \(\sigma \to \tau\).

The marking of the type as superscript on a term and its subterms is indispensable (even though it will often be omitted for convenience). Hence, as for finite terms and pure \(\lambda\)-terms, the very writing of a term is a proof that it is formed correctly. By omitting the type annotations, a pure term is obtained: the formation rules for typed terms thus appear to be a restriction on the formation rules for pure terms.

For each type \(\tau\) there is an identity \((\lambda x^\tau. x^\tau)^{\tau \to \tau}\), but there is no typed term corresponding to \(\lambda x. xx\).

As for the pure \(\lambda\)-calculus, this set of terms can be extended by typed constants, for example \(0, 1, 2, \ldots\) of type \(\text{nat}\), \(\text{succ}\) of type \(\text{nat} \to \text{nat}\), \(+\) of type \(\text{nat} \to \text{nat} \to \text{nat}\); a higher-order signature \(\Sigma\) is produced, formed from the set of its constants, along with an arity function from \(\Sigma\) into the set of types, and the following clause must be added to the preceding definition:

4. If \(f \in \Sigma\) and \(\operatorname{ar}(f) = \tau\), then \(f\) is a typed term of type \(\tau\).

For it to be useful, this extension supposes that specific computing rules (the so-called \(\delta\) rules) be given for the constants, for example \(1 + 1 \to^\delta 2\). From a practical point of view, in a functional programming language these rules consist of calling external procedures to handle arithmetic efficiently. Furthermore, other computing mechanisms, which the type system cannot allow, such as recursion, can be added in the same manner. However, the addition of constants is theoretically unnecessary to increase the computational power of pure \(\lambda\)-calculus. These \(\delta\) rules complicate the study of reduction considerably: the confluence of \(\beta\)-reduction can be destroyed by adding such rules. Therefore, no \(\delta\) rule will be considered here, and, in general, only typed terms without constants will be considered.

Simple types are stratified according to their order. A function \(\text{ord}\) is defined, associating an integer to each atomic type, for example \(\text{ord}(\text{nat}) = 1\). This function
is extended by:
\[ \text{ord}(\tau_1 \rightarrow \tau_2) = \max(\text{ord}(\tau_1) + 1, \text{ord}(\tau_2)). \]

Hence, \text{nat} is of order 1, \text{nat} \rightarrow \text{nat} and \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}) are of order 2, and \((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \) and \((\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}) \) are of order 3. Note the difference between the \textit{false} higher-order type \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}), which is actually the type of \textit{curried functions of two variables}, and \((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \), which is the type of \textit{functionals}.

The order of a variable is that of its type. A typed term is of order \(\leq n\) if all of its variables (free or bound) are of order \(\leq n\): \(\text{nat} \) and \(\lambda x. x\) are of order 1, and \(\lambda f. f x\) is of order 2. When a term contains constants, it is of order \(\leq n\) if the types of its constants are also of order \(\leq n + 1\): \(0\) is of order 0, 1 + 1 and \(x + 1\) are of order 1. Hence, a \textit{first-order} \(\Sigma\)-term is of order \(\leq 1\): there is an atomic type of order 1 (or several, in the case of many-sorted signatures), its variables are of order 1, and the types of symbols in \(\Sigma\) are of order 1 for the constants, and 2 for the function symbols. A \(\Sigma\)-term of order 0 is a constant symbol.

This notion of order allows one to classify computer languages. A language allowing definitions of functions over numeric values is of order 1. Pascal is of order 2, as function definitions can have functional parameters of order 1. Functional languages such as ML are of order \(\omega\), since they allow the use of variables of arbitrarily high order.

The main definitions of the untyped \(\lambda\)-calculus translate directly to the typed one (graft, substitution, redex, \(\beta\)-reduction, residues, \ldots). For substitution, a variable must be substituted with a term of the same type. The Church–Rosser theorem is therefore valid, as are its corollaries. There are new phenomena: the weak and strong normalization properties.

\textit{Theorem 3.10} All typed terms are normalizable.

\textit{Theorem 3.11} All typed terms are strongly normalizable.

Hence the normal form of a typed term is obtained by any reduction. Strong normalization is an important result which is difficult to prove. The \(\beta\)-convertibility of two terms is therefore decidable: it suffices to compute the normal forms of two terms and to compare them syntactically to decide if they are \(\beta\)-convertible. This follows from the strong normalization theorem and the proof of the weak normalization theorem, which can be adapted into an effective reduction strategy. The proof of the weak normalization theorem will be given in the last section of the chapter, which deals with the problem of termination.

\textit{Typable terms}
Typed terms were introduced so that terms would better suit the set definition of a function. However, this approach is not really natural. There are an infinite
number of terms $I'$ for the identity function, but they all do the same thing, regardless of the argument. Furthermore, even in mathematics, it is common to define a function—using an algorithm, for example—before determining the domain over which it is defined, e.g., the set of values for which the algorithm terminates. This suggests that it is the pure terms which are essential. Typing them just adds more information. To assign a type to a pure term is already giving it a kind of semantics. It is an abstract interpretation, which does not compute the value of a term, but, rather, the set of possible values: a pure term of type $\text{nat} \to \text{nat}$ can be interpreted as a function over the integers with integer values.

The problem is therefore to type pure terms. A first difficulty comes from the variables. Variables must be assigned a type, as in programming, but if one wants polymorphic types, as for the identity, what type should they be given? The solution is to introduce type variables, written $\xi, \eta, \ldots$. Simple types were terms closed over a signature $A \cup \{\to\}$. They are now terms, with variables, over this signature.

Since every term is obtained from its variables, it suffices to type its variables, then to infer its type. For example, if $x$ is declared to be of type $\xi$, it can be inferred that $I \equiv \lambda x x$ is of type $\xi \to \xi$. One can also declare $x$ to be of type $\text{nat} \to \text{nat}$ and then deduce that $I$ is of type $(\text{nat} \to \text{nat}) \to (\text{nat} \to \text{nat})$. An ‘assumption’ is made about the types of the variables. A finite set of variable type declarations forms a typing context, of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$, where $x_1, \ldots, x_n$ are variables of the $\lambda$-calculus and $\tau_1, \ldots, \tau_n$ are types.

Here is an inference system. Its judgments are of the form $\Gamma \vdash M : \tau$, stating that in the typing context $\Gamma$, the term $M$ is assigned type $\tau$. When $\Gamma$ is empty, write $\vdash M : \tau$ instead of $\emptyset \vdash M : \tau$:

\[
\begin{array}{c}
x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i \\
C \vdash M : \sigma \to \tau \quad C \vdash N : \sigma \\
C \vdash MN : \tau \\
\{x : \sigma\} \cup C \vdash M : \tau \\
C \vdash \lambda x M : \sigma \to \tau
\end{array}
\]

A closed term $M$ is typable if there exists a type $\tau$ such that the judgment $\vdash M : \tau$ can be derived.

**Example** Recall that $K \equiv \lambda xy. x$. To prove that $\vdash K : \xi \to \eta \to \xi$, start with $x : \xi$ and $y : \eta \vdash x : \xi$. Abstracting over $y$ yields $x : \xi \vdash \lambda y x : \eta \to \xi$; abstracting over $x$ yields $\vdash \lambda x(\lambda y x) : \xi \to (\eta \to \xi)$. \qed

Note that in the third rule (for $\lambda x M$), the assumption $x : \sigma$ appearing in the context of the premise does not appear in the context of the conclusion (unless it is also in $\Gamma$): the assumption $x : \sigma$ is discharged by the inference rule. On the
other hand, in the second rule (for $MN$), it seems superfluous to mention context $C$ since it is not modified. One can then write these rules simply by indicating the discharged hypotheses, usually written between square brackets ($[\ldots]$) or slashed ($\div$).

$$\frac{M : \sigma \rightarrow \tau \quad N : \sigma}{MN : \tau}, \quad \frac{[x : \sigma]}{M : \tau} \quad \frac{\lambda x \, M : \sigma \rightarrow \tau}{\lambda x \, M : \sigma \rightarrow \tau}.$$  

This formulation will be used when discussing natural deduction, where the problem will be different: the type $\tau$ will be known, and terms of type $\tau$ will have to be found.

A type $\tau$ such that $C \vdash M : \tau$ is not necessarily unique. Consider the case of I. There is only one rule for which the conclusion is an abstraction; one can only have inferred $\lambda x \, x : \sigma \rightarrow \tau$ from $x : \sigma \vdash x : \tau$ using the first rule, and with $\sigma = \tau$. Hence, all judgments derivable for I are the $\vdash I : \tau \rightarrow \tau$, with $\tau$ arbitrary. They are therefore all instances of $\xi \rightarrow \xi$, where $\xi$ is an arbitrary variable. The type $\xi \rightarrow \xi$ is called a principal type of I. This principal type is not even unique: $\eta \rightarrow \eta$ is another one, obtained by renaming. This question will be cleared up when the subsumption preorder is defined over types.

Typed programming languages Each type language has a ‘type system’ which contains atomic types, such as `nat` or `bool`, type formation rules and expression typing rules. The formation rules introduce type constructors. In functional languages, the $\rightarrow$ constructor is essential, but the majority of languages also have constructors for the product or the union of types. Typing rules, including typed constant declarations, e.g., `true : bool` and `suc : nat \rightarrow nat`, allowing the inference of judgments $C \vdash M : \tau$, can be used:

- if $C$, $M$ and $\tau$ are known, to check that $M$ is correctly typed: this is type-checking, used for typed languages such as Pascal where all the variables, functions, etc., must be declared along with their type;
- if only $C$ and $M$ are known, to infer the (principal) type $\tau$ of $M$: this is type inference, used by ML and most other contemporary functional languages.

### 3.2 Rewriting systems

The second general construction forming the basis for functional languages has a geometric flavor. It is the subsumption preordering over first-order terms. When combined with the $\lambda$-calculus, this relation creates an extension of $\beta$-reduction, call by pattern matching, found in most functional languages. A more general form, call by unification, is the basis for logic programming. Pattern matching allows one to write computation rules with more flexibility than does the bare $\lambda$-calculus: these rules are rewrite rules, whose study is now an integral part of computer science.
3.2.1 Pattern matching

Recall the notation of Chapter 1: $\Sigma$ is a signature, $X$, $T_\Sigma[X]$ and $S$ are, respectively, the set of variables, terms and substitutions. Instead of writing $T_\Sigma[X]$, $T[X]$ will be used. The definition of substitutions allows one to distinguish variables from constants, by internalizing the concept of metasymbol. It allows one to define a ‘topology’ over $T[X]$, and, more precisely, to define a preorder.

The definition of a term $M \in T[X]$ determines the set $S M = \{ \theta M ; \theta \in S \}$. Its elements are the instances of $M$, and $M$ is its pattern. If $x$ is a variable, then $S x = T[X]$, and if $M$ is a closed term, then $S M = \{ M \}$. Hence, the ‘size’ of $S M$ measures the genericity of $M$. As $S M$ becomes smaller, term $M$ becomes more defined, i.e., gives information. This level of information defines an order relation. These ideas are very general and can of course be applied to many objects other than terms, such as words, graphs and ‘forms’.

Definition 3.8 Let $M, M' \in T[X]$. The term $M'$ is an instance of $M$, or $M$ is a generalization of $M'$, written $M \preceq M'$, if there exists a substitution $\theta \in S$ such that $\theta M \equiv M'$. The restriction of $\theta$ to $\text{var}(M)$, which is uniquely determined by $M$ and $M'$, is called the matcher of $M$ towards $M'$. The relation $\preceq$ is the subsumption preorder over terms.

It is clear that $M \preceq M'$ if and only if $S M \supseteq S M'$. Hence, $\preceq$ is a preorder relation, i.e. it is reflexive and transitive. But it is not antisymmetric, so it is not an order relation: for example, if $x$ and $y$ are distinct variables, $x \leq y$ and $y \leq x$, and $x \neq y$. This defect comes from the nature of variables, which is to carry minimal information. Consider the equivalence relation generated by the preorder.

Definition 3.9 Let $M, M' \in T[X]$. The term $M$ is a variant of $M'$, written $M \sim M'$, if $M \preceq M'$ and $M' \preceq M$. Define $\bar{T} = T/\sim$ and use $\preceq$ for the order relation induced by $\preceq$ over $\bar{T}[X]$.

Proposition 3.12 $M \sim M'$ if and only if there exists a renaming, i.e., a bijective substitution $\rho$, taking variables to variables, such that $M' = \rho M$.

Proof If $M' = \rho M$, where $\rho$ is a renaming, then $M = \rho^{-1} M'$, hence $M \preceq M'$ and $M' \preceq M$, so $M \sim M'$.

Conversely, if $M \sim M'$, let $\theta$ and $\theta'$ be matchers from $M$ towards $M'$ and from $M'$ towards $M$ respectively. Then $\theta \theta' M = M$ and $\theta' \theta M' = M'$. In particular, if $x \in \text{var}(M)$, $\theta' \theta(x) = x$, which shows that the restriction of $\theta$ to $\text{var}(M)$ is injective. Since $\theta(x)$ is a variable as well as a subterm of $M'$, $\theta(x) \in \text{var}(M')$. Similarly, $\theta'(x) \in \text{var}(M)$ if $x \in \text{var}(M')$. Hence, $\theta_{|\text{var}(M)} : \text{var}(M) \rightarrow \text{var}(M')$ and $\theta'_{|\text{var}(M')} : \text{var}(M') \rightarrow \text{var}(M)$ are injections, which proves that $\text{var}(M)$ and $\text{var}(M')$ are equipotent. Define the substitution $\rho$ by

$$
\rho(x) = \begin{cases} 
\theta(x), & \text{if } x \in \text{var}(M) \\
\phi(x), & \text{if } x \in \text{var}(M') \setminus \text{var}(M) \\
x & \text{otherwise}
\end{cases}
$$
where $\phi$ is a bijection from $\text{var}(M') \setminus \text{var}(M)$ into $\text{var}(M) \setminus \text{var}(M')$. \hfill \Box

The equivalence $\sim$ equates terms which are equal up to the renaming of variables. In particular, all variables are equivalent and their class constitutes the smallest element of $\bar{T}[X]$, written $\bot$. The maximal elements of $\bar{T}[X]$ are the closed terms. The class of term $M$ is written $[M]$.

**Example** If $\Sigma = \{0, s\}$, then $\bar{T}[X]$ includes the terms $[s^00]$ and $[s^px]$, with relations $[s^px] \leq [s^{p+1}x]$ and $[s^{p+1}x] \leq [s^00]$ for all $p \in \mathbb{N}$.

The ordered set $(\bar{T}, \leq)$ has remarkable properties. In what follows, the strict order $>$ is used: $t > t'$ if $t \geq t'$ and $t \neq t'$, for $t, t' \in \bar{T}[X]$, and in $T[X]$, $M > M'$ if $M \geq M'$ and $M \not\leq M'$.

**Definition 3.10** An ordered set $(E, \leq)$ is a well-founded inf-semilattice if

1. $<$ is a well-founded strict order over $E$, i.e., there does not exist any sequence $(e_i)_{i \geq 0}$ such that $e_i > e_{i+1}$ for all $i$;
2. every pair $e, e'$ of elements of $E$ has a greatest lower bound, written $e \wedge e'$.

**Proposition 3.13** The order $\leq$ over $\bar{T}[X]$ is well-founded.

**Proof** Define the size $|M|$ of a term $M$ by

1. $|x| = 0$, if $x \in X$;
2. $|f M_1 \ldots M_n| = 1 + \sum_{i=1}^{n} |M_i|$, if $\text{ar}(f) = n \geq 0$;

and let $v(M) = \text{card} \text{var}(M)$. Note that $|\;|$, and $v$ are defined over $\bar{T}[X]$, i.e., are invariant under renaming of variables.

If $M' > M$, then $M' = \sigma M$, where $\sigma$ is not a permutation, and $|M'| \geq |M|$.

Let $(M_i)_{i \geq 0}$ be a strictly decreasing sequence of terms. Then the sequence $\langle |M_i| \rangle_{i \geq 0}$ is decreasing in $\mathbb{N}$. Since $\mathbb{N}$ is well-founded, this sequence becomes stationary at some point: there exists $p \geq 0$ such that $|M_i| = |M_p|$ for all $i \geq p$.

If $|\sigma M| = |M|$, then $\sigma$ does not introduce any new symbol. By assumption $\sigma$ is not a renaming, i.e., is not injective, hence $v(\sigma M) < v(M)$.

Therefore, $v(M_i) < v(M_{i+1})$ for every $i \geq p$. Yet, since the terms $M_i$ all have the same height, the sequence $(v(M_i))_{i \geq p}$ is bounded by the number of occurrences of variables in $M_p$.

The existence of a strictly decreasing sequence of terms is therefore contradictory, which proves that $\bar{T}[X]$ is well-founded. \hfill \Box

This property of the order over $\bar{T}[X]$ can be made stronger. It is not just that a class of terms cannot have a strictly decreasing infinite sequence of lower bounds, but that the number of lower bounds of each class is finite. This property is proven by examining the covering relation associated with the order. In a preorder set $(E, \leq)$, $e$ covers $e'$ if $e > e'$ and there does not exist an element $e''$ such that $e > e'' > e'$. By interpreting the subsumption preorder in terms of information,
the covering relation determines the ‘quanta’ of information. In general, it is not true that, given an order relation λ, the order generated by the covering relation of λ is still λ. This is what actually happens with terms. By using this relation, from exercises 9 and 10 (p. 103), one can deduce that the set of lower bounds of a term is finite.

Generalization The problem of generalization is, given a set A of terms, to find the largest term \( M_0 \), with \( M \supseteq M_0 \) for all \( M \in A \). This situation is typical of artificial intelligence, where, starting from a set of objects such as forms or frames, an attempt is made to find an object of which these objects are all particular cases. The theorem below means that \( \bar{T}[X] \) is a good domain for artificial intelligence.

According to a general result on well-founded inf-semilattices, every non-empty subset admits a least upper bound (the proof is left as an exercise). There is no analogous result for upper bounds. However, each set with an upper bound has a least upper bound, since the non-empty set of upper bounds has a greatest lower bound, which is the least upper bound.

To solve the generalization problem, construct the greatest lower bound of two terms, called the greatest generalization of those terms. This lower bound is not unique in \( T[X] \), but it is in \( \bar{T}[X] \). It is, however, easier to work in \( T[X] \).

For \( M, N \in T[X] \), one constructs a term \( M \land N \) such that \([M \land N]\) is the greatest lower bound of \([M]\) and \([N]\).

If \( M \) and \( N \) have the same head symbol, the inf is defined recursively:

\[
f(M_1 \ldots M_p \land f N_1 \ldots N_p) = f(M_1 \land N_1) \ldots (M_p \land N_p).
\]

If they do not, \( M \land N \) is an arbitrary variable whose class is \( \perp \) in \( \bar{T}[X] \). To apply the recursive case, pairs of terms \((M_i, N_i)\) must be distinguished by different variables.

**Example** Let \( M = f(g(a, b), a, b) \) and \( N = f(g(b, c), b, h(c)) \). Then \( M \land N = f(g(x, y), x, z) \), variable \( x \) being the inf of pair \((a, b)\), which has two occurrences in \((M, N)\).

The easiest way to distinguish pairs of terms is to use an arbitrary bijection \( \Phi \) from \( T^2 \) to \( X \) (the sets \( X \) and \( \Sigma \) are supposed to be countable). Write the converse bijection \( \pi : X \rightarrow T^2 \). Its component functions are \( \pi^1 \) and \( \pi^2 \) from \( X \) to \( T[X] \). Let

\[
M \land N = \Phi(M, N)
\]

if \( M \) and \( N \) do not have the same head symbol.

It is easy to prove that \( M \land N \) is a lower bound for \( M \) and \( N \) (exercise 11). It is more difficult to prove that it is the greatest lower bound, since the order is not compatible with the term structure.

**Lemma 3.14** If \( M'_i = \sigma'M_i \) and \( M''_i = \sigma''M_i \), \( 1 \leq i \leq n \), then there exists a substitution \( \sigma \) such that \( M'_i \land M''_i = \sigma M_i \) for all \( i \).
Proof. By induction over \( l = \sum_{i=1}^{n} |M_i| \).

If \( l = 0 \), then each \( M_i \) is a variable \( x_i \), and \( \sigma(x_i) = M'_i \land M''_i \) defines the wanted substitution.

If \( l > 0 \), then there exist indices \( k_1, \ldots, k_p \) such that \( |M_{k_i}| > 0 \). Define \( M_{k_i} = f_i(M_{k_i}^{i,j}) \). Then \( \sum_i \sum_j |M_{k_i}^{i,j}| = \sum_i (|M_{k_i}| - 1) < \sum_i |M_{k_i}| \leq l \), and \( M'_{k_i} = \sigma' M_{k_i} = f_i(\sigma' M_{k_i}^{i,j}), M''_{k_i} = \sigma' M_{k_i} = f_i(\sigma'' M_{k_i}^{i,j}) \).

By the inductive hypothesis, there exists \( \sigma \) such that \( M'_{k_i}^{i,j} \land M''_{k_i}^{i,j} = \sigma M_{k_i}^{i,j} \). Deduce that \( M'_{k_i} \land M''_{k_i} = \sigma M_{k_i} \). All that is now needed is to extend \( \sigma \) to the \( M_i \) which are variables, as in the case \( l = 0 \): a \( \sigma \) is thus obtained such that \( M'_i \land M''_i = \sigma M_i \) for all \( i \).

It suffices to consider the lemma for \( n = 1 \) to determine that \( M' \land M'' \) is a greatest lower bound for \( M' \) and \( M'' \), i.e., that if \( M \leq M' \) and \( M \leq M'' \), then \( M \leq M' \land M'' \). However, if an attempt had been made to prove the property by induction over terms directly, the inductive hypothesis would have been too weak and would not have sufficed to prove the inductive step. It was necessary to take \( M'_i = \sigma' M_i \) instead of \( M'_i \geq M_i \), for all \( i \). This is a typical example of reinforcing or generalizing a property before proving it by induction: one does not always generalize for the fun of it!

Pattern matching. The problem is to determine, for \( M, M' \in T[X] \), if \( M \leq M' \). If so, the matcher which transforms \( M \) to \( M' \) must be computed. To write a pattern-matching program, it is useful to consider a substitution as an environment, i.e., as a sequence of variable-term bindings: in this manner, for a fixed \( M \), all instances \( \theta M \) share the same structure, that of \( M \).

A matcher from \( M \) to \( N \) in the environment \( \theta \) is needed. Initially, \( \theta \) is the identity. There are two failure cases (halt):

- if \( M \) and \( N \) have distinct head symbols;
- if \( M \) is a variable bound by \( \theta \) to a distinct term of \( N \);

one success case (halt):

- if \( M \) is a variable not bound by \( \theta \) or already bound to \( N \), the matcher is \( \theta \), extended by \((M, N)\);

one iteration case:

- if \( M \) and \( N \) have the same head symbol, iterate over the children of \( M \) and \( N \).

The last case means that if \( M = fM_1 \ldots M_p \) and \( N = fN_1 \ldots N_p \), the matching continues in parallel with \( M_1 \) to \( N_1 \), \ldots, \( M_p \) to \( N_p \). The matching will not be done in parallel, but, rather, sequentially for each pair \((M_i, N_i)\), and by propagating the matcher obtained in the environment. This procedure is standard in functional programming.
The 'cone' properties of the instances of a term are illustrated by the four sketches of Figure 3.1: matchers $\theta$ of $M$ to $M'$, greatest generalization of $M$ and $N$, least upper bound of two terms, when it exists, and disjoint cones of terms otherwise.

3.2.2 Rewrite rules

The subsumption preorder allows one to define computing rules over first-order terms, analogous to $\beta$-reduction, but more flexible.

Definition 3.11 A rewriting system consists of a signature $\Sigma$ and a set $R$ of pairs of terms $(P, Q) \in T[\Sigma][X] \times T[\Sigma][X]$ such that \( \text{var}(Q) \subseteq \text{var}(P) \) and $P \notin X$.

Each pair $(P, Q)$, called a rule, also written $P \rightarrow Q$, defines a generic computing rule: $P$ is a generic redex and $Q$ is the result of contracting $P$. An instance $\sigma P$ is called a redex; the result of contracting $\sigma P$ is $\sigma Q$.

The definition of a reduction relation from a rewriting system is analogous to $\beta$-reduction. A construction can be made using inference rules or by directly using grafts.

Definition 3.12 A term $M$ rewrites itself into a term $N$ by applying the rule $(P, Q) \in R$ to the occurrence $u \in O(M)$ if $M/u \geq P$ and $N = M[u \leftarrow \sigma Q]$, where $\sigma$ is the matcher of $P$ to $M/u$. This relation is written $M \xrightarrow{P \rightarrow Q, u} N$ or simply $M \xrightarrow{u} N$.

The system $(\Sigma, R)$ determines an inference system over judgments $M \triangleright M'$, $M$ and $M'$ being terms. For each $(P, Q) \in R$, there is a family of redex contraction rules. There are also context-passing rules.

\[
\text{(red): } \frac{\sigma P \triangleright \sigma Q}{(P, Q) \in R, \quad \sigma \in S} \\
\text{(f, i): } \frac{M_i \triangleright M'_i \quad fM_1 \cdots M_i \cdots M_n \triangleright fM'_1 \cdots M'_i \cdots M_n}{f \in \Sigma, \quad 1 \leq i \leq \text{ar}(f)}
\]

This inference system defines a binary relation over $T[X]$, written $\rightarrow_R: M \rightarrow_R M'$ if and only if there exists a closed derivation of $M \triangleright M'$. As for the $\lambda$-calculus,
one proves that \( M \rightarrow_{\mathcal{R}} M' \) is equivalent to the existence of \( u \in \mathcal{O}(M) \) and of \((P, Q) \in \mathcal{R} \) such that \( M \xrightarrow{P \rightarrow Q, u} M' \). Passing the context is also written using the single rule
\[
\text{(cont): } \quad \frac{P \triangleright Q}{C[P] \triangleright C[Q]},
\]
\( C \) being a term context, which states that reduction is a local process, independent of the context, as in context-free grammars.

### 3.2.3 Properties of rewriting systems

Unlike the \( \lambda \)-calculus, which uses only one rule, \( \beta \)-reduction rewriting systems use a number of forms. As there are no general theorems, each case must be examined individually.

A rewriting system is not necessarily confluent. For example, the rules \( x \text{ or } y \rightarrow x \) and \( x \text{ or } y \rightarrow y \) which express non-determinism are not confluent. A technique allowing, in certain cases, the proof of confluence will be presented later on. In general, the confluence problem is undecidable.

A system \( \mathcal{R} \) is noetherian if all its terms are strongly normalizable by \( \rightarrow_{\mathcal{R}} \), i.e. if every reduction terminates. A system containing a permutative rule of the form \( x + y \rightarrow y + x \) cannot be noetherian. But there are more subtle cases (see exercise 12). The proof of noetherianity of a system is usually difficult, and linked to the concept of well-order.

**Theorem 3.15** The noetherianity of systems formed of at least two rules is undecidable.

### 3.2.4 Examples

Rewriting systems are used in many areas.

An example is for non-deterministic functional programs, where function calls are made by *pattern matching*. Here is the definition of addition
\[
\begin{align*}
x + 0 & \rightarrow x, \\
x + s(y) & \rightarrow s(x + y),
\end{align*}
\]
and of Ackermann's function
\[
\begin{align*}
A(x + 1, 0) & \rightarrow A(x, 1), \\
A(0, x) & \rightarrow x + 1, \\
A(x + 1, y + 1) & \rightarrow A(x, A(x + 1, y)).
\end{align*}
\]
These systems are confluent and noetherian. The proof of noetherianity, easy for the first, is a termination proof for the program.
The calculus also uses rules implemented in computer algebra systems. Here is a program which differentiates with respect to $\xi$:

\[
\begin{align*}
\partial \xi & \rightarrow 1, \\
\partial a & \rightarrow 0, \\
\partial(u + v) & \rightarrow \partial u + \partial v, \\
\partial(uv) & \rightarrow (\partial u)v + u(\partial v), \\
\partial(-u) & \rightarrow -(\partial u), \\
\partial(u - v) & \rightarrow \partial u - \partial v, \\
\partial(u/v) & \rightarrow ((\partial u)v - u(\partial v))/v^2, \\
\partial(\ln u) & \rightarrow (\partial u)/u, \\
\partial(u^n) & \rightarrow nu^{n-1}\partial u + u^n(\ln u)\partial v,
\end{align*}
\]

where $\xi$ is a constant, the second rule being defined for all constants $a \neq \xi$.

### 3.3 Functional programming

The $\lambda$-calculus is the paradigm of functional programming. High-level functional languages exist, often with a syntax inspired by the $\lambda$-calculus, along similar evaluation mechanisms. The best known are Haskell, Hope, Lisp (particularly Scheme), Miranda and ML.

#### 3.3.1 Evaluation rules in $\lambda$-calculus

**Weak $\beta$-reduction**

No language, functional or not, fully implements $\beta$-reduction, for procedure bodies are never evaluated, except in a partial manner, for optimization. In the case of the $\lambda$-calculus, restrictions are made on $\beta$-reduction; the resulting weak $\beta$-reduction has only the following context rules:

\[
\begin{align*}
(1) : & \quad \frac{M \triangleright M'}{MN \triangleright M'N} \\
(2) : & \quad \frac{N \triangleright N'}{MN \triangleright MN'}
\end{align*}
\]

For this reduction, an abstraction $\lambda x M$ is irreducible, even if $M$ contains a redex. Weakly normal terms are therefore the variables, abstractions and applications $MN$, where $M$ and $N$ are weakly normal, $M$ not being an abstraction. In particular, weakly-normal closed terms are the closed abstractions.

The purpose of evaluation is to obtain a value from a program. Only closed terms correspond to programs, entities which can run by themselves. In pure $\lambda$-calculus, a value is a closed abstraction. In an extended $\lambda$-calculus, the set of values also contains constants such as numbers, Booleans and arithmetic operators, and
partial applications, such as the term $\text{plus} \ M$, for all $M$, which needs a second argument to be evaluated.

However, weak $\beta$-reduction does not give a deterministic reduction order: given an application, should the left or the right subterm be reduced first, and to what point?

**Weak reduction strategies**

There are two weak reduction strategies for terms: call by value and call by name. **Call by value** is a new restriction for $\beta$-reduction, with the rules

$$(\text{red}_s) : \ (\lambda x \ M)V \triangleright M[x := V]$$

$$(1v) : \ \frac{M \triangleright M'}{MV \triangleright M'V} \quad (2) : \ \frac{N \triangleright N'}{MN \triangleright MN'}$$

if $V$ is a value

This strategy is easy to implement: most functional languages use it. Its major fault is its ‘eagerness’: when presented with an application $MN$, it evaluates both $M$ and $N$, even if the value of $N$ is not needed, or if computing it is harmful (if $N$ is not normalizable).

To avoid this problem, it suffices to delay the evaluation of $N$ by passing it directly as an argument to an abstraction: this is called **call by name**, another restriction on $\beta$-reduction, with only one context passing rule:

$$(\text{red}) : \ (\lambda x \ M)N \triangleright M[x := N]$$

$$(1) : \ \frac{M \triangleright M'}{MN \triangleright M'N}$$

Call by value is easier to implement in a machine with environments. It is the solution used by the majority of languages, from C to Lisp, ML and FP. Call by name is normally implemented on graph reduction machines; this is done for Lazy ML, Haskell and Miranda. It produces better-defined programs.

**Evaluation rules**

The judgment $M \triangleright M'$ says only that $M$ reduces to $M'$. To evaluate $M$, one must successively reduce $M \triangleright M'$, then $M' \triangleright M''$, ... until a value is produced. It is preferable to construct a judgment $M \rightarrow V$ which directly expresses that $V$ is the value of $M$.

Furthermore, the substitution which appears in the contraction of a redex is not a free operation. It is replaced by a transformation of the environment. Instead of reducing a term, it will be evaluated, and a non-closed term can only be evaluated in an environment where its free variables are bound.

Since Landin [92], the following definitions are standard in programming.
• A binding is a pair \((x, v)\), where \(x\) is a variable and \(v\) a value.
• An environment is a list of bindings.
• A closure is a pair \((M, \rho)\), where \(M\) is a term, and \(\rho\) an environment containing a binding for each free variable of \(M\). A closed term is a closure (in any environment) and a closure generates a closed term through repeated substitutions.
• A value is a closure \((M, \rho)\), where \(M\) is normal for weak \(\beta\)-reduction. In particular, a closed abstraction is a value.

Reduction rules are applied to \(\lambda\)-terms, but evaluation rules treat terms 'in an environment'. The judgment \(\rho \vdash M \rightarrow v\) is introduced to express that, in the environment \(\rho\), the term \(M\) has value \(v\).

Only the evaluation rule for an application is given. For call by value,

\[
\text{(App)}: \frac{\rho \vdash M \rightarrow (\lambda x M', \rho') \quad \rho \vdash N \rightarrow v \quad (x, v); \rho' \vdash M' \rightarrow v'}{\rho \vdash MN \rightarrow v'}
\]

For call by name, the notion of value must be extended to arbitrary closures: environments are therefore formed of bindings of the form \((x, F)\), where \(F\) is a closure:

\[
\text{(App)}: \frac{\rho \vdash M \rightarrow (\lambda x M', \rho') \quad (x, (N, \rho)); \rho' \vdash M' \rightarrow v'}{\rho \vdash MN \rightarrow v'}
\]

Note that, in both cases, the evaluation of \(M'\), the body of the abstraction, is done in the environment \(\rho'\), extended by a new binding because of parameter passing, and not in \(\rho\). This is characteristic of static or lexical binding. If \(\rho\) were used instead of \(\rho'\), the result would be dynamic binding. LISP1.5, like many interpreted languages, used dynamic binding. Today, new Lisps, such as Scheme and Common Lisp, use static binding.

These evaluation rules are perfectly executable. They can easily be translated into a program. They constitute the dynamic semantics, expressed in the formalism of structural operational semantics, due to G.D. Plotkin, and developed under the name of natural semantics at the French INRIA.

**Evaluation of data structures**

Pairs are introduced in the \(\lambda\)-calculus by the rules:

1. if \(M\) and \(N\) are two terms, then \((M, N)\) is a term;
2. if \(M\) is a term then \(\pi^1 M\) and \(\pi^2 M\) are terms.

Pairs allow one to construct all sorts of data structures, such as binary trees and lists, just as in Lisp.

From the point of view of reductions, the introduction of pairs creates a new kind of redex, terms of the form \(\pi^i(M, N)\), which contract naturally:

\[
\pi^1(M, N) \rightarrow M, \quad \pi^2(M, N) \rightarrow N.
\]
When this notion of contraction is done within a context, four new reduction rules are introduced (two for pair formation, one for \( \pi^1 \), one for \( \pi^2 \)), along with the \( \beta \)-reduction rules. The notion of value must be extended to pairs: there are at least two possibilities, which define strict pairs and lazy pairs. In each case, contraction and how evaluation is done in context must be restricted.

For strict pairs, a pair \((M, N)\) is a value if \(M\) and \(N\) are values, and \(\pi^i M\) is a value if \(M\) is a value which is not a pair. This choice is taken by most programming languages (see Table 3.1), which implement call by value.

\[
\begin{align*}
\text{(pair)} & : \frac{\pi^1(U, V) \triangleright U}{(U, V)} \quad \quad \text{(pair)} : \frac{\pi^2(U, V) \triangleright V}{(U, V)} \\
(1, -) & : \frac{M \triangleright M'}{(M, N) \triangleright (M', N)} \\
(\pi^1) & : \frac{M \triangleright M'}{\pi^1 M \triangleright \pi^1 M'} \\
(\pi^2) & : \frac{M \triangleright M'}{\pi^2 M \triangleright \pi^2 M'} \\
\text{if } U \text{ and } V \text{ are values}
\end{align*}
\]

**Table 3.1 Reduction of strict pairs**

For lazy pairs, or non-strict pairs, every pair \((M, N)\) is a value. These lazy pairs are similar to abstractions for weak \( \beta \)-reduction: they are values, whatever their contents. They can be implemented by placing strict pairs within an abstraction. Their reduction rules are similar to the rules used by programming languages that implement call by name.

\[
\begin{align*}
\text{(pair)} & : \frac{\pi^1(M, N) \triangleright M}{\pi^1(M, N)} \\
\text{(pair)} & : \frac{\pi^2(M, N) \triangleright N}{\pi^2(M, N)} \\
(\pi^1) & : \frac{M \triangleright M'}{\pi^1 M \triangleright \pi^1 M'} \\
(\pi^2) & : \frac{M \triangleright M'}{\pi^2 M \triangleright \pi^2 M'} \\
\text{if } M \text{ is not a pair}
\end{align*}
\]

**Table 3.2 Reduction of lazy pairs**

Lazy pairs (see Table 3.2) are interesting as they allow for the incremental evaluation of objects, which might normally be infinite. As an example program, the Sieve of Erastothenes, which generates all the primes, will be given in the form of a rewriting system.
Lazy reduction of first-order terms

Consider the following rewriting system:

\[
\begin{align*}
\text{primes} & \rightarrow \text{sieve}(\text{integers}(2)) \\
\text{integers}(x) & \rightarrow \text{cons}(x, \text{integers}(s(x))) \\
\text{sieve}(\text{cons}(x, l)) & \rightarrow \text{cons}(x, \text{sieve}(\text{filter}(x, l))) \\
\text{filter}(m, \text{cons}(x, l)) & \rightarrow \text{if } m \text{ divides } x \text{ then } \text{filter}(m, l) \text{ else } \text{cons}(x, \text{filter}(m, l))
\end{align*}
\]

These rules state that \(\text{filter}(m, l)\) removes from list \(l\) the multiples of the integer \(m\); \(\text{integers}(x)\) is the list of integers \(\geq x\); \(\text{primes}\) is the list of prime numbers: \(2, 3, 5, 7, 11, \ldots\). It is supposed that appropriate rules have been given for \textit{divide} and for \textit{if then else}.

Consider the reduction of \(\text{primes}\). If the innermost redex is contracted at each step, only the first two rules are used and the following infinite reduction is produced:

\[
\begin{align*}
\text{primes} & \rightarrow \text{sieve}(\text{integers}(2)) \\
& \rightarrow \text{sieve}(\text{cons}(2, \text{integers}(3))) \\
& \rightarrow \text{sieve}(\text{cons}(2, \text{cons}(3, \text{integers}(4)))) \\
& \rightarrow \ldots \\
& \xrightarrow{\infty} \text{sieve}(2, 3, 4, 5, 6, \ldots)
\end{align*}
\]

The reduction is infinite, yet its 'limit', written informally using the symbol \(\xrightarrow{\infty}\), has not even begun computing \(\text{sieve}\). The other rules are never used, even though the third rule can be applied at every step, starting from \(\text{sieve}(\text{cons}(2, \text{integers}(3)))\). This reduction is called unfair.

\textit{Definition 3.13} A reduction is \textit{fair} if it is finite or if any rule which can be applied an infinite number of times is actually applied an infinite number of times.

If the leftmost–outhermost redex is always contracted, another reduction is obtained. It is still infinite, but fair:

\[
\begin{align*}
\text{sieve}(\text{cons}(2, \text{integers}(3))) & \rightarrow \text{cons}(2, \text{sieve}(\text{filter}(2, \text{integers}(3)))) \\
& \rightarrow \text{cons}(2, \text{sieve}(\text{cons}(3, \text{filter}(2, \text{integers}(3))))) \\
& \rightarrow \text{cons}(2, \text{cons}(3, \text{sieve}(\text{filter}(3, \text{filter}(2, \text{integers}(4)))))) \\
& \rightarrow \ldots
\end{align*}
\]

Even though this reduction is infinite, and \(\text{primes}\) is not normalizable, the computation is done in an incremental manner,

\[\xrightarrow{\infty} \text{primes} \rightarrow 2, 3, 5, 7, 11, \ldots\]
in the sense that, for all \( n \), there exists a \( p \) such that the \( n \) first primes appear at the head of the term obtained at the end of \( p \) steps.

What has been used is the analogue to call by name in the \( \lambda \)-calculus. To avoid an infinite computation, this strategy must be restricted by making \( \text{cons} \) a lazy constructor, i.e., by allowing terms \( \text{cons}(M, L) \) as values, no matter what the terms \( M \) and \( L \) are. Lazy reduction consists of contracting the outermost redexes not contained in a \( \text{cons} \).

The reduction of \( \text{primes} \) stops at \( \text{cons}(2, \text{sieve}(\text{filter}(2, \text{integers}(3)))) \). To obtain the successive elements of \( \text{primes} \), the reduction inside the \( \text{cons} \) must be explicitly forced, with the rules

\[
\text{hd}(\text{cons}(M, L)) \rightarrow M, \quad \text{tl}(\text{cons}(M, L)) \rightarrow L.
\]

The second prime number is obtained by:

\[
\text{hd}(\text{tl}(\text{primes})) \rightarrow \text{hd}(\text{tl}(\text{sieve}(\text{cons}(2, \text{integers}(3))))))
\]
\[
\rightarrow \text{hd}(\text{cons}(2, \text{sieve}(\text{cons}(3, \text{filter}(2, \text{integers}(3))))))
\]
\[
\rightarrow 3.
\]

### 3.3.2 Call by pattern matching

\( \beta \)-reduction models a procedure call in programming. However, there are more intelligent mechanisms, based on pattern matching, which are used in functional programming languages. A rewriting system can be interpreted as a non-deterministic program, each term determining, according to its form, the rule which must be used or fired.

This mechanism can be introduced in the framework of the \( \lambda \)-calculus. Take a signature \( \Sigma \) (single-sorted to simplify) and consider the \( \lambda \Sigma \)-terms formed from variables and constants, and using symbols from \( \Sigma \), as well as application and abstraction:

- if \( M \) and \( N \) are \( \lambda \Sigma \)-terms, then \( (MN) \) is a \( \lambda \Sigma \)-term;
- if \( m \) is a \( \Sigma \)-term (a pattern) and \( P \) a \( \lambda \Sigma \)-term, then \( \lambda m \ P \) is a \( \lambda \Sigma \)-term;
- if \( f \in \Sigma \) and \( \text{ar}(f) = r \), if \( M_1, \ldots, M_r \) are \( \lambda \Sigma \)-terms, then \( fM_1 \ldots M_r \) is a \( \lambda \Sigma \)-term;
- every variable and every constant of \( \Sigma \) is a \( \lambda \Sigma \)-term.

For example, if \( \Sigma \) contains a binary pair forming symbol \( (\_, \_) \), terms such as \( \lambda(x, (y, 0)) \cdot (x, x(yx)) \) can be formed.

The contraction of a redex is done by pattern matching,

\[
(\lambda m \ P) Q \rightarrow \theta P
\]
if \( m \leq Q \) and \( \theta \) is the matcher of \( m \) to \( Q \).

Hence

\[
[\lambda(x, (y, 0)). (x, x(yx))](R, (S, 0))
\]

is a redex yielding \((R, R(SR))\), but

\[
[\lambda(x, (y, 0)). (x, x(yx))]|I
\]

is not a redex, since \((x, (y, 0)) \not\in I\).

For reasons of efficiency and decidability, patterns \( m \) must be linear, i.e., a variable name can only occur once. Hence, terms such as \((\lambda(x, x). M)(P, Q)\), whose contraction would require an equality test on \( P \) and \( Q \), are not allowed. A programming language can only reasonably be expected to test textual equality \((P \equiv Q)\) or equality of representations in memory (equality of two pointers of the same type). These two kinds of equality are not satisfactory from a programming point of view. The equality between values of \( P \) and \( Q \) \((P \equiv_\beta Q)\) is undecidable in general. In particular, the extensional equality between two functions is undecidable.

### 3.3.3 The principal type scheme of a \( \lambda \)-term

A type inference system was presented for \( \lambda \)-calculus terms and simple types, which are terms with variables over a signature \( A \cup \{ \rightarrow \} \). Write \( T \) for the set of these types. It can be shown that every typable closed term has a most general type, unique up to renaming of its type variables, i.e., it is an element of \( T \). Instead of associating a type with a term, it is better to associate a set of types. For example, \( I \) has as types all of the instantiations of the form \( \xi \rightarrow \xi \), a set which can be written \( (\xi) \cdot (\xi \rightarrow \xi) \). The typed terms remain monomorphic, so this choice remains a convenience.

Despite what was stated previously, the term \( \text{let val } x = Q \in P \) end is not exactly an abbreviation for the redex \((\lambda x P)Q\). The difference is that in a redex, all occurrences of \( x \) in \( P \) must have the same type, while in a \( \text{let} \), they can have distinct types. This allows one to write terms such as: \( \text{let val } i = \lambda x x \text{ in } (i \text{ true}, i \text{ 2}) \) end It is the \text{let} form which introduces polymorphism in the system of simple types. The inference system must be modified to take this into account, by introducing type schemes.

If \( \tau \) is a type, and if \( \{\xi_1, \ldots, \xi_n\} \) is a set of variables of type \((n \geq 0)\), then

\[
(\xi_1 \ldots \xi_n) \cdot \tau
\]

is a type scheme, representing the set of types \( \theta(\tau) \), for all substitutions \( \theta \) whose domains are included in \( \{\xi_1, \ldots, \xi_n\} \). The following properties hold, among others:

\[
() \cdot \tau = \{\tau\},
\]

\[
(\xi) \cdot \tau = \{\tau\} \quad \text{if } \xi \not\in \text{var } \tau,
\]

\[
(\xi) \cdot \xi = T.
\]
Note that by equating the type \( \tau \) with the type scheme \((\cdot) \cdot \tau\), types can be considered to be particular cases of type schemes. These schemes are also more general than type classes for the \( \sim \) of renaming. The class in \( \bar{T} \) of a type \( \tau \in T \) is the type scheme \([\tau] = (\xi_1 \ldots \xi_n) \cdot \tau\), where \( \{\xi_1, \ldots, \xi_n\} = \text{var}(\tau) \). On the other hand, a type scheme is not necessarily closed by renaming type variables, since every type is a type scheme according to the above equality. Finally, if \( \sigma' \subseteq \sigma \), \( \sigma' \) is a generic instance of \( \sigma \). For example, \( (\eta) \cdot (\eta : b) \rightarrow a \) is a generic instance of \( (\xi) \cdot (\xi : a) \), where \( a \) and \( b \) are type constants.

The \( \lambda \)-terms continue to be typed by types, but in a typing context, a list of declarations \( \alpha : \sigma \) is allowed, where \( \alpha \) is a \( \lambda \)-calculus variable and \( \sigma \) is a type scheme. Table 3.3 gives the Damas–Milner system, where \( \text{gen}_C(\tau) \) is the type scheme \((\bar{\xi}) \cdot \tau\) and \( \bar{\xi} \) is formed of the variables of \( \tau \) which have no free occurrence in the schemes figuring in the typing context \( C \).

The \((\text{var})\) rule allows a bound variable \( i \) to have two distinct types. For example:

\[
i : (\xi) \cdot (\xi : a) \vdash i : \text{bool} \rightarrow \text{bool}
\]

from which the type for \text{let} \text{val} \( i = \lambda x x \in (i \text{true}, i2)\text{end} \) is computed:

\[
(\text{let}) : \frac{\vdash \lambda x x : \xi \rightarrow \xi \quad i : (\xi) \cdot (\xi : a) \vdash (i \text{true}, i2) : \text{bool} \ast \text{nat}}{\vdash \text{let} i = \lambda x x \in (i \text{true}, i2)\text{end} : \text{bool} \ast \text{nat}}
\]

\[
(\text{var}) : \frac{x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x_i : \tau_i}{\text{if } \tau_i \in \sigma_i}
\]

\[
(\text{App}) : \frac{C \vdash M : \tau \rightarrow \tau' \quad C \vdash N : \tau}{C \vdash MN : \tau'}
\]

\[
(\lambda) : \frac{\{x : \tau\} \cup C \vdash M : \tau'}{C \vdash \lambda x M : \tau \rightarrow \tau'}
\]

\[
(\text{let}) : \frac{C \vdash Q : \tau \quad \{x : \text{gen}_C(\tau)\} \cup C \vdash P : \tau'}{C \vdash \text{let} x = Q \text{ in } P \text{ end} : \tau'}
\]

Table 3.3 The Damas–Milner type inference system

### 3.3.4 ML, a typed functional language

ML started out as the metalanguage for the proof system LCF (Logic for Computable Functions) due to D. Scott and developed by R. Milner at Edinburgh in 1978 [110]. It was designed to define proof strategies, by manipulating objects of type ‘theorem’, ‘rule’, … . The ideas essentially go back to Landin, whose ISWIM (If You See What I Mean) language can be considered to be the ancestor of functional languages.
ML has since become the prototype of functional languages. There are several versions: Standard ML (Cambridge, UK, and Bell Labs, USA), LML (Gothenburg, Sweden) and CAML (Paris, France). Lazy ML is a lazy version, in which programs are compiled to a (virtual) supercombinator graph reduction machine. The name CAML comes from its compilation to the CAM (Categorical Abstract Machine), developed by P.-L. Curien and G. Cousineau. The dialect used here is Standard ML of New Jersey, written by A. Appel and D. MacQueen.

The applicative language
ML is, first of all, an applicative language. A program is a sequence of definitions, each introducing a name along with its value, and expressions:

```ml
(* all the commands, definitions and expressions to evaluate *)
(* end with a semicolon *)
let val a = 1 and b = 2 and c = 3
in b*b - 4*a*c
end;
```

val it = 8 : int  (* value and type returned by SML *)

The execution of a program consists of constructing an environment, then of evaluating expressions in the new environment. The evaluation mechanism has no side effects on the environment. Only declarations, local to an expression (preceding example), or global, as in

```ml
val pi = 3.1415926535 ;
```

val pi = 3.1415926535 : real

can change the environment. The latter consists of name–value bindings, unlike many languages where names are bound to locations which can contain values.

Numbers, Booleans, strings and lists  The power of the language comes from the expressions which can be constructed. The usual expressions of programming languages are available. Here are some examples:

Numbers:

```ml
5 ;
```

val it = 5 : int  (* the value of 5 is 5, of type int *)

```ml
3.5 ;
```

val it = 3.5 : real  (* a floating point number *)

The syntax of arithmetic expressions is infix, as is usual:

```ml
((2 + 7)*5) + (2*(3 div (1+1))) ;
```

Booleans:

```ml
(false orelse true) andalso true ;
```
Tuples:

\( (1,2) \);

\( \sim \text{val it} = (1,2) : \text{int} \times \text{int} \)

#1(1,2);

\( \sim 1 : \text{int} \)

#2(1,2);

\( \sim 2 : \text{int} \)

(1,2,3);

\( \sim \text{val it} = (1,2,3) : \text{int} \times \text{int} \times \text{int} \)

Strings:

"here " "is very ordinary Standard ML" ;

(* concatenation, infix operation *)

\( \sim \text{val it} = "\text{here is very ordinary Standard ML}" : \text{string} \)

Lists:

\[ [1,2,3,4] \);

\( \sim \text{val it} = [1,2,3,4] : \text{int list} \)

hd [1,2,3,4] ;

(* hd is the equivalent of Lisp's car *)

\( \sim \text{val it} = 1 : \text{int} \)

tl [1,2,3,4] ;

(* tl is the equivalent of Lisp's cdr *)

\( \sim \text{val it} = [2,3,4] : \text{int list} \)

1 :: [2,3,4] ;

(* infix list constructor *)

\( \sim \text{val it} = [1,2,3,4] : \text{int list} \)

[1,2,3] @ [4,5] ;

(* list concatenation *)

\( \sim \text{val it} = [1,2,3,4,5] : \text{int list} \)

[] ;

(* the empty list, generic constant *)

\( \sim \text{val it} = [] : 'a \text{ list} \)
Functions ML is also, and particularly, a functional language, in the sense that it allows the construction of ‘functions’, by abstraction of an expression with respect to a variable, as in \( \lambda \)-calculus. The syntax of functional constructions in ML is that of the \( \lambda \)-calculus, with its simplifications (left-associativity of applications, right-associativity of abstractions). One writes \( \texttt{fn} \) instead of ‘\( \lambda \)’, and \( \Rightarrow \) instead of ‘\( \cdot \)’:

\[
\text{val discriminant} = \text{fn} (a, b, c) \Rightarrow b \cdot b - 4 \cdot a \cdot c ;
\]

\[
\sim \text{val discriminant} = \text{fn} : \text{int} \cdot \text{int} \cdot \text{int} \Rightarrow \text{int}
\]
or, more simply,

\[
\text{fun discriminant} (a, b, c) = b \cdot b - 4 \cdot a \cdot c ;
\]

The value of a function, which cannot be printed, is represented by a \( \text{fn} \) in front of its type. Local declarations are in fact redxes, for which the \( \text{let} \) form is an alternate notation, introduced by Landin in the ISWIM language:

\[
(\text{fn} \ x \Rightarrow x + 1) \ 4 ;
\]

\[
\text{let val x = 4 in x + 1 end ;}
\]

CAML also allows the form

\[
x + 1 \ \text{where} \ x = 4 ; ;
\]

it is, too, a notation used in ISWIM.

As for any functional language, binding is static or lexical, i.e., the value associated with a name is determined by the environment in which the name is introduced, and not by the environment in which it is used. The use of this form of binding means that when a function is defined, the current environment must be packaged in a \text{closure}, along with the code implementing the function:

\[
\text{let val x = 0}
\]

\[
\text{in let fun f y = x + y}
\]

\[
\text{in let val x = 2}
\]

\[
\text{in f(0)}
\]

\[
\text{end end end ;}
\]

\[
\sim \text{val it = 0} : \text{int}
\]

Argument passing is done \text{by value}, and is guided by \text{pattern matching}: the redex

\[
\text{let val (x,y) = (2,3) in x + y end ;}
\]

\[
\sim \text{val it = 5} : \text{int}
\]
is equivalent to

\[
(\text{fn} (x,y) \Rightarrow x + y) (2,3) ;
\]
Recursion The type system does not allow the definition of fixpoint combinators as does Lisp. There is therefore a primitive construct in the language for recursive definitions, called $\text{val rec}$:

$$\text{val rec } f = \text{fn } x \Rightarrow \text{if (x=0) then 1 else } x \ast f(x-1);$$

A derived form of the same definition would be

$$\text{fun f x = if (x=0) then 1 else } x \ast f(x-1);$$

Since general recursion has been introduced, ML programs do not necessarily terminate.

An example of mutual recursion is:

$$\text{fun even } x = \text{if } x=0 \text{ then true else odd(x-1)}$$
$$\text{and odd } x = \text{if } x=0 \text{ then false else even(x-1);}$$

The type system

ML is a strongly typed language, i.e., every name is typed (in a functional Lisp, such as Scheme, objects are typed, but not the names which denote them). However, it is not necessary to declare the type of a name when it is introduced, as must be done in Pascal when a variable’s name is being declared: the expression defining the name’s value can be used to infer the type. This inference is done at compilation time: it is a static process, based on the analysis of the text of the program (it is the expression which is used, not its value, which can only be known at execution time). The first phase of the semantic analysis of a program is to infer a type. If the compiler can not infer a type, the definition must be incorrect. However, the error messages are not always helpful:

$$\text{fun f x = x x ;}$$

$\triangleright$ std_in:2.11-2-13 Error: operator is not a function

operator: 'Z
in expression:

x x

Similarly, the fixpoint combinators $Y_C$ and $Y_T$ cannot be typed.

Predefined constructors The type system contains binary type constructors ($\rightarrow$, $\ast$), unary ones (list), as well as type constants, such as bool (along with the two constants of that type, true and false), int, real, string (character strings, written between ""), and unit (with () as the constant and only value of that type).

First-order terms over this signature are types; in particular, because of types such as int $\rightarrow$ (int $\rightarrow$ int), a 'higher-order' functional language is achieved.

The unary type constructor list is predefined, with a generic constant [] (the empty list), a constructor :: ('cons') and two selectors hd and tl analogous to car and cdr in Lisp. For example, int list is the type of lists of numbers.
The \( n \)-products of types are written using \(*\); it is an \( n \)-ary type constructor. As for \( n \)-tuples, they are generated by putting \( n \) expressions in parentheses with surrounding parentheses. \("Hello", "world", 3\) is a triple of type \( \text{string} \times \text{string} \times \text{int} \). For each \( n \), there is a \( \#n \) operator to obtain the \( n \)-th field. \( N \)-tuples are special cases of records.

Type abbreviations can be declared as follows:

\[
\text{type relation} = \text{int} \times \text{int} \to \text{bool} ;
\]

\[\sim\text{type relation} = \text{int} \times \text{int} \to \text{bool}\]

Furthermore, types can contain type variables, written \('a, 'b, \ldots\), which allows the definition of generic objects:

\[
\text{type 'a relation} = 'a \times 'a \to \text{bool} ;
\]

Concrete types Finally, new types, called concrete types, can be defined. The declaration

\[
\text{datatype direction} = \text{south} | \text{east} | \text{north} | \text{west} ;
\]

\[\sim\text{datatype direction}
\begin{align*}
\text{con east} : \text{direction} \\
\text{con north} : \text{direction} \\
\text{con south} : \text{direction} \\
\text{con west} : \text{direction}
\end{align*}
\]

introduces the type \text{direction}, with four new values which are type constants. Call by pattern can be extended from predefined types to concrete types:

\[
\text{fun mirror} \ \text{south} = \text{north} \\
\text{mid mirror} \ \text{east} = \text{west} \\
\text{mid mirror} \ \text{north} = \text{south} \\
\text{mid mirror} \ \text{west} = \text{east} ;
\]

Recursive types can be defined by defining their constructors. Here are the natural numbers:

\[
\text{type natural} = \text{Zero} | \text{Succ of natural} ;
\]

and generic lists, which will be redefined using the name \text{List} (\text{list} being predefined):

\[
\text{datatype 'a List} = \text{Nil} | \text{Cons of 'a} \ * \ 'a \ \text{List} ;
\]

\[\sim\text{datatype 'a List}
\begin{align*}
\text{con Cons} : 'a \ * 'a \ \text{List} \to 'a \ \text{List} \\
\text{con Nil} : 'a \ \text{List}
\end{align*}
\]

Concrete types allow an algebraic style of programming with control structures particular to a datatype, to traverse or iterate, among other things.
Higher-order functions
A functional language must allow the writing of 'higher-order' functional programs, as in λ-calculus. There are several reasons:

- writing combinators, either general (composition, currying, ... ) or particular to a given datatype (iteration over a list);
- conceiving general reusable programs, which is one of the objectives of software engineering (for example a sorting algorithm using an order relation as a parameter; this can be done using a functional parameter in Pascal or a generic procedure in Ada);
- showing the higher-order nature of a computation even if its result is of type int -> int.

Combinators  Here is a higher-order function, composition, whose definition shows the utility of infix notation:

```
inf ix o
fun f o g = fn x => f(g(x)) ;
```

```
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

The operation o can be used in infix form (i.e., f o g) or in prefix form, as for any other function, if it is preceded by op: op o (f,g). The following combinators are well known in λ-calculus:

```
fun I x = x ;  (* the identity I ≡ λx.x *)
```

```
val I = fn : 'a -> 'a
```

```
fun K x y = x ;  (* K ≡ λxy.x *)
```

```
val K = fn : 'a -> 'b -> 'a
```

The K combinator is a higher-order generic functional value. It is not a function of two variables, rather of one variable, and K 0, for example, is a function (the constant function of value 0): the parentheses of K x y is (K x) y. It is generic, in the sense that one can evaluate K 0 or K [] just as easily as K ("qwertv", [1;2;3]).

One can define an operator which takes an uncurried function and produces a curried function, as well as the reverse:

```
fun curry f x y = f(x,y) ;
fun uncurry g (x,y) = g x y ;
```

Here is the successor function:

```
val succ = (curry (op +)) 1 ;
```
Concrete types and iterators  Iteration combinators linked to concrete types will now be examined, first with the natural integers:

datatype natural = Zero | Succ of natural ;

A definition using patterns over the integers uses the iterator:

fun natural_iter z s = let fun f Zero = z

| f (Succ n) = s (f n)

in f end ;

val natural_iter = fn : 'a -> ('a -> 'a) -> natural -> 'a

This iterator allows one to translate the elements of the concrete type natural to predefined integers:

val natural_int = natural_iter 0 succ ;

val natural_int = fn : natural -> int

natural_int (Succ(Succ(Succ Zero))) ;

3 : int

Programming an iterator corresponds exactly to reasoning by induction; this will be shown for lists. A proof by induction of a property P over lists goes as follows:

- prove P(Nil);
- prove that for each a of type 'a and for each list l of type 'a List, if P(l), then P(Cons(a,l)).

Similarly, to define a function f over lists, one needs:

- a value n for f(Nil);
- for each a of type 'a and each list l of type 'a List, knowing f(l), compute f(Cons(a,l)) with the help of function c: f(Cons(a,l)) = c(a,f(l)).

The function definition is transcribed using pattern matching:

datatype 'a List = Nil | Cons of 'a * 'a List

fun List_iter n c = let fun f Nil = n

| f (Cons(a,l)) = c(a,f(l))

in f end ;

val List_iter = fn : 'a -> ('b * 'a -> 'a) -> 'b List -> 'a

Note that the 'signature' of type 'b List is of the form n : 't and c : 'b * 't -> 't with two sorts of objects 'b and 't. Building the iterator means just 'interpreting' Nil and Cons by n and c. From a functional point of view, a list operates over functions of this type: that is how lists are defined in the higher-order typed λ-calculi such as Girard’s system F. The above iterator is often better known as reduce, or fold-right.
Here are some simple applications of the iterator over lists: the first computes the sum of the elements of a list of integers, the second (better known as map) applies a function of one variable to all the elements of a list and forms a list of the results:

```ml
val List_sum = List_iter 0 (op +) ;

val List_sum = fn : int List -> int
  fun List_map f = List_iter Nil (fn (a,l) => Cons(f(a),l)) ;

val List_map = fn : ('a -> 'b) -> 'a List -> 'b List
```

An implementation of first-order terms When one is working with a given signature \( \Sigma \), one can construct a concrete type using \( \Sigma \), as was done for lists. The values of the introduced concrete type are then exactly the \( \Sigma \)-terms. In order to be able to work with an arbitrary signature, a general type Term is defined. A term is either a variable, associated with an integer, or the combination of a symbol and a list of terms:

```ml
type symbol = string

datatype Term = Var of int | Comb of symbol * Term List ;

val type symbol = string
val datatype Term
val con Comb : symbol * Term List -> Term
val con Var : int -> Term
```

To this definition, which introduces the constructors Var and Comb, is associated an iterator. The signature is not quite first-order, due to the use of List:

```ml
fun Term_iter v c n cs =
  let fun g (Var n) = v n
       | g (Comb(f,l)) =
        c(f, List_iter n (uncurry ((curry cs) o g)) l)
  in g end ;

val Term_iter = fn : (int -> 'a) ->
  (symbol * 'b -> 'a) ->
  'b ->
  ('a * 'b -> 'b) ->
  Term -> 'a
```

Here is a simpler iterator, less general as it does not ‘interpret’ the list constructors (the variable of type 'b and the variables n and cs of Term_iter are replaced by the type 'a List along with its constructors Nil and Cons):

```ml
fun Term_iter' v c = let fun g (Var n) = v n
                      | g (Comb(f,l)) = c(f, List_map g l)
                      in g end ;
```
\[ \text{val Term iter'} = \text{fn : (int} \rightarrow \text{'}a\text{)} \rightarrow \]
\[ (\text{symbol } \rightarrow \text{'}a\text{ List} \rightarrow \text{'}a\text{)} \rightarrow \]
\[ \text{Term} \rightarrow \text{'}a\text{ } \]

**Examples of higher-order computation** From now on, the predefined integers will be used instead of the concrete type natural. The curried form of the Ackerman function can be written simply by using the following recursive definition:

\[
\text{fun A 0 n = } n+1 \\
\mid \text{A m 0 = A (m-1) 1} \\
\mid \text{A m n = A (m-1) (A m (n-1))} ;
\]

Although the Ackerman function is of type \(\text{int} \rightarrow \text{int} \rightarrow \text{int}\), hence first-order, the algorithm which computes it is not first-order. A generic iteration operator It over the integers is introduced, analogous to the natural iter defined above: given a type variable \(\text{'a}\), it takes as arguments \(z : \text{'a}\) and \(s : \text{'a} \rightarrow \text{'a}\) to produce a function \(F : \text{int} \rightarrow \text{'a}\) which satisfies \(F n = s^n z\):

\[
\text{fun It z s = let fun F 0 = z} \\
\mid \text{F n = s(F(n-1))} \\
\text{in F end ;}
\]

\[ \text{\text{val It} = fn : \text{'a} \rightarrow (\text{'a} \rightarrow \text{'a}) \rightarrow \text{int} \rightarrow \text{'a}} \]

Reconsider Ackerman's function. It can be proved by double induction that \(A(m+1)n = (A(m))^n 1\). One can then use the following, concise definition of \(A\), which emphasizes the double iteration:

\[
\text{val A = It succ (fn f => It (f 1) f) ;}
\]

Note the polymorphism of \(\text{It}\) in this definition: the first occurrence is of (fourth-order) type

\[
(\text{int} \rightarrow \text{int}) \rightarrow \\
((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})) \rightarrow \\
\text{int} \rightarrow (\text{int} \rightarrow \text{int}),
\]

and the second is of (third-order) type

\[
\text{int} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}
\]

A recursor, or recursion operator, can also be introduced. It is the \(\text{R}\) of Gödel's system \(T\):

\[
\text{fun R z g = let fun F 0 = z} \\
\mid \text{F n = g (F(n-1)) (n-1)} \\
\text{in F end ;}
\]

\[ \text{\text{val R} = fn : \text{'a} \rightarrow (\text{'a} \rightarrow \text{int} \rightarrow \text{'a}) \rightarrow \text{int} \rightarrow \text{'a}} \]

The recursion operator is generic, with a type variable \(\text{'a}\), and is applied to \(z : \text{'a}\) and \(g : \text{'a} \rightarrow \text{int} \rightarrow \text{'a}\). When \(\text{'a}\) is \text{int}, the classical primitive recursion
operator is produced. It is first-order, and here is its decurryfied version:

\[
\begin{align*}
F(0) &= z, \\
F(n+1) &= g(F(n), n).
\end{align*}
\]

The usual definition contains a parameter, often a tuple of integers, which corresponds to the case where one adds a variable p to z (which becomes a function), \(R\) and g.

A higher-order type The preceding examples showed higher-order functions, but the concrete types used, lists, terms and trees, are only first-order: they are first-order terms over a many-sorted signature. Here is an example of a higher-order concrete type that resembles the description of the fundamental sequences of countable ordinals. This analogy will be used to explain the construction of this type and the ‘arithmetic’ operations.

An ordinal is zero, the successor of an ordinal or the limit of a sequence of ordinals:

\[
\text{datatype ord} = \text{Zero} \mid \text{Succ of ord} \mid \text{Lim of (int -> ord)} ;
\]

\[
\text{datatype ord}
\]

\[
\begin{align*}
&\text{con Lim : (int -> ord) -> ord} \\
&\text{con Succ : ord -> ord} \\
&\text{con Zero : ord}
\end{align*}
\]

Finite ordinals correspond to natural numbers, by iteration of Succ over Zero:

\[
\begin{align*}
&\text{fun nat n = if n=0 then Zero else Succ (nat (n-1))} ; \\
&\text{val nat = fn : int -> ord} \\
&\text{nat 4 ;} \\
\end{align*}
\]

\[
\text{(Succ (Succ (Succ (Succ Zero))))} : \text{ord}
\]

nat could have been defined using the iterator It over the integers:

\[
\begin{align*}
&\text{val nat = It Zero Succ ;} \\
\end{align*}
\]

In set theory, \(\omega\) designates the smallest infinite ordinal: it is the set of all the finite ordinals, i.e., the integers:

\[
\begin{align*}
&\text{val omega = Lim nat ;} \\
\end{align*}
\]

\[
\text{val omega = Lim fn : ord}
\]

An iterator is introduced over ord by basing its definition on the proof by induction scheme over the ordinals:

\[
\begin{align*}
&\text{fun ord_iter z s 1 = let fun F Zero = z} \\
&\quad | \text{F (Succ x) = s(F(x))} \\
&\quad | \text{F (Lim f) = l(F o f)} \\
&\quad \text{in F end ;}
\end{align*}
\]
\( \top \) \text{val ord_iter = fn : } \forall \mathcal{A} \rightarrow \\
(\forall \mathcal{A} \rightarrow \mathcal{A}) \rightarrow \\
(\forall \mathcal{A} \rightarrow \mathcal{A}) \rightarrow \mathcal{A} \rightarrow \\
\text{ord} \rightarrow \mathcal{A} \)

This iteration operator allows one to define an arithmetic over the ordinals in an analogous manner to integer arithmetic: addition is obtained by iterating the successor, multiplication by iterating addition and exponentiation by iterating multiplication. The curried versions of these functions are given:

\[
\text{fun ord_add x y = ord_iter x Succ Lim y ;}
\]

\[
\text{fun ord_mult x y = ord_iter Zero} \\
(\text{fn u => ord_add u x}) \\
\text{Lim y ;}
\]

\[
\text{fun ord_exp x y = ord_iter (Succ Zero)} \\
(\text{fn u => ord_mult u x}) \\
\text{Lim y ;}
\]

By using type \text{ord}, one can define a hierarchy of \textit{numeric} functions, by repeated iterations and diagonalizations, generalizing the construction of Ackerman’s function. In fact, the \textit{fast hierarchy} is reproduced: the family, indexed by the ordinals, consisting of more and more complex functions, is given by:

\[
\begin{align*}
  f_0(n) &= n + 1, \\
  f_{\alpha+1}(n) &= f^{n+1}_\alpha(n), \\
  f_{\lim \alpha}(n) &= f_\alpha(n).
\end{align*}
\]

The first function is the successor function \text{succ} over the integers. To compute \( f_{\alpha+1}(n) \), an iterated iterator \text{It2} is used over the integers:

\[
\text{fun It2 f n = It (f n) f n ;}
\]

\[
\text{fun It2 = fn : (int -> int) -> int -> int}
\]

To compute \( f_{\lim \alpha}(n) \), the diagonal combinator is used:

\[
\text{fun diag f n = f n n ;}
\]

\[
\text{fun diag = fn : (\forall \mathcal{A} \rightarrow \forall \mathcal{A} \rightarrow \mathcal{B}) \rightarrow \forall \mathcal{A} \rightarrow \mathcal{B}}
\]

\[
\text{val hierarchy = ord_iter succ It2 diag ;}
\]

\[
\text{val hierarchy = fn : ord -> int -> int}
\]

Ackermann’s function is very low in the hierarchy: \( \omega \) is the first infinite ordinal. There are many others. The ordinal \( \varepsilon_0 \) became very famous in 1936 when Gentzen proved the consistency of Peano’s arithmetic, by transfinite induction up to \( \varepsilon_0 \) (see [144]). This proof cannot be formulated in Peano’s arithmetic (Gödel’s second
incompleteness theorem). Besides, Peano’s induction axiom allows induction only over the integers, i.e., up to \( \omega \). By definition, \( \epsilon_0 \) is the limit of the sequence \( \omega_0 = 1, \omega_1 = \omega, \omega_2 = \omega^\omega, \omega_{n+1} = \omega^{\omega_n}, \ldots \). Although infinite, these ordinals correspond to perfectly ordinary types in functional programming. The sequence is generated by the local function \( f \) in the following definition:

```plaintext
val epsilon_0 =  
let val f = ord_iter (Succ Zero) (ord_exp omega) Lim 
in f omega end ;
```

This ordinal is the limit of the usual ordinals. It is the smallest ordinal \( x \) which is a solution to the equation \( x = \omega^x \).

In the fast hierarchy, \( f_\alpha(n) = f_{\omega_\alpha}(n) \) (an iteration over the ordinals was done up to \( \epsilon_0 \)); it is a (total) function, but the fact that it is defined for all the integers is not provable in arithmetic. From a practical point of view in arithmetic, all of these functions are easily programmed. They can be defined by a program, but it would be unthinkable to actually compute their values, except for a few at the beginning. For these functions, the algorithmic complexity of a program computing its values is related to the logical complexity of the termination proof of its program.

**Non-functional aspects**

Functional programming, in fact, the \( \lambda \)-calculus, has a serious limitation for the formulation of algorithms: expressions must always produce a value. The \( \lambda \)-calculus lacks a means of controlling the sequentiality of the calculus. For example, it should be possible to pop out of a recursion with a final result immediately without being forced to pass the result back through preceding recursive invocations. As in logic programming, where a goal can fail, it should be possible for a function to return no result, and to raise an exception signaling its failure. This ‘exception’ should then be treatable. This requirement is mandatory for error recuperation. Such mechanisms exist in most languages, e.g., Lisp’s exits and Ada’s exceptions.

The most general escaping mechanism, found in Scheme, allows direct access to the *continuation* of an expression, a low-level theoretical concept from denotational semantics, akin to the execution stack. The call-with-current-continuation construction which allows this access must be handled carefully; it motivated the study of extensions of the \( \lambda \)-calculus, such as the \( \lambda \nu C \)-calculus of M. Felleisen[42].

Standard ML now offers continuations. However, it also offers a solution of medium power: exceptions are declared by `exception ...`, triggered by `raise ...`, and handled selectively by pattern matching, using `... handle ...`. Exceptions are declared in a similar manner to concrete types, by introducing constant exceptions and constructors:

```plaintext
exception error of string and stop ;

catch exception error
  exception stop
```

A raised exception is propagated so long as it has not been handled:
if false then ""
else raise error "bang"
handle (error x) -> x ~ "!!!"
| stop -> "finished" ;
≡ "bang!!!" : string

Note also that Standard ML has a unary type constructor ref such that an
object of type \texttt{t ref} is a reference to an object of type \texttt{t}. Objects with a mutable
local state can be created, and purely sequential programming becomes possible
by the use of assignment \texttt{:=}, the dereferencing operator \texttt{!} and the sequencing
operator;

val place = ref 3 ;
≡ val place = ref 3 : int ref
!place ;
≡ val it = 3 : int
(place := 10 ; place := !place + 1 ; 1 ; 2)
≡ val it = () : unit
val it = () : unit
val it = 1 : int
val it = 2 : int

It is possible to construct mutable data structures. Here is a model of Lisp ob-
jects:

\[
\text{datatype lisp = Nil | Pair of lisp ref * lisp ref ;}
\]

≡ datatype lisp
con Nil : lisp
con Pair : lisp ref * lisp ref -> lisp

exception NilPair
fun set_car Nil _ = raise NilPair
| set_car (d as Pair(car,cdr)) v = (car := v ; d);
≡ val set_car = fn : lisp -> lisp -> lisp

Finally, one can program in an ‘object-oriented’ style by constructing objects which
are closures with mutable local state, i.e., closures whose environment contains
name–reference bindings, the reference value being modifiable. Here is a creator of
counters which responds to the messages 0!, ++, -- and ?

fun counter () =
  let val value = ref 0 in
  let fun F "0!"= (value := 0 ; !value)
  | F "++"= (value := !value + 1; !value)
Reduction

| F "--" = (if !value > 0  
            then value := !value - 1  
            else raise error "decrement: null value";  
                  !value)  
| F "?" = !value  
| F _ = raise error "unknown command";  
  in F end

\(\sim\) val counter = fn : unit \rightarrow\) string \rightarrow\) int
val c = counter () ;
c "++" ; c "++" ; c "--" ; c "?" ;

\(\sim\) val it = 1 : int
val it = 2 : int
val it = 1 : int
val it = 1 : int
(c "a" handle error x \rightarrow\) (print x ; print "\n" ; 1))

\(\sim\) unknown command
val it = 1 : int

Closures and delayed evaluation
The evaluation mechanism of Standard ML reproduces weak \(\beta\)-reduction with call by value. Hence, the body of functions is never evaluated. An expression can be frozen by making it an abstraction; this is not possible in non-functional languages.

In Standard ML, an expression such as

\[\text{fun Suc_r () = x+1 ;}\]

has as value a closure containing the value of \(x\) and the code necessary to increment \(x\). It is only when Suc_r() is called that \(x+1\) will be computed.

With this technique, call by name can be simulated in Standard ML.

3.3.5 Scheme, an untyped functional language

Scheme is a dialect of Lisp with static and monovalued binding. The same space is used for all names, be they 'values' or 'functions'. As for other Lisps, environments define name-memory location bindings rather than name-value bindings. Recursion and redefinition of global variables become easy. First conceived at MIT as an interpreter for the \(\lambda\)-calculus, Scheme has had considerable influence on the evolution of Common Lisp towards static binding, and is at the center of the European project of Lisp normalization.

Since the language is not typed, it is possible, as in the \(\lambda\)-calculus, to define a fixpoint combinator, which is not possible in ML, except by using some recursive concrete types:
(define Y
  (lambda(f)
    (let ([g (lambda(h) (lambda(x) ((f (h h)) x)))]
         (g g)))))

The factorial can be defined by applying Y to the ‘functional’ (lambda(f) ...):

(defun fact (n)
  (funcall (lambda (x)
             (if (zerop x) 1 (* x (funcall f (- x 1))))
          n)))

3.3.6 Compilation of λ-terms into applicative terms

What purpose do bound variables play in the λ-calculus? They are only introduced, by abstraction, to be eliminated, by application. Can one make do without them? There exists a formalism, called combinatory logic, which shows effectively that one can construct a calculus of the same expressive power as the λ-calculus, without bound variables. It is a variant of the latter, but was developed independently by Schönfinkel in 1924 [132], and later by Curry, with the intention of creating a ‘logic’ which could be manipulated in a combinatorial manner.

The debate has been taken up in the area of programming languages: is abstraction, made popular by Lisp’s lambda, useful and necessary? Backus, upon his reception of the Turing Award in 1978 [7], said no: abstraction is not necessary and reduces the clarity of programs. Backus’s presentation was particularly designed to attack languages such as Fortran which are based on the von Neumann architecture, but he was also taking a position against Lisp-like languages.
From an implementation point of view, bound variables require an environment access mechanism which makes compilation quite complex. It would be simpler if this step could be avoided.

Applicative terms and concrete syntax were introduced in Chapter 1. They are first-order terms formed of constant symbols and a single binary function symbol, called application, written using a notation similar to that of the $\lambda$-calculus. In programming, an applicative language is a language based on the evaluation of expressions (first-order terms), just as arithmetic expressions are evaluated in calculators. For these expressions, the available constants are the integers and the elementary operations ($+, \cdot, \ldots$), and there are rules to compute with these. Of course, other constants are needed to produce more complex calculi. In fact, the $\lambda$-calculus example shows that arithmetic operations can be represented as soon as a sufficiently general functional calculus is defined. Combinators which permute or duplicate the arguments of a function, or which solve a given equation, can easily be defined.

The compilation of several functional languages, such as SASL and Miranda, is based on the translation of programs which are terms of a $\lambda$-calculus extended with applicative terms constructed using $S$, $K$ and some other combinators. The resulting term, represented as a tree, and even by a graph if subterms are shared, is then reduced by the rewrite rules of combinatory logic. Since the rule associated with each combinator can be seen as an elementary machine instruction, there have been many attempts to construct a ‘graph reduction machine’. Use of such a virtual machine is also a standard implementation technique.

A mapping of $\lambda$-terms to applicative terms, $M \mapsto M_\emptyset$, will be defined; the mapping is used for compiling $\lambda$-terms. A reverse mapping, $A \mapsto A_\lambda$, is also defined. For the moment, the constants with which applicative terms are formed have not been made explicit. This will be done by associating a closed $\lambda$-term $c_\lambda$ with each constant $c$, along with a rewrite rule.

The problem is to translate abstractions into applicative terms. The first step is to simulate abstraction in combinatory logic. For each variable $x$, define an abstraction operator $A \mapsto [x].A$, associating the applicative term $A$ with the applicative term $[x].A$ representing the ‘function $x \mapsto A$’. By structural induction over $A$, the cases of a constant $c$, of an application $PQ$, and of a variable $y$ must be treated; in the latter case, $y \equiv x$ and $y \not\equiv x$ must be distinguished.

So that the simulation of abstraction accurately reflects the meaning in $\lambda$, at least one renaming property must be verified:

\[ [x].A \equiv [y].(A[x := y]) \]

if $y \not\in \text{var} A$. In particular, $[x].x$ must be independent of $x$: a constant symbol $I$ is used for this value and is defined by $I_\lambda \equiv \lambda x . x \equiv I$, with the rule $Ix \rightarrow x$, obviously derived from $Ix \rightarrow x$.

Furthermore, if $A$ is a distinct variable from $x$, if $A$ is a constant symbol (for example $I$), or more generally, if $x$ does not appear in $A$, the ‘function $x \mapsto A$’ is
a constant value function \( A \). A second constant symbol is introduced: \( K \), with \( K_x \equiv \lambda xy. x \) and the rule \( K x y \to x \) to define the case \( [x]. A \) by \( K A \).

Finally, consider an application \( AB \) in the case where \( x \in \text{var}(AB) \). A new constant symbol \( S \) is introduced to define \( [x]. AB \equiv S([x]. A)([x]. B) \). Using the standard notation of functions, the combinator applied to two functions \( f \) and \( g \) produces the function \( S(f, g) : x \mapsto f(x, g(x)) \). Hence its translation \( S_x \equiv \lambda f g x . f x(g x) \equiv S \) and the rule \( S f g x \to f x(g x) \).

Recall the definition of \( [x]. A \):

\[
\begin{align*}
[x]. x & \equiv I \\
[x]. A & \equiv KA & \text{if } x \notin \text{var } A \\
[x]. AB & \equiv S([x]. A)([x]. B) & \text{if } x \in \text{var } AB
\end{align*}
\]

the definition of the translation of \( \lambda \)-terms into applicative terms:

\[
\begin{align*}
(x)_@ & \equiv x \\
(P Q)_@ & \equiv (P)_@ (Q)_@ \\
(\lambda x. M)_@ & \equiv [x]. (M)_@
\end{align*}
\]

and the rewrite rules:

\[
\begin{align*}
Ix & \to x \\
K x y & \to x \\
S x y z & \to x z(y z)
\end{align*}
\]

These rules define weak reduction of combinatory logic, written \( \to_@ \).

The reverse translation \( A \mapsto A_\lambda \) is perfectly natural (if there were other constants, one would add their translation to those of \( K \) and \( S \)):

\[
\begin{align*}
x_\lambda & \equiv x \\
I_\lambda & \equiv I \\
K_\lambda & \equiv K \\
S_\lambda & \equiv S \\
A B_\lambda & \equiv A_\lambda B_\lambda
\end{align*}
\]

**Proposition 3.16**

- if \( A \to_@ B \), then \( A_\lambda \to_\beta B_\lambda \);
- \( (M_@)_\lambda \to_\beta M \).

The proof is trivial. Conversely, it could be that \( M \to_\beta N \) without \( M_@ \to_@ N_@ \).

**Example** \( \text{SK} \to_\beta 0 \), but \( \text{SK} \) is irreducible for \( \to_@ \).

This fact is due to the definition of the rewrite rules \( \to_@ \) which have no context passing rule similar to

\[
\frac{M \triangleright M'}{\lambda x M \triangleright \lambda x M'}
\]

Furthermore, it is not necessarily true that \( (A_\lambda)_@ =_@ A \).
Example \((K_\lambda)_a \equiv SKK)I\), which does not reduce to \(K\). However, it is true that \(SKK)Ix y \rightarrow x \leftarrow Kxy\).

It follows that the two theories, although related, are not equivalent.

One can immediately check that \((SKK)x \rightarrow_a x\), hence that the term \(SKK\) plays the same role as the constant \(I\). It therefore suffices to add the constants \(S\) and \(K\) and to define \([x].x = SKK\). The set \(\{S, K\}\) (similarly \(\{S, K, I\}\)) forms a combinatorial basis for the \(\lambda\)-calculus. There exist other bases, some with a single element. However, from a practical point of view the presented compilation rules are unusable because of the combinatorial explosion of the resulting terms.

Example Consider the complexity of the translation of the recursive definition of the factorial function:

\[ f \equiv \lambda n. \text{zero} n 1(\text{mult} n(f(\text{pred} n))). \]

No attempt will be made to solve this equation using a fixpoint combinator. Rather, the right hand side will be translated using applicative terms. Here are a few application steps of the translation rules:

\[
[n]. \text{zero} n 1(\text{mult} n(f(\text{pred} n))) \\
\equiv S([n].\text{zero} n 1)(([n].\text{mult} n(f(\text{pred} n)))) \\
\equiv S(S([n].\text{zero} n 1)([n].1))(S([n].\text{mult} n)([n].f(\text{pred} n))) \\
\equiv S(S(S([n].\text{zero})([n].n)))([n].1) \\
\quad (S(S([n].\text{mult})([n].n))(S([n].f)(S([n].\text{pred})([n].n)))) \\
\equiv S(S(S(K \text{ zero})(I))(K 1))(S(S(K \text{ mult})(I))(S(K f)(S(K \text{ pred})(I))))).
\]

To avoid this proliferation of incomprehensible applicative terms, it is possible to modify the definition of \([x].A\) by adding combinators less universal than \(S, K, I\). For example, \(B\) (respectively \(C\)) is a specialized version of \(S\) for applications whose right-hand (respectively left-hand) subterm contains the variable which is being abstracted. Their rules are:

\[
B f g x \rightarrow f(g x), \\
C f x a \rightarrow (f x)a.
\]

These new combinators allow significant optimizations. The following rules are used:

\[
S(KM)I \rightarrow_\text{opt} M, \\
S(KM)(K N) \rightarrow_\text{opt} K(M N), \\
S(KM)N \rightarrow_\text{opt} B M N, \\
S M(K N) \rightarrow_\text{opt} C M N.
\]
They can be justified by adding an extensionality rule to combinatory logic:

\[
\frac{Ax = Bx}{A = B} \quad \text{if } x \notin \text{var}(AB).
\]

See exercise 20.

Example (cont.) Here is the optimized form of the preceding recursive definition:

\[
f \equiv S(S(S(K\text{ zero})I)(K1))(S(S(K\text{ mult})I)(S(K f)(S(K \text{ pred})I)))
\]

\[
=_{\text{opt}} S(C\text{ zero}1)(S\text{ mult}(B f \text{ pred})),
\]

which is much simpler than the preceding one.

\[
\square
\]

3.3.7 Supercombinators

A more flexible translation of \(\lambda\)-terms into applicative terms, proposed by Hughes in 1982 [72], uses supercombinators.

Definition 3.14 Let \(E\) be a set of terms. The set \(\text{comb}(E)\) of (applicative) combinations of \(E\) is defined by induction:

1. \(E \subseteq \text{comb}(E)\);
2. if \(M, N \in \text{comb}(E)\), then \(MN \in \text{comb}(E)\).

A closed term \(M\) is a supercombinator of arity \(r\) if it has form \(M \equiv \lambda x_1 \ldots x_r . C\), where \(C\) is a combination of \(x_1, \ldots, x_r\) and of supercombinators; \(x_1, \ldots, x_r\) are the parameters and \(C\) is the body of \(M\).

Note that a supercombinator is a combinator, and that in the base case the body is simply a combination of the parameters.

Examples

- \((M_2 M_3)(M_1(M_3 M_1)) \in \text{comb}\{M_1, M_2, M_3, M_4\}\);
- \(S \equiv \lambda x y z . x z(y z)\) is a supercombinator of arity 3, with body \(xz(y z) \in \text{comb}(x, y, z)\);
- \(\lambda y . x y\) is not a supercombinator because it is not a combinator;
- \(\lambda x . x(\lambda y . x y)\) is a combinator but not a supercombinator since \((\lambda y . x y)\) is not one;
- \(\lambda x . K x\) is a supercombinator.

Proposition 3.17 Any term \(M\) can be transformed into a supercombinator \(M^*\) such that \(M^* x_1 \ldots x_r \rightarrow M\), if \(\{x_1, \ldots, x_r\} = \text{var}(M)\). In particular, if \(M\) is a combinator, then \(M^* \rightarrow M\).
Proof  If $M \equiv x$, let $M^* \equiv I$.

Let $M \equiv \lambda \vec{y}. C$, where $C$ is not an abstraction, so that all the leading $\lambda$ are in $\lambda \vec{y}$, and $\{\vec{x}, \vec{y}\} = \text{var}(C)$. The term $C$ is a combination of $\vec{x}, \vec{y}$ and abstractions $C_1, \ldots, C_p$ which have their free variables in $\{\vec{x}, \vec{y}\}$. By induction, there exist supercombinators $C^*_i$ such that $C^*_i \overset{\ast}{\rightarrow} C_i$, where $\vec{u}^{(i)}$ is formed with free variables of $C_i$. The term $C'$ is constructed by replacing $C_i$ by $C^*_i \vec{u}^{(i)}$ in $C$ for $i = 1, \ldots, p$; $C'$ is therefore a combination of $\{\vec{x}, \vec{y}\}$ and of supercombinators $C^*_i$. Let $M^* \equiv \lambda \vec{x} \vec{y}. C'$: it is a supercombinator. From $C^*_i \vec{u}^{(i)} \overset{\ast}{\rightarrow} C_i$, deduce $C' \overset{\ast}{\rightarrow} C$; hence, $M^* \vec{x} \vec{y} \overset{\ast}{\rightarrow} M$.  

A supercombinator $M$ can be associated with an applicative rewriting system $(\Sigma, R)$, containing a principal rule $fx_1 \ldots x_r \rightarrow \ldots$, $f$ being a new constant symbol corresponding to $M$.

Let $M \equiv \lambda \vec{x}. C$ be a supercombinator of arity $r$. Write its body $C$ as a combination of its parameters $x_1, \ldots, x_r$, and of supercombinators $S_1, \ldots, S_p$, respectively of arity $r_1, \ldots, r_p$. Suppose that one has already associated a symbol $f_i$ and a rewriting system $(\Sigma_i, R_i)$ with the supercombinator $S_i$. Let

$$\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_p \cup \{f\},$$

where the signatures $\Sigma_i$ are disjoint and $f$ is a new constant symbol, and

$$R = R_1 \cup \ldots \cup R_p \cup \{r\},$$

where $r$ is the rule $fx_1 \ldots x_r \rightarrow C_0$, the applicative term $C_0$ being obtained by replacing $S_i$ by $f_i$ in $C$.

Example  Let $M \equiv \lambda x. (\lambda y. y(\lambda zt. xt(zyt)))(\lambda yt. z(yt))$. The term $Mx$ is a combination of two applications $C_1$ and $C_2$ having $x$ as free variable. The term $C_1 x y$ is a combination $y C_3$, and $C_2 x y t = x(yt)$ and $C_3 x y z t = x t(z y)$. Let

$$C_3^* \equiv \lambda y z t. x t(z y),$$
$$C_2^* \equiv \lambda x y t. x(yt),$$
$$C_1^* \equiv \lambda x y. y(C_3^* x y),$$
$$M^* \equiv \lambda x. (C_1^* x)(C_2^* x).$$

Associate with the supercombinators $M^*, C_1^*, C_2^*$ and $C_3^*$ the constant symbols $f$, $f_1$, $f_2$ and $f_3$. The following rewriting system is obtained:

$$f^* x \rightarrow (f_1 x)(f_2 x),$$
$$f_1^* x y \rightarrow y(f_3 x y),$$
$$f_2^* x y t \rightarrow x(yt),$$
$$f_3^* x y z t \rightarrow x t(z y).$$
It would be unthinkable to ask a programmer to write all programs using a finite base of combinators such as \( \{S,K,I,B,C\} \). However, it would be possible to write definitions using only supercombinators, leading to a descending method of programming by refinement. The result would be a programming style intermediate between the pure applicative programming (no variables) and general functional programming (arbitrary \( \lambda \)-terms). In fact, the criticisms made by Backus about \( \lambda \)-languages do not touch these programs. The result could be considered as a form of 'structured functional programming'.

In fact, supercombinators are of interest for the implementer, as an intermediate language between the source and machine language. It is easy to conceive the first phase of a compiler which transforms an arbitrary term into a supercombinator. Do note that this compilation is more complex when recursion must be treated and when optimizations such as maximal sharing are done. The result is then an applicative rewriting system, bound variables having been eliminated. To write an interpreter for this intermediate language means designing a program to evaluate rewriting systems. By choosing the data structures aiding this evaluation correctly, the program can run using the operations of an abstract machine. The best actual implementation based on this method is Lazy ML, a lazy dialect of ML designed at Chalmers University of Technology in Gothenburg (Sweden).

### 3.4 Termination and confluence

If a computation is seen as a process producing a finite result, the problem of termination is crucial. This will not be the case for all processes, in particular concurrent programs, since, for some, stopping can mean deadlock or failure! Furthermore, as was shown with the lazy reduction of first-order terms, non-terminating processes may produce results by the 'increasing' evaluation of some data structure. Other properties must then be studied, such as liveness or fairness. In the 'general' case, it will be seen that the termination problem is undecidable. This in no way prevents the study of special cases. A reduction relation satisfies the (weak) normalization property if all terms are normalizable. It satisfies the strong normalization property if there are no infinite reductions.

Confluence is an even more important property. If the writing of programs which do not necessarily end is allowed, one would expect at least that, no matter what the execution strategy, the eventual result be the same. To ensure this, \( \beta \)-reduction will be shown to be confluent.

#### 3.4.1 Termination and well-orders

A rewrite rule can reduce the size visibly, be it measured in the number of symbols, the height or some other measure, of its terms. In the case of rule \((-x) \rightarrow x\), a
function \( h : T[X] \rightarrow \mathbb{N} \) (for example \( h(M) = |M| \)) is used to check that \( h(P) > h(Q) \) for any rule \((P, Q)\). To deduce that \( h(M) > h(M') \) if \( M \rightarrow M' \), it suffices to show that \( h \) satisfies the following conditions, which follow from the definition of rewriting by contraction and context passing:

1. \( h(\sigma P) > h(\sigma Q) \) for every substitution \( \sigma \) and for every rule \((P, Q)\);
2. \( h(M_i) > h(M'_i) \) implies \( h(f M_1 \ldots M_i \ldots M_n) > h(f M'_1 \ldots M'_i \ldots M_n) \).

However, a usual rule such as associativity

\[(x + y) + z \rightarrow x + (y + z),\]

cannot be treated by simply reasoning about the size of terms. The solution is to assign a heavier weight to the left subterm of a sum. A function \( h \) is defined by structural induction:

\[
\begin{align*}
    h(a) &= 1, \\
    h(M + N) &= 2h(M) + h(N),
\end{align*}
\]

where \( a \) is a constant or a variable. This inductive definition consists of giving \( \mathbb{N} \) a \( \Sigma \)-algebra structure (see Chapter 4):

\[
\begin{align*}
    a^N &= 1, \\
    +^N(x, y) &= 2x + y.
\end{align*}
\]

It is easy to see that function \( h \) satisfies conditions (1) and (2) above.

Finally, the size of terms can quite easily increase by rewriting, as the following example shows, where \( f \) and \( g \) are unary symbols:

\[f f x \rightarrow g f g x.\]

In this case, \( h(M) \) could be defined by the number of occurrences of \( f \) in \( M \).

So, it suffices to have a function \( h \) satisfying properties (1) and (2) to prove the noetherianity of a rewriting system. It is not necessary that \( h \) has its values in \( \mathbb{N} \). The only property of the integers that is used is that the order \( > \) is well-founded: there is no infinite, strictly decreasing sequence of natural integers. A function \( h \) can be sought whose range is in a well-ordered set \((E, >)\). Referring to \( E \) is a notational convenience, since one can reason directly in \( T[X] \) with the order,

\[M \succ_h N \quad \text{if} \quad h(M) > h(N).\]

Hence a well-founded order \( \succ \) is sought over the set of terms such that it satisfies properties (1) and (2): decreasing the instances of rules and context passing. This condition is also necessary because if \( \rightarrow \) is noetherian, the \( \rightarrow^* \) itself is a well-order, by the definition of noetherianity. This necessary and sufficient condition constitutes the Manna–Ness theorem [97].
Theorem 3.18 A rewriting system $\mathcal{R}$ is noetherian if and only if there exists a well-order $\succ$ over the set of terms such that

1. $\sigma P \succ \sigma Q$ for every rule $(P, Q) \in \mathcal{R}$ and every substitution $\sigma$;
2. $M_i \succ M'_i$ implies $fM_1 \ldots M_i \ldots M_n \succ fM_1 \ldots M'_i \ldots M_n$ ($\succ$ is compatible with $\Sigma$).

In practice, the external form is more easily used, i.e., with a function $h$ into a well-ordered set $E$.

3.4.2 Simplification orders

The Manna–Ness theorem is very strict: it requires a strict order $\succ$ to be well-founded over the set of terms when in fact it would suffice if it were well-founded 'along the reductions'. That there be an infinitely decreasing sequence of terms has no importance if the sequence is not a reduction.

In applications of the Manna–Ness theorem, one works with a known set $(E, \succ)$, for example $\mathbb{N}$ or $\mathcal{M}(\mathbb{N})$, and one knows once and for all that the order $\succ$ is well-founded. It is much more difficult to prove that an order is well-founded along a chain of reductions: this proof would normally depend on the rewriting system being studied. There does exist, however, a class of orders which are necessarily well-founded along reduction chains. They are the simplification orders, introduced by N. Dershowitz.

Definition 3.15 A strict order relation $\succ$ over $T[X]$ is a simplification order if for every symbol $f$ of arity $n$,

1. $M_i \succ M'_i$ implies $fM_1 \ldots M_i \ldots M_n \succ fM_1 \ldots M'_i \ldots M_n$;
2. $fM_1 \ldots M_i \ldots M_n \succ M_i$ for every $i$.

The embedding relation is defined by

$$M = fM_1 \ldots M_p \sqsubseteq gM'_1 \ldots M'_q = M'$$

if one of the following three conditions is satisfied:

1. $M = M'$;
2. $f = g$ and $M_i \sqsubseteq M'_i$ for every $i, 1 \leq i \leq p$;
3. there exists $i, 1 \leq i \leq p$ such that $M \sqsubseteq M'_i$.

Hence, $M \sqsubseteq M'$ if $M$ can be obtained from $M'$ by 'pruning' $M'$. It is a (partial) order relation. The slightly more general case of preorders is considered, for a stronger concept than well-order will be useful.

Definition 3.16 A preorder $\sqsubseteq$ is a well-preorder if for every infinite sequence of elements $(x_n)_{n \geq 0}$, there exists $i, j$ such that $i < j$ and $x_i \sqsubseteq x_j$.

Lemma 3.19 A preorder $\sqsubseteq$ is a well preorder if and only if for every infinite sequence of elements $(x_n)_{n \geq 0}$, there exists an increasing extracted sequence $(x_n)_{i \geq 0}$.
Proof. There can only be a finite set of elements \( x_i \) which are not bounded by any \( x_j, j > i \): if there were infinitely many, the definition of \( \sqsubseteq \) would imply that two of the \( x_i \) would be comparable, and the smallest would be bounded by the other. An increasing sequence is extracted by starting with an arbitrary element of the sequence, beyond that finite set, by choosing an upper bound in the sequence and by iterating the procedure. \( \square \)

There are other useful characterizations. For example, a preorder is a well-preorder if the associated strict order is well-founded and if every set of incomparable elements is finite.

**Theorem 3.20 (Kruskal)** If \( \Sigma \) is a finite signature, \( \sqsubseteq \) is a well-preorder over \( T_\Sigma \).

**Proof.** Suppose that a counterexample exists, i.e., there exists a sequence \( (M_n)_{n \geq 0} \) not satisfying the well-preorder property, which is minimal, i.e., chosen in the following manner: if \( M_0, \ldots, M_p \) are already constructed, choose for \( M_{p+1} \) a term \( C_{p+1} \) of minimal size among all the counter-examples \( (C_n)_{n \geq 0} \) such that \( C_0 = M_0, \ldots, C_p = M_p \).

Since \( \Sigma \) is finite, one can extract from \( (M_n)_{n \geq 0} \) a sequence of terms whose elements have the same head symbol, \( (f(N_{m_1}^1, \ldots, N_{m_p}^p))_{m \geq 0} \).

By the minimality assumption, the sequence \( (N_{m_i}^1)_{m \geq 0} \) cannot be a counterexample: if it were, the sequence \( M_0, \ldots, M_{s-1}, N_0^1, \ldots, N_s^1, \ldots \), where \( s \) is such that \( f(N_0^1, \ldots, N_s^1) = M_s \), would be a counterexample strictly 'smaller than' \( (M_n)_{n \geq 0} \), which is impossible according to the construction.

According to the lemma, a sequence of terms can be extracted, each embedded in the next: \( N_{m_1}^1 \sqsubseteq N_{m_2}^1 \ldots \) is the result. Consider, then, the sequence \( (f(N_{m_i}^1, \ldots, N_{m_i}^p))_{i \geq 0} \) extracted from \( (M_n)_{n \geq 0} \). The minimality assumption also applies to \( (N_{m_i}^2)_{i \geq 0} \). By iterating this extraction procedure for each of the arguments of \( f \), the sequence \( (f(M_k^1, \ldots, M_k^p))_{k \geq 0} \) is extracted from \( (M_n)_{n \geq 0} \) such that for every \( r \ (1 \leq r \leq p) \), and every \( k \geq 0 \), \( M_k^r \sqsubseteq M_{k+1}^r \).

By the definition of the embedding relation, deduce for every \( k \) that

\[
f(M_k^1, \ldots, M_k^p) \sqsubseteq f(M_{k+1}^1, \ldots, M_{k+1}^p),
\]

hence that the sequence \( (M_n)_{n \geq 0} \) has an increasing subsequence for \( \sqsubseteq \). The initial assumption over \( (M_n)_{n \geq 0} \) is contradicted. \( \square \)

One proves by induction over \( M' \) that

**Lemma 3.21** If a term \( M \) is embedded in a term \( M' \), then \( M' \gg M \) for every simplification order \( \gg \).

**Theorem 3.22 (Dershowitz)** \( R \) is noetherian if there exists a simplification order \( \gg \) such that \( \sigma P \gg \sigma Q \) for every rule \( (P, Q) \in R \) and every substitution \( \sigma \).
Proof If there exists an infinite reduction \( M_1 \rightarrow M_2 \rightarrow \ldots \), Kruskal’s theorem ensures the existence of \( i \) and \( j > i \) such that \( M_i \sqsubseteq M_j \). According to the preceding lemma, it follows that \( M_j \triangleright M_i \). Yet, by definition, \( \triangleright \) verifies conditions (1) and (2) of the Manna–Ness theorem. Therefore, \( M \rightarrow M' \) implies \( M \triangleright M' \) and, in particular, \( M_i \not\triangleright M_j \) implies \( M_i \triangleright M_j \). But \( \triangleright \) being a strict order, \( M_j \triangleright M_i \) and \( M_i \triangleright M_j \) are incompatible. \( \square \)

A necessary condition can be given for non-noetherianity: if a rewriting system is not noetherian then there exist terms \( M \) and \( M' \) such that \( M \sqsubseteq M' \) and \( M \not\triangleright M' \). Note that this necessary condition is not sufficient, as is shown by the example of rule \( ffx \rightarrow gfgx \). The noetherianity of this rule cannot be proven using a simplification order.

There are systematic methods for constructing simplification orders. They are used in rewriting systems such as REVE.

Let \( \triangleright \) be a ‘precedence’ order over the signature \( \Sigma \). Using \( \triangleright \), define the recursive path ordering \( \triangleright_{rpo} \) over \( T_\Sigma \) and the associated order \( \triangleright_{rpo} \) over \( \mathcal{M}(T_\Sigma) \), the set of multisets of terms. Write \( \simeq \) for the equivalence under permutation of subterms: \( M = \{ M_1, \ldots, M_p \} \triangleright_{rpo} \{ M'_1, \ldots, M'_q \} \) if \( M_i \triangleright M'_{\pi(i)} \) for a permutation \( \pi \) of indices \( \{1, \ldots, n\} \), and \( M \simeq_{rpo} M' \) if \( M \triangleright_{rpo} M' \) or \( M \triangleright M' \).

\[
M = \{ M_1, \ldots, M_p \} \triangleright_{rpo} \{ gM_1', \ldots, M_q' \} = M'
\]

if one of the three following conditions is satisfied:

1. **RPO1** \( f = g \) (hence \( p = q \)) and \( \{ M_1, \ldots, M_p \} \triangleright_{rpo} \{ M'_1, \ldots, M'_q \} \);
2. **RPO2** \( f \triangleright g \) and for every \( j, 1 \leq j \leq q \), \( M \triangleright_{rpo} M'_j \);
3. **RPO3** \( f \not\triangleright g \) and there exists \( i, 1 \leq i \leq p \) such that \( M_i \simeq_{rpo} M' \).

**Examples** Suppose \( x \land (y \lor z) \triangleright_{rpo} (x \land y) \lor (x \land z) \)

is to be proven. Choose \( \land \triangleright \lor \), which corresponds to case RPO2: \( x \land (y \lor z) \triangleright_{rpo} x \land y \) and \( x \land (y \lor z) \triangleright_{rpo} x \land z \) must be proven. Then \( x \land (y \lor z) \triangleright_{rpo} x \land y \) corresponds to case RPO1 (same head symbol): \( \{ x, y \lor z \} \triangleright_{rpo} \{ x, y \} \) must be proven. By the definition of the order over multisets, this follows from \( y \lor z \triangleright_{rpo} y \), which is true according to RPO3 (\( y = \) hence \( \not\triangleright \) a subterm of \( y \lor z \)).

In the following example, a decreasing sequence is proven using \( rpo \) even though the size increases:

\[
\neg (x \lor y) \triangleright_{rpo} \neg x \land \neg y.
\]

Suppose \( \neg \triangleright \land \). One must prove (case RPO2) that \( \neg (x \lor y) \triangleright_{rpo} \neg x \), and \( \neg y \), then (case RPO1) that \( \{ x \lor y \} \triangleright_{rpo} \{ x \} \) and \( \{ x \lor y \} \triangleright_{rpo} \{ y \} \). These results follow from \( x \lor y \triangleright_{rpo} x \) and \( x \lor y \triangleright_{rpo} y \), respectively, which are true by RPO3. \( \square \)

Certain rules are not suitable for an \( rpo \):
Example  For the associativity rule, \((xy)z \to x(yz)\), one would have to prove \(\{xy, z\} \rightharpoonup \{x, yz\}\), which does not work. However, \((xy)z \rightharpoonup x\) and \(yz\), and by replacing the multisets by \(n\)-tuples, \((xy, z) \succ (x, yz)\) for the lexical order, since \(xy \succ x\) by RPO3.

This leads to another case for defining the \(rpo\), which applies when a functional symbol has a lexical status:

\[
\text{RPO1LEX} \quad f = g \text{ (hence } p = q\text{), and } (M_1, \ldots, M_p) \rightharpoonup (M'_1, \ldots, M'_q), \text{ where } \rightharpoonup^* \text{ is the lexical extension of } \rightharpoonup \text{ to } n\text{-tuples of terms.}
\]

3.4.3 The Church–Rosser theorem

The Church–Rosser theorem, stating that \(\beta\)-reduction is confluent, is proven by introducing the concept of reduction relative to a set of ‘visible’ redexes.

Residues
Recall the choice of redex which is contracted in a term and the consequences of that choice on subsequent reductions. Note, first, certain phenomena:

- Let \(R\) be a redex, \(C\) the result of contracting \(R\), and \(M \equiv (\lambda x. xx)R\). \(M\) has two redexes, which are the subterms \(M'/e \equiv M\) and \(M'/2 \equiv R\). If \(R\) is contracted first, only one contraction of the obtained term, \((\lambda x. xx)C\), yields \(CC\). If \(M\) is contracted to \(RR\), two contractions of \(R\) are necessary to reach \(CC\).
- Let \(M \equiv (\lambda x y)N\). It is unnecessary to try to reduce \(N\), since the contraction of \(M\) immediately yields \(y\).
- Contracting can generate new redexes, e.g., \((\lambda x. xI)(\lambda y N) \to (\lambda y N)I\).

To study these phenomena, the concept of residue is introduced. Let \(M \rightharpoonup M'\), \(M/u \equiv (\lambda x P)Q\) and \(v \in \mathcal{O}_\text{red}(M)\). According to the relative position of \(u\) and \(v\) under the prefix order, we examine the effect of contracting \(M/u\) at redex \(M/v\), which yields a set of redex subterms of \(M'\), residues of \(M/v\):

- If \(u \perp v\), the contraction of \(M/u\) does not modify \(M/v\); the residue of \(M/v\) is \(M'/v\) (conservation).
- If \(u > v\), \(M/v\) is reduced, as \(M\), by contraction of \(M/u\); the residue of \(M/v\) is \(M'/v\) (reduction).
- If \(u = v\), \(M/v\) is contracted and does not have a residue (vanishing).
- If \(u < v\), then \(M/v\) is a redex of \(P \equiv M/u1\lambda\), or of \(Q \equiv M/u2\):
  - if \(M/v\) is a redex of \(P\), it is transformed in the same way that \(P\) is by substituting \(Q\) for \(x\); the residue of \(M/v\) is the subterm of \(M'\) corresponding to \((M/v)[x := Q]\) (substitution);
if $M/v$ is a redex of $Q$, it is grafted on to each occurrence of $x$ in $P$; these grafted subterms are the residues of $M/v$ in $M'$ (multiplication); the occurrences of the residues are independent in $O(M')$.

If this definition appears to be complicated, the practical determination of residues is simple. It suffices to mark the $\lambda$ of the redex $M/v$ and to 'trace' its reduction in $M$. The symbol $\hat{}$ is used for this marking. With an ML-like syntax, a marked redex $(\hat{\lambda}xP)Q$ could be written let `val x = Q in P end.

Example

$$(\lambda x. (\lambda z . zx)x)((\hat{\lambda}yP)Q) \rightarrow \lambda z . z((\hat{\lambda}yP)Q)((\hat{\lambda}yP)Q).$$

The definition of residues can be extended:

- The residues of a redex by a $\beta$-reduction are easily obtained by tracking the movements of the $\lambda$ of the redex.
- The residues of a set of redexes are obtained by marking the $\lambda$ of each of the considered redexes.

**Internal developments**

It is convenient to name reductions. One can write $u : M \rightarrow M'$ for the immediate $\beta$-reduction $M \rightarrow^u M'$, and $U : M \rightarrow^{*} M'$ for the parallel reduction of a set $U$ of independent occurrences of redexes. If $\rho : M \rightarrow M'$ and $\rho' : M' \rightarrow M''$ are two reductions, $\rho;\rho'$ denotes the composed reduction $M \rightarrow^{*} M' \rightarrow^{*} M''$. The composition has as neutral element $0 : M \rightarrow^{0} M$.

If $u$ and $v$ are two redex occurrences, write $u \setminus v$ for the set of residues of $u$ by contraction of $v$, and extend this notation to $U \setminus v$, $u \setminus \rho$ and $U \setminus \rho$, when $U$ is a set of redexes and $\rho$ a reduction, by

$$U \setminus v = \bigcup_{u \in U} u \setminus v,$$

$$U \setminus 0 = U,$$

$$U \setminus (\rho;\rho') = (U \setminus \rho) \setminus \rho'.$$

If $U \subset O_{red}(M)$, the pair $(M, U)$ denotes the term $M$, with the set $U$ of marked redexes.

**Definition 3.17** Let $M$ be a term and $U \subset O_{red}(M)$. A reduction of $M$ relative to $U$, or reduction of $(M, U)$, is any $\beta$-reduction of $M$, such that at each step the contracted redex is either an element of $U$ or a residue of an element of $U$ by the preceding steps.
By contracting only marked redexes, new redexes, invisible in \( M \), cannot be contracted. More precisely, a relative reduction is a reduction \( u_1; \ldots; u_n \), where for every \( i, 1 \leq i \leq n \), \( u_i \in U\setminus(u_1; \ldots; u_{i-1}) \). This reduction is written \( (M, U) \xrightarrow{\beta} (M', U') \), where \( U' \) is the set of residues of the redexes of \( U \) by this \( \beta \)-reduction of \( M \). If \( U' = \emptyset \), this reduction is called a development of \((M, U)\), or of \( M \), if \( U = O_{red}(M) \).

Which redex \( M/u, u \in U \), should be contracted first? For independent occurrences, the initial choice has no influence on subsequent choices.

Let \( M \) be a term and \( u_1 \) and \( u_2 \) be two redex occurrences in \( M \), with \( M/u_i \equiv (\lambda x_i P_i)Q_i, i = 1, 2 \). Consider the two immediate \( \beta \)-reductions \( u_1 \) and \( u_2 \) of \( M \), \( u_i: M \xrightarrow{u_i} M_i \). The sets of residues \( u_1 \setminus u_2 \) and \( u_2 \setminus u_1 \) are formed of independent occurrences which can therefore be contracted in any order. Form the reductions \( M_1 \xrightarrow{u_2 \setminus u_1} M' \) and \( M_2 \xrightarrow{u_1 \setminus u_2} M' \); then \( M' \equiv M'' \). This is a local confluence result, but it does not suffice to prove the Church–Rosser theorem.

In the case of an arbitrary set of redex occurrences, to avoid the multiplication of residues, one must begin with the innermost occurrences, i.e., maximal under the prefix order. Let \( u \in U \) be a maximal occurrence in \( U \). Then, by contracting \( M/u \), each \( v \in U \setminus \{u\} \) has a unique residue (the case \( v > u \), which could multiply \( v \), is excluded by the assumption about \( u \)). If \( U \) has \( n \) elements, the set of residues of \( U \) by contraction of \( u \) has \((n - 1)\) elements. By subsequently contracting a maximal redex in the residues of \( U \), a sequence of \( n \) immediate \( \beta \)-reductions \( M \rightarrow M_1 \rightarrow \ldots \rightarrow M_n \) is obtained, and the set of residues of \( U \) by this reduction is empty. Write \( M' \) for the term \( M_n \) when \( U \) is the set of marked redexes in \( M \), which is uniquely determined by \( M \) and \( U \), and can be computed by

\[
\begin{align*}
(x)^* & \equiv x, \\
(M_1M_2)^* & \equiv (M_1)^*(M_2)^* \text{ if } M_1M_2 \text{ is not a marked redex,} \\
(\lambda x M)^* & \equiv \lambda x M^*, \\
((\lambda x M_1)M_2)^* & \equiv (M_1)^*[x \equiv (M_2)^*].
\end{align*}
\]

Hence, relative reduction has the (weak) normalization property:

**Proposition 3.23** Every term has a finite development.

One should not think that \( M' \) is normal: this is due to the creation of new redexes which are not residues of redexes.

**The parallelogram property**

To continue with the proof of the Church–Rosser theorem, the parallelogram property is intermediate between confluence and local confluence: it is satisfied by a relation \( \rightarrow \) if for all \( x, x_1 \) and \( y, \) if \( x \rightarrow x_1 \) and \( x \xrightarrow{\ast} y \), then there exists \( z \) such that \( x_1 \xrightarrow{\ast} z \) and \( y \xrightarrow{\ast} z \). See Figure 3.2

**Lemma 3.24 (Parallelogram)** Confluence follows from the parallelogram property.
Proof. By induction on the length of the reduction $x \rightarrow^p x_p$. Let $x \rightarrow^* y$ and $x \rightarrow x_1 \rightarrow \ldots \rightarrow x_p$. If $p = 0$, let $z \equiv y$.

If $p > 0$, the parallelogram property implies the existence of $z_1$ such that $x_1 \rightarrow^* z_1$ and $y \rightarrow^* z_1$. By the inductive hypothesis, there exists $z$ such that $x_p \rightarrow^* z$ and $z_1 \rightarrow^* z$, from which, $x_p \rightarrow^* z$ and $y \rightarrow^* z$. \qed

Figure 3.2 Proof of the parallelogram property

All that is needed now is to prove the parallelogram property for $\beta$-reduction. The proof is constructive. Let $M \rightarrow M_1$ by contraction of the redex $u$, and $\rho : M \rightarrow N$. Compute the set $u \setminus \rho$ of the residues of $u$ under $\rho$, by tracing $u$ in $\rho$. It suffices to contract the elements of $u \setminus \rho$ in $N$ to obtain $P$ such that $M_1 \rightarrow P$. To be sure of the termination of this reduction, an internal development of $(N, u \setminus \rho)$ is used. Define $P \equiv N^\dagger$. Now $M_1 \rightarrow P$ must still be proven, using the following lemma:

Lemma 3.25. Let $M, N$ be marked terms. If $M \rightarrow N$ then $M^\dagger \rightarrow N^\dagger$.

Proof. It suffices to treat the case of an immediate $\beta$-reduction $M \rightarrow^\beta N$, by induction on the length of $M \rightarrow^* N$. The only case to be studied is that of redex contraction. Since it has already been shown that

\[
((\lambda x P)Q)^* \equiv (\lambda x P^*)Q^* \rightarrow^\beta P^*[x := Q'],
\]

\[
((\tilde{\lambda} x P)Q)^* \equiv P^*[x := Q'],
\]

it suffices to prove

\[
(P[x := Q])^* \equiv P^*[x := Q']
\]

by induction over $P$. Consider the case where

\[
P \equiv (\tilde{\lambda} y P_1)P_2.
\]

Then

\[
(P[x := Q])^* \equiv ((\tilde{\lambda} y P_1[x := Q])P_2[x := Q])^* \equiv (P_1[x := Q])^*[y := (P_2[x := Q])^*],
\]
from which, by the inductive hypothesis,

\[(P[x := Q])^* \equiv (P_1[x := Q^*])(y := (P_2[x := Q^*]));\]

by substitution,

\[\equiv P_1[y := P_2][x := Q^*] \equiv ((\lambda y P_1) P_2)^*[x := Q^*].\]

With this lemma, the Church–Rosser theorem is proven.

\[\square\]

3.4.4 Newman’s lemma

Local confluence is a property much easier to prove than confluence.

Lemma 3.26 (Newman) A noetherian relation is confluent if and only if it is locally confluent (see Figure 3.3).

Proof By induction over the well-founded relation $\rightarrow$, by showing that the set $P$ of $x$ where $\rightarrow$ is confluent is progressive. Let $x$ be such that $y \in P$ for every $y$ such that $x \rightarrow y$. One must prove that $x \in P$.

Let $x \rightarrow x'$ and $x \rightarrow x''$. Split the two relations into $x \rightarrow x_1' \rightarrow x_1$ and $x \rightarrow x_2' \rightarrow x_2$. By local confluence, there exists $x'$ such that $x_1' \rightarrow x'$ and $x_2' \rightarrow x'$. By the inductive hypothesis, $x_1' \in P$ and $x_2' \in P$, which means that there exist, respectively, $x_1''$ such that $x_1 \rightarrow x_1''$ and $x' \rightarrow x_1''$, and $x''$ such that $x_2'' \rightarrow x''$ and $x_2 \rightarrow x''$.

To prove that $x \in P$, it remains to check the trivial cases $x \equiv x_1$ or $x \equiv x_2$ where there is confluence, without using induction. \[\square\]
3.4.5 Normalization in λ-calculus

β-reduction has no normalization property. However, if the term formation rules are restricted, for example using types, or the use of β-reduction is restricted, by marking redexes, then there are normalization results.

Finiteness of developments
Let \((M, U)\) be a marked term, \(U\) being a set of redex occurrences in \(M\). It has already been shown that relative reduction satisfies the weak normalization property by successively contracting the innermost redexes. Here it will be shown that relative reduction also satisfies the strong normalization property, i.e., it is noetherian.

Define a function \(h\) over terms with values of \(N\) which is strictly decreasing when a marked redex is contracted. The principle is to associate an integer weight \(> 0\) with each variable occurrence, and to represent it by annotation: for example \(M \equiv (\lambda x. x^{11} x^{8})(y^{2}(\lambda x. x^{5}))\). Let \(h(M)\) be the sum of the weights: for the preceding term \(h(M) = 11 + 8 + 2 + 5 = 26\). The weight of each occurrence is carried by reduction, exactly like the variable it annotates. Hence \(M \rightarrow (y^{2}(\lambda x. x^{5}))(y^{2}(\lambda x. x^{5}))\). Clearly it should be the case that \(h(M) > h(M')\) if \((M, U) \rightarrow (M', U')\). To ensure this decreasing property, then, for every marked redex \((\lambda x P) Q\) and for every occurrence of \(x\) in \(P\), \(h(x) > h(Q)\): it is easy to prove that this condition suffices. A decreasing weighting is constructed by numbering the variable occurrence from right to left starting from 0 and by giving to the \(n\)-th the weight \(2^n\): the above term \(M\) would be annotated as \(M \equiv (\lambda x. x^8 x^4)(y^2(\lambda x. x^4))\).

For the reasoning to work, the decreasing nature of the weighting must be conserved by reduction. This property can be checked by distinguishing the cases of the relative positions of the redexes.

See Barendregt [8, section 11.2] for the details. Note that this result is constructive: if the initial term contains \(n\) variable occurrences, there are at most \(2^n\) reduction steps.

**Proposition 3.27**

1. Every reduction of \((M, U)\) can be extended to a development.
2. The set of terms obtained by any relative reduction of \(M\) is finite.

There is also a confluence result. The Church–Rosser theorem cannot be applied directly, for one would have to check that the β-reductions showing confluence contract only marked redexes. The proof of the local confluence of β-reduction only used residues of initial redexes: the reduction of marked redexes is therefore locally confluent. From Newman’s lemma, confluence follows from local confluence and strong normalization:

**Theorem 3.28** Relative reduction is noetherian and confluent.
Because of this theorem, development is as basic as immediate $\beta$-reduction. In the same manner that $N$ is uniquely determined by $M$ and by the contracted redex in the relation $M \rightarrow_\beta N$, the final term $N$ of a development $(M, U)$ is uniquely determined by $M$ and $U$, written $M \Rightarrow N$. The term $N$ cannot be computed, for example, by an internal reduction of $M$. If the let form of functional languages is interpreted as the marking of a redex, the preceding theorem allows this form to be considered as a 'syntactic extension' of the $\lambda$-calculus, definable as a redex, with the rule:

$$\text{let val } x = Q \text{ in } P \text{ end } \triangleright P[x := Q]$$

The application of this rule a finite number of times produces a term with no let.

**Normalization in typed $\lambda$-calculus**

The weak normalization theorem is proven for the typed $\lambda$-calculus, which guarantees the existence of a normal form.

A few useful definitions are given. The degree of a redex is

$$\text{deg}((\lambda x^\sigma M^\tau)N^\rho) = \text{ord}(\sigma \rightarrow \tau).$$

Note that $\text{ord}(\sigma) < \text{ord}(\sigma \rightarrow \tau)$ and that $\text{deg}(R^\tau) > \text{ord}(\tau)$. The total degree $\text{Deg}(M)$ of a term $M$ is the lub of the degrees of its redexes. A normal term has total degree 0.

**Lemma 3.29** Let $R$ be a redex of $M$, $M \rightarrow M'$ by contraction of $R$ and $R'$ be a redex of $M'$:

- if $R'$ is the residue $R_0 \setminus R$ of a redex $R_0$ of $M$, then $\text{deg}(R') = \text{deg}(R_0)$;
- otherwise, $R'$ is a new redex, and $\text{deg}(R') < \text{deg}(R)$.

**Proof** $M'$ is obtained by contracting the redex $R \equiv M/u \equiv (\lambda x^\sigma P^\tau)^{x \rightarrow v}Q^\sigma$ of $M$, to $C \equiv M'/u \equiv P[x := Q]$. Let $R' \equiv M'/v$ be a redex of $M'$. There are three cases depending on the position of $u$ with respect to $v$.

1. If $v \perp u$, $R'$ is the residue of a redex $R_0$ of $M$ which was conserved in $M'$.
2. If $v < u$, $R'$ is the residue of a redex $R_0$ of $M$ which was reduced by contraction of $R$.
3. If $v \geq u$, $R'$ is a redex of $C$. It is the residue of a redex $R_0$ of $M$ in $Q$, multiplied for each occurrence of $x$ in $P$, the residue of a redex $R_0$ of $M$ in $P$ substituted by $[x := Q]$, or a redex which was created during the contraction of $R$.

For the cases of conservation, reduction, multiplication or substitution of a redex, the degree of the residue does not change: $\text{deg}(R') = \text{deg}(R_0)$.

A new redex is created when $Q$ is an abstraction $\lambda y^\sigma Q'^{x\rightarrow}$, for each occurrence of $x^\sigma$ in $P^\tau$ which is the left-hand side of an application, written $x^\sigma P'^{x\rightarrow}$. The
type constraints force $\sigma' \rightarrow \tau' = \sigma$. The contraction of $R$ does the substitutions $(x^\sigma P^{\tau\sigma})[x := \lambda y Q']$ which create new redexes $(\lambda y^{\sigma'} Q''\tau')^{\sigma' \rightarrow \tau} P^{\tau\sigma}[x := \lambda y Q']$. Their degree is $\text{ord}(\sigma' \rightarrow \tau') = \text{ord}(\sigma) < \text{ord}(\sigma \rightarrow \tau) = \text{deg}(R)$. \qed

To conclude, a redex of $M'$ has a degree strictly bounded by that of $R$ if it is new or it has the degree of the redex of which it is a residue. This shows in particular that $\text{Deg}(M) \geq \text{Deg}(M')$. This lemma is used to prove the weak normalization theorem of the typed $\lambda$-calculus.

Proof. Let $h(M) = (d, n)$, where $d = \text{Deg}(M)$ and $n$ is the number of redexes of $M$ of degree $d$. If $M$ is normal, $h(M) = (0, 0)$. The term $M$ is reduced in order to decrease $h(M)$ in $N^2$ for the lexical order (i.e. $(d, n) > (d', n')$ if $d > d'$ or else $d = d'$ and $n > n'$). If $h(M) = (d, n)$, with $d > 0$, choose a redex $R$ of $M$ of degree $d$, itself not containing any redex of degree $d$, and let $M \rightarrow M'$ by contraction of $R$.

By choosing a redex $R$ of maximum degree $d$ of which every redex subterm is of degree strictly less than itself, the multiplication of redexes of degree $\geq d$ is avoided. By contracting $R$, a redex of degree $d$ disappears, and the created redexes are of strictly inferior degree. This proves that $h(M) > h(M')$. \qed

Exercises\footnote{Hints to the exercises labeled with a $\star$ can be found at the end of the book.}

1. Construct the graph of immediate $\beta$-reduction of the terms $WWW$ and $NNNN$, where $W \equiv \lambda xy.xyy$ and $N \equiv \lambda xy. yxx$.

2.\* Construct the graph of immediate $\beta$-reduction of $\Delta(WT)$, $W$ as above.

3.\* Let $M \equiv \lambda pq.p F(Ka)(q F(Kb))$, where $a$, $b$ are variables, and $F \equiv \lambda fg.g f$. What is the normal form of $Mpq$ if $p$ and $q$ are Church numerals. What is the length of the reduction?

4. Check that the $\lambda$-term zero represents an arithmetic if-then-else, and that pred represents the predecessor.

5. Show that $\lambda x. xx$ is not typable but that $\lambda x. xx$ is typable. Give a principal type.

6.\* Give a principal type for the combinators $S$, $B$ and $C$. Their definition in $\text{ML}$ is:

- $\text{fun } S \ f \ g \ x = \ f \ x \ (g \ x)$
- $\text{and } B \ f \ g \ x = \ f \ (g \ x)$
- $\text{and } C \ f \ x \ y = \ f \ y \ x$ ;
7. Find the covering relations defined by the usual order over \( \mathbb{N} \) in \( \mathbb{R} \), the prefix order in a free monoid \( A^* \) and the lexical order \( \leq^* \) over \( A^* \), for \( A \) totally ordered (see Chapter 1, exercise 2).

8. Let \( M, N \in T[X] \); show that \( M \) covers \( N \) if and only if \( M = \sigma N \), where \( \sigma \) is a substitution equating two distinct variables of \( N \) or associating a term \( f y_1 \ldots y_n \) (with new variables \( y_i \)) with one of the variables of \( N \).

9. Using the result of the preceding exercise, compute the number of classes in \( T[X] \) of terms covering a term \( N \), as a function of the number of variables of \( N \). Show that the number of classes of terms covered by \( N \) is finite.

10. Show that the order relation generated by the covering relation is the subsumption preorder.

11.* Show that \( M \land N \) is a lower bound of \( M \) and \( N \).

12. Is the system formed by the rule \( \neg x + y \rightarrow (\neg \neg x + y) + y \) noetherian?

13. Define a generic data type for binary trees (with leaves of type 'a) and an iterator over these trees. Use the iterator to compute the height, the number of leaves and the list of leaves of a tree.

14.* Here is a recursive definition of trees and forests:

\[
\text{datatype} \ 'a \text{ Tree} = \text{Union of} \ 'a \ast 'a \text{ Forest}
\]

\[
\text{and} \quad 'a \text{ Forest} = \text{Empty} \mid \text{Cons of} \ 'a \text{ Tree} \ast 'a \text{ Forest} ;
\]

Define two iterators Tree_iter and Forest_iter.

15. Let

\[
\text{type signature} = \text{symbol} \rightarrow \text{int} ;
\]

Define a function correct of type signature \( \ast \) term \( \rightarrow \) bool which indicates if a term is correctly formed over a signature. Implement the usual operations over terms and occurrences, computing \( O(M) \), \( M(u) \), \( M/u \) and \( M[u \leftarrow N] \).

16. Use the iterator It to define addition, multiplication and exponentiation over integers.

17. Define the iterator It over the integers using the recursor \( R \), and inversely.

18. What functions do hierarchy Zero and hierarchy (Succ Zero) compute? Prove that hierarchy omega is the diagonal \( \lambda n A(n, n) \) of the Ackermann function.
19. Derive the rewrite rules for the applicative terms $B$ and $C$, where $B \equiv S(KS)K$ and $C \equiv S(BBS)(KK)$.

20. Prove the optimization rules of combinatory logic using the extensionality rule.

21. Prove that the lexical order $\succ_{\times}$ over a product of well-founded orders is also well-founded.

22. Using an rpo order, prove the termination of the rewriting system computing Ackerman's function $A$, giving $A$ a lexical status.

23. Show that the redexes which contract to a redex are of one of three forms: $(\lambda x . xM)A$, $(\lambda x R)M$ or $IR$, where $M$ is an arbitrary term, $A$ is an abstraction and $R$ is a redex.

Bibliographic notes

The principal reference for the pure $\lambda$-calculus and combinatory logic is Barendregt's book [8]. The typed $\lambda$-calculi and their relation to logic are treated in [54, 64, 90]. Rewriting systems have been the subject of surveys by Huet and Oppen [71], and Dershowitz and Jouannaud [39].

Lindqvist's articles [92, 94] are the origins of functional programming. Another point of view was presented by Backus [7] as he received the Turing Award. Structural operational semantics, due to G.D. Plotkin [122], are developed under the name of natural semantics by G. Kahn and T. Despeyroux at the French INRIA [18]. The Damas–Milner type system, which is implemented in ML, is described in [110]. The variant presented here is found in [19].

Standard ML is described in [111, 159]; the first is a good introduction to functional programming. The book [1] by Abelson, Sussman and Sussman is highly recommended for programming in general. Scheme is described in [20], Ada, for example, in [9] and Common Lisp in [140]. CAML and its implementation are described in [29, 158]. Miranda is a trademark of Research Software Ltd.

The idea of compilation of functional languages by translation into applicative terms is due to Turner [152]. Supercombinators were introduced by Hughes [72]. The implementation of functional languages (interpretation and compilation) is treated in [43, 119, 120].

Kruskal's theorem, proved in [91], is a fundamental result in combinatorics. The given proof is Dershowitz's [38] (according to Nash–Williams), where he introduces simplification orders. Termination proof techniques have been particularly studied by Manna and Dershowitz. The REVE rewriting system is a joint project of MIT and the Centre for Research in Computer Science of Nancy, France.

Term $M$ of exercise 3 is due to B. Maurey, as given in [89].
4.1 First-order formulas

First-order languages were first introduced by Frege in 1879, who used a very complicated notation. Today, their use is widespread in mathematics.

**Example**  In the expression

\[(x > 0) \land \forall y \forall z (\neg(y * z = x) \lor y = x \lor z = x)\]

there are

- one constant (or 0-ary function) symbol 0;
- one binary (or 2-ary) function symbol *;
- two binary relation symbols = and >;
- variables \(x, y\) and \(z\); \(y\) and \(z\) are quantified, \(x\) is not;
- terms \(x, 0, y * z, y\) and \(z\);
- connectives \(\land, \neg\) and \(\lor\); and
- a quantifier \(\forall\).

The whole expression is a formula stating a property about \(x\). It will be shown later on that this formula can mean that \(x\) is prime in \(\mathbb{N}\).

4.1.1 Signature of a first-order language

The signature of a first-order language is the set \(\Sigma\) of function and relation symbols which are found in its formulas. In the above example, 0, *, = and > are members of the signature.

A *predicate calculus signature* is a pair of disjoint signatures \(\Sigma_f\) and \(\Sigma_r\), respectively called *functional* and *relational* signatures (see § 2.2.1). Write \(\Sigma = \Sigma_f \cup \Sigma_r\).
The elements of $\Sigma_f$ (respectively $\Sigma_r$) are the function (respectively relation) symbols.

**Examples**

- The signature of the language for arithmetic is $\{0, s, +, \times, <, =\}$, where $0$, $s$, $+$ and $\times$ are the functional symbols and $=\,$ and $<$ are the binary functional symbols.
- The signature of the language for groups is $\{e, \ast, ^{-1}, =\}$.

### 4.1.2 Formulas

Let a countably infinite set $X$ of variables be given. Then $T_{\Sigma_f}[X]$ is the set of finite terms over the signature $\Sigma_f$, also written $T_{\Sigma}[X]$ or simply $T[X]$. The construction of $T[X]$ gives a functional meaning to the elements of $\Sigma_f$. A relational meaning must still be given to the elements of $\Sigma_r$.

The **atomic formulas** (or **atoms**) are the $Rt_1 \ldots t_n$ with $R \in \Sigma_r$ of arity $n \geq 0$ and $t_1, \ldots, t_n \in T[X]$; they express a relation between terms. When they exist, standard notations for relations will be used. For example, $t_1 = t_2$ will be used instead of $\equiv t_1t_2$.

**Example** $x = x$ and $x > (y \ast 0)$ are atoms.  

The set of variables of an atom $Rt_1 \ldots t_n$ is:

$$\text{var}(Rt_1 \ldots t_n) = \bigcup_{i=1}^{n} \text{var}(t_i).$$

The set of atoms is written $\text{At}_\Sigma[X]$ or $\text{At}[X]$ and the set of closed atoms (no variable) is written $\text{At}_\Sigma$.

Formulas are constructed from atoms using connectives and quantifiers. The connectives can be chosen quite freely from the propositional calculus (§2.2.3). To begin, the **connectives** $\land$ (‘and’), $\lor$ (‘or’), $\Rightarrow$ (‘implies’) and $\bot$ (‘false’) will be used, as will the **universal** ($\forall$) and **existential** ($\exists$) **quantifiers**.

**Definition 4.1** The set $\text{L}_\Sigma[X]$ of formulas of the first-order predicate calculus over the signature $\Sigma$ is the smallest set $E$ of the free monoid $(\Sigma \cup X \cup \{\land, \lor, \Rightarrow, \bot, \forall, \exists\})^*$ such that:

1. $\text{At}_\Sigma[X] \subseteq E$;
2. $\bot \in E$, and if $\varphi, \psi \in E$ then $\varphi \land \psi, \varphi \lor \psi, \varphi \Rightarrow \psi \in E$;
3. if $x \in X, \varphi \in E$, then $\forall x \varphi$ and $\exists x \varphi \in E$. 

First-order logic

Example \( \forall x (x = x), (x > y \land y) \land (x = z) \) and \( \forall x (y > z) \) are formulas, but \( x = \bot \) and \( \forall (x \land x) > 0 \) are not. \( \square \)

The construction of \( L[X] \) only gave a functional meaning to connectives and quantifiers: \( \land, \lor, \Rightarrow \) are binary, \( \forall x, \exists x \) are unary, and \( \bot \) is 0-ary. Their logical meaning has yet to be given. The quantifier \( \forall \) could have been presented as a unary operator applied to 'functional-formulas', in which case one would write \( \forall (\lambda x. (x = x)) \) instead of \( \forall x (x = x) \).

Higher-order languages \( L[X] \) is a first-order language because variables only range over terms. One cannot quantify over functional or relational symbols, which are 'constants' of the language. In a second-order language, there are also an infinite number of relational variables of each arity, all quantifiable. The power of expression is much greater: for example, equality is defined by Leibniz's formula,

\[
(x = y) \equiv \forall P (P(x) \Rightarrow P(y)),
\]

where \( P \) is a unary relation symbol, and conjunction by

\[
\varphi \land \psi \equiv \forall P ((\varphi \Rightarrow (\psi \Rightarrow P)) \Rightarrow P).
\]

Higher-order logic is starting to be studied seriously in computer science.

Occurrences As for \( \lambda \)-terms, there are several levels of abstract syntax. The occurrences of a formula, its subformulas, and the logical symbol at an occurrence are defined as if the formula consisted of terms over the signature formed of connectives, \( \forall x \) and \( \exists x \). For example, for \( \varphi \equiv \forall x (A(x) \Rightarrow \exists y B(x, y)) \), \( O(\varphi) = \{\varepsilon, 1, 11, 12, 121\} \), \( \varphi/12 \equiv \exists y B(x, y) \) and \( \varphi(12) = \exists y \).

Sign of a subformula A formula occurrence is positive (respectively negative) if it is in the scope of an even (respectively odd) number of negations (explicitly, or implicitly, in the left-hand side of an implication).

More precisely, the sign of occurrences of \( \varphi \) is defined by a mapping \( \text{sg} : O(\varphi) \rightarrow \{+,-\} \), setting \( +' = - \) and \( -' = + \):

1. \( \text{sg}(\varepsilon) = +; \)
2. if \( \varphi/u = \chi \land \psi \) or \( \chi \lor \psi \), \( \text{sg}(u1) = \text{sg}(u2) = \text{sg}(u); \)
3. if \( \varphi/u = \chi \Rightarrow \psi \), \( \text{sg}(u1) = \text{sg}(u) \)' and \( \text{sg}(u2) = \text{sg}(u) \);
4. if \( \varphi/u = \forall x \psi \) or \( \exists x \psi \), \( \text{sg}(u1) = \text{sg}(u) \).

The sign of \( u \) is, by definition, that of the subformula \( \varphi/u \) and the logical symbol \( \varphi(u) \). Note that it is not a deductive definition over formulas: in terms of attribute grammars, the sign of a subformula is not synthesized starting from the signs of its components, but, rather, inherited from its context.
Free and bound variables The set \( \text{var}(\varphi) \) of free variables of a formula \( \varphi \) is defined by induction by:

\[
\begin{align*}
\text{var}(Rt_1 \ldots t_n) &= \bigcup_{i=1}^n \text{var}(t_i), \\
\text{var}(\bot) &= \emptyset, \\
\text{var}(\varphi \land \psi) &= \text{var}(\varphi \lor \psi) = \text{var}(\varphi \Rightarrow \psi) = \text{var}(\varphi) \cup \text{var}(\psi), \\
\text{var}(\forall x \varphi) &= \text{var}(\exists x \varphi) = \text{var}(\varphi) \setminus \{x\}.
\end{align*}
\]

Every occurrence of \( x \) in \( \forall x \varphi \) and \( \exists x \varphi \) is bound. A quantifier binds a variable, like the \( \lambda \) does in the \( \lambda \)-calculus. Recall that the name of a bound variable does not really have any meaning: \( \forall x(x > 0) \) and \( \forall y(y > 0) \) have the same meaning, but not \( \forall x(x > y) \) and \( \forall y(y > y) \). Since a written formula contains only a finite number of variables, defining \( X \) to be infinite allows renaming (implicit \( \alpha \)-conversion): replacing a bound variable in \( \varphi \) by a new variable not appearing in \( \varphi \) yields a new formula equal to \( \varphi \).

Examples

- In \( \forall x \exists y(x + y = z) \), \( y \) can be renamed to \( u \), but not to \( x \) or \( z \).
- \((x = 0) \lor \forall x(x \geq 0)\) is the same formula as \((x = 0) \lor \forall y(y \geq 0)\), since only the bound occurrence of \( x \) was renamed.

The elements of the set \( \text{var}(\varphi) \) defined above are called free variables of \( \varphi \). One can check they are not modified by renaming bound variables of \( \varphi \). Henceforth, any formula \( \varphi \) will be treated up to renaming of bound variables (implicit equivalence relation) and its bound variables will be written so they are not in \( \text{var}(\varphi) \), or not in a given set containing \( \text{var}(\varphi) \). The syntactic equality of two terms, just as for the equality of two formulas up to renaming of bound variables, is written \( \equiv \), when it is necessary, or simply =.

\( L[x_1, \ldots, x_n] \) is the set of formulas \( \varphi \in L[X] \) such that \( \text{var}(\varphi) \subseteq \{x_1, \ldots, x_n\} \), \( x_1, \ldots, x_n \in X \). One often writes \( \varphi(x_1, \ldots, x_n) \) to indicate that \( \varphi \in L[x_1, \ldots, x_n] \).

Closed formulas or propositions are the \( \varphi \) such that \( \text{var}(\varphi) = \emptyset \). Their set is written \( L_\Sigma \) instead of \( L[X] \). Formulas containing no quantifier are called open. Their set, written \( L^o[X] \), can be equated with Prop[At[X]], the set of propositional formulas formed over atoms. If \( \varphi \in L[x_1, \ldots, x_n] \) is an arbitrary formula, its universal and existential closures are, respectively,

\[
\begin{align*}
\forall(\varphi) \text{ or } \forall \bar{x} \varphi & \equiv \forall x_1 \ldots \forall x_n \varphi \in L_\Sigma \text{ and} \\
\exists(\varphi) \text{ or } \exists \bar{x} \varphi & \equiv \exists x_1 \ldots \exists x_n \varphi \in L_\Sigma.
\end{align*}
\]

Substitutions Grafting a term on to a term occurrence and substituting a term for a variable were presented in §2.2.4 and §2.2.6.
Example  $(x + yz)[y := x + y] \equiv x + (x + y)z$.

The definition of substitution in a formula is analogous to the \( \lambda \)-calculus case (§2.3.3), due to the bound variables. Here is a simple definition. Let \( \varphi \in L[X] \) and a term \( t \in T[X] \) such that the bound variables of \( \varphi \) are renamed outside of \( \text{var}(t) \) (always possible): \( \varphi[x := t] \) is the formula obtained from \( \varphi \) by replacing each occurrence of \( x \) (necessarily free) in \( \varphi \) by \( t \). If \( x \not\in \text{var}(\varphi) \), then \( \varphi[x := t] \) is \( \varphi \).

Example  Let \( \varphi \) be the formula \( \exists y (x > y) \). To replace \( x \in \text{var}(\varphi) \) by the term \( y+u \), one must first rename \( y \) in \( \varphi \) by rewriting \( \varphi \) as \( \exists z (x > z) \) and then \( \varphi[x := y + u] \) becomes \( \exists z (y + u > z) \). It is certainly not \( \exists y (y + u > y) \)!

4.2  Natural deduction

In this section, reasoning as it has been done since the ancient Greeks will be formalized. The standard meaning will also be given to the logical symbols. Doing this requires a new syntactic object, the derivation. Its inductive definition is made up of inference rules (§2.2.2).

There are several kinds of inference systems. The natural deduction systems that will be presented in detail allow the derivation of formulas modulo assumptions which are introduced and then discharged during a proof: the structure of derivations is that of \( \lambda \)-terms. In Hilbert systems, formulas are derived from axioms: derivations have a first-order term structure, where constants correspond to axioms. Finally, derivations of the sequent calculus also have a term structure, but the judgments that they contain are sequents, not formulas.

Gentzen proposed several inference systems in 1934, including the natural deduction systems NK (Natural Classic) for classical logic, NJ for intuitionistic logic and NM for minimal logic (see [144]). An ‘elementary’ formulation (in the sense of subatomic particles in physics) of the meaning of a logical formula is given using the meanings of its components: it is an operational interpretation of the connectives and quantifiers. For each of the logical symbols, there is an introduction (\( I \)) rule and an elimination (\( E \)) rule. The natural deduction system will be presented in four steps: the propositional minimal calculus, formed of rules for implication, conjunction and disjunction; the intuitionistic calculus, which adds an elimination rule for \( \perp \) and the quantifiers; and the classical system, with the law of the excluded middle.

4.2.1  The minimal system

There are two possible presentations for these rules. The first presentation derives formulas in a context of assumptions, i.e., judgments of the form \( \Gamma \vdash \psi \), where \( \Gamma = \{ \varphi_1, \ldots, \varphi_n \} \) is a finite set of formulas and \( \vdash \) is the deduction symbol. This
judgment is generally written \( \varphi_1, \ldots, \varphi_n \vdash \psi \) and states that \( \psi \) can be deduced from \( \varphi_1, \ldots, \varphi_n \). Write \( \Gamma, \varphi \) instead of \( \Gamma \cup \{ \varphi \} \) and \( \Gamma, \Delta \) instead of \( \Gamma \cup \Delta \).

The rules for implication are

\[
(I_{\Rightarrow}) : \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \quad \quad (E_{\Rightarrow}) : \frac{\Gamma \vdash \varphi \Rightarrow \psi}{\Gamma, \Delta \vdash \psi}.
\]

Consider the management of the assumptions, i.e., the expressions to the left of the \( \vdash \), in these rules. In general, the assumptions of the conclusion of a rule are the union of the assumptions of the premises, except when an assumption is ‘discharged’. In the case of \( (I_{\Rightarrow}) \), the formula \( \varphi \) is an assumption of the premise \( \Gamma, \varphi \vdash \psi \), but no longer occurs as an assumption in the conclusion \( \Gamma \vdash \varphi \Rightarrow \psi \) — except if \( \varphi \in \Gamma \): that occurrence of \( \varphi \) has been discharged by applying \( (E_{\Rightarrow}) \).

Applying \( (E_{\Rightarrow}) \) can discharge a bunch of assumptions, i.e., a possibly empty set of occurrences of \( \varphi \) appearing to the left of \( \vdash \) in the premise.

To avoid having to carry around the assumptions, the formulas themselves will be used as judgments, and only the assumptions to be discharged will be shown. The latter will be placed above the judgments for which they are assumptions, in square brackets or slashed. This second presentation for natural deduction rules will be used henceforth.

The rules for the minimal propositional system are given in Table 4.1, where \( \varphi \), \( \chi \) and \( \psi \) are formulas.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_{\varphi} )</td>
<td>\frac{\varphi}{\psi}</td>
<td>( \varphi \Rightarrow \psi )</td>
</tr>
<tr>
<td>( E_{\varphi} )</td>
<td>\frac{\varphi \Rightarrow \psi}{\varphi}</td>
<td>( \varphi \Rightarrow \psi )</td>
</tr>
<tr>
<td>( I_{\land} )</td>
<td>\frac{\varphi}{\varphi \land \psi}</td>
<td>( \varphi \land \psi )</td>
</tr>
<tr>
<td>( E_{\land} )</td>
<td>\frac{\varphi \land \psi}{\varphi}</td>
<td>( \varphi \land \psi )</td>
</tr>
<tr>
<td>( I_{\lor}^1 )</td>
<td>\frac{\varphi}{\varphi \lor \psi}</td>
<td>( \varphi \lor \psi )</td>
</tr>
<tr>
<td>( I_{\lor}^2 )</td>
<td>\frac{\psi}{\varphi \lor \psi}</td>
<td>( \varphi \lor \psi )</td>
</tr>
<tr>
<td>( E_{\lor} )</td>
<td>\frac{\varphi \lor \psi}{\chi}</td>
<td>( \varphi \lor \psi )</td>
</tr>
</tbody>
</table>

Table 4.1 The NM propositional system

- \( I_{\land}, E_{\land}, I_{\lor} \) and \( E_{\lor} \) (modus ponens) are clear.
- \( I_{\Rightarrow} \) and \( E_{\lor} \) contain discharged assumptions, in square brackets. For example, \( I_{\Rightarrow} \) (rule often used in mathematical reasoning) means: ‘if \( \psi \) can be deduced from \( \varphi \) then deduce \( \varphi \Rightarrow \psi \) without supposing \( \varphi \)’. Rule \( E_{\lor} \) reasons by case: if \( \varphi \lor \psi \) has been proven, then if \( \chi \) follows from \( \varphi \) and if \( \chi \) follows from \( \psi \), then deduce \( \chi \) without supposing \( \varphi \) or \( \psi \).

An inference rule acts as a rewriting rule, by pattern matching. The variables \( \varphi, \psi \) and \( \chi \) appearing in such rules are formula metasymbols.
4.2.2 Derivations

Inference rules are used to construct the set $D_\Sigma[X]$ of derivations inductively. These have the form of a binding tree whose nodes are labeled by a rule and by the judgment that is inferred. A binding links a node (the application of a rule) and one or more assumptions which were discharged by applying the rule. The root of a derivation tree is its conclusion. Each inference rule gives a rule for constructing derivations, which defines how assumptions are to be managed during an inference.

$\textbf{(DO)}$ For each formula $\varphi$, the tree with one node (labeled by) $\varphi$ is a derivation, of root $\varphi$ and assumption $\varphi$.

$\textbf{(DI}_\wedge\textbf{)}$ If $d$ is a derivation with root $\varphi$, $d'$ is a derivation with root $\psi$, then

\[
\frac{d \quad d'}{(I_\wedge)}: \frac{\varphi \quad \psi}{\varphi \wedge \psi}
\]

is a derivation of root $\varphi \wedge \psi$.

$\textbf{(DI}_\Rightarrow\textbf{)}$ If $d$ is a derivation with root $\psi$, then

\[
\frac{d}{(I_\Rightarrow): \frac{\psi}{\varphi \Rightarrow \psi}},
\]

with an arbitrary number ($\geq 0$) of links between the root and the leaves labeled by $\varphi$ not already linked, is a derivation of root $\varphi \Rightarrow \psi$.

The other rules are written in an analogous manner. The set of leaves of a derivation is the set of its assumptions and the root is its conclusion. The application of either of the rules $E_\wedge$ and $I_\Rightarrow$ discharges the slashed assumptions appearing in the rule: each occurrence of the assumptions slashed in the rule can also be (but is not forced to) slashed in the derivation. In a derivation, an assumption is either active, or discharged.

Example The two occurrences of assumption $\varphi \wedge \psi$ are discharged by $(I_\Rightarrow)$:

\[
(E_\wedge^2): \frac{[\varphi \wedge \psi]^1}{\psi}, \quad (E_\wedge^1): \frac{[\varphi \wedge \psi]^1}{\varphi}
\]

\[
(I_\wedge): \frac{\psi \wedge \varphi}{\varphi \wedge \varphi \Rightarrow \psi \wedge \varphi}
\]

$\Box$

To represent a derivation of $\varphi$ under assumptions $\Gamma$, it is not always convenient to use a tree format. A sequential presentation can also be used: this is a sequence
of formulas \( \varphi_1, \ldots, \varphi_n \), possibly in brackets, along with their justification, such that \( \varphi_n = \varphi \) and for all \( k \) (\( 1 \leq k \leq n \)), where either \( \varphi_k \in \Gamma \), either \( \varphi_k \) results in the application of a rule whose premises appear among the \( \varphi_i \), with \( l < k \), or \([\varphi_k]\) is a discharged assumption introduced during the proof.

Here is a presentation of the derivation of \((\varphi \land \psi) \Rightarrow (\psi \land \varphi)\) given above:

\[
\begin{align*}
1 & ~ [\varphi \land \psi] & \text{assumption} \\
2 & ~ \psi & \text{}\text{\( E^\land_2 \) on 1} \\
3 & ~ \varphi & \text{\( E^\land_1 \) on 1} \\
4 & ~ \psi \land \varphi & \text{I}_\land \text{on 2 and 3} \\
5 & ~ (\varphi \land \psi) \Rightarrow (\psi \land \varphi) & \text{I}_\Rightarrow \text{on 4, discharging 1}
\end{align*}
\]

This representation, often used, masks the real (abstract) nature of derivations.

Furthermore, a derivation dag is often used instead of a tree, which allows certain formulas, such as assumptions and intermediate conclusions, to be shared. In the preceding example, assumption 1 would be shared.

The difficult part is to create the derivations. In some cases, the structure of a formula \( \varphi \) to be proved guides the proof: the last rule to be applied is typically the introduction of the principal connector of \( \varphi \). If it is an implication, an assumption must be put ‘above’ the tree, and it should be taken advantage of. There are good reasons for applying eliminations to assumptions to decompose them, before applying introductions to construct the conclusion progressively. These remarks can be formalized.

### 4.2.3 The deduction relation

Building of derivations is usually thought of as an intermediate step for defining the deduction relation.

**Definition 4.2 Deduction** is the relation \( \vdash \) defined over \( \mathcal{P}(\text{L}[X]) \times \text{L}[X] \) by: \( \Gamma \vdash \varphi \) if there exists a derivation \( \varphi \) whose active assumptions belong to \( \Gamma \).

The relation is written \( \vdash_M \) for minimal logic, \( \vdash_I \) for intuitionistic logic and \( \vdash_C \) for classical logic. If \( \Gamma = \{ \varphi_1, \ldots, \varphi_n \} \) is finite, write \( \varphi_1, \ldots, \varphi_n \vdash \varphi \) instead of \( \Gamma \vdash \varphi \). If \( \Gamma = \emptyset \), write \( \vdash \varphi \), meaning that \( \varphi \) is a (deductive) tautology.

This relation satisfies the following property, which immediately follows from the rules \( I_\Rightarrow \) and \( E_\Rightarrow \):

\[ \varphi \vdash \psi \quad \text{if and only if} \quad \vdash \varphi \Rightarrow \psi; \]

more generally, also using \( I_\land \) and \( E_\land \),

\[ \varphi_1, \ldots, \varphi_n \vdash \psi \quad \text{if and only if} \quad \vdash \varphi_1 \land \ldots \land \varphi_n \Rightarrow \psi. \]

This property, called the ‘deduction theorem’ is not as easy to prove when other inference systems are used, such as Hilbert systems. A consequence of this property
is that it suffices to know how to prove tautologies: a tautology is no easier to prove than a ‘difficult’ theorem.

Note that much information is lost by only keeping (for \( \vdash \)) the leaves and the root of a derivation tree. The problem \( d \in D_B[X]? \) is decidable, while \( \Gamma \vdash \varphi? \) is generally not.

**Proven equivalence** A new binary connective, \( \Leftrightarrow \), short for *equivalence*, is introduced: write \( \varphi \Leftrightarrow \psi \) instead of \((\varphi \Rightarrow \psi) \land (\psi \Rightarrow \varphi)\). This connective is often used to state tautologies, as in

\[
\vdash (\varphi \land \psi) \Leftrightarrow (\psi \land \varphi),
\]

where the first part \( (\Rightarrow) \) was proven above.

It is easy to show that the binary relation between \( \varphi \) and \( \psi \), \( \vdash \varphi \Leftrightarrow \psi \), is an equivalence relation over the set of formulas: it is the *proven equivalence*, which should not be confused with \( \Leftrightarrow \), which is not a relation but a functional symbol just like the other connectives. This equivalence relation is compatible with the formation of formulas. For example, it can easily be shown that:

\[
\text{If } \begin{cases} 
\Gamma \vdash \varphi_1 \Leftrightarrow \psi_1 \\
\Gamma \vdash \varphi_2 \Leftrightarrow \psi_2 
\end{cases} \text{ then } \Gamma \vdash (\varphi_1 \land \varphi_2) \Leftrightarrow (\psi_1 \land \psi_2).
\]

There are five other analogous properties (for \( \lor, \Rightarrow, \neg, \forall x \) and \( \exists x \)) which allow the simplification of derivations by replacing, in each formula, a subformula \( \varphi \) by \( \psi \) where \( \vdash \varphi \Leftrightarrow \psi \). More generally, if \( \varphi \) is a tautology (i.e., \( \vdash \varphi \)), then \( \varphi \) can be introduced as an assumption and then immediately discharged, since by the definition of \( \vdash \), there exists a derivation of \( \varphi \) whose assumptions are discharged. Hence, a tautology can be used in a manner analogous to an axiom.

The connectives satisfy a number of ‘algebraic’ properties which can be written using proven equivalence. For example, conjunction is idempotent *modulo* equivalence: \( \varphi \land \varphi \) and \( \varphi \) are distinct formulas, but provably equivalent. In logic, only the class of a formula is important for the equivalence. This will not be true in logic *programming*, where, from an algorithmic point of view, it can take twice as long to prove \( \varphi \land \varphi \) than to prove \( \varphi \).

Conjunction and disjunction are ‘associative’, ‘commutative’, ‘idempotent’ and mutually ‘distributive’:

\[
\begin{align*}
\Gamma \vdash_M \varphi \land \psi & \Leftrightarrow \psi \land \varphi, \\
\Gamma \vdash_M (\varphi \land \chi) \land \psi & \Leftrightarrow \varphi \land (\chi \land \psi), \\
\Gamma \vdash_M (\varphi \land \chi) \lor \psi & \Leftrightarrow (\varphi \lor \psi) \land (\chi \lor \psi), \\
\Gamma \vdash_M \varphi \land \varphi & \Leftrightarrow \varphi,
\end{align*}
\]

and the relations which are yielded by exchanging \( \land \) and \( \lor \).
Example These properties allow the simplification of formulas:
\[ \vdash (\varphi \land (\psi \land \varphi)) \land (\chi \land \psi) \iff (\varphi \land \chi \land \psi) \]
using the commutativity, associativity and idempotence of \( \land \).

Derived rules The rules of natural deduction are elementary; they form a 'machine language' for logic. From a practical point of view, other (higher-level) rules can be introduced, allowing much shorter derivations. However, these rules cannot be added in an arbitrary manner, for the consistency of the system can be destroyed. It is best to add a rule derived from the initial inference system, i.e., whose conclusion allows a derivation under the assumptions formed by the premises in an inference system.

The use of a derived rule can then be replaced in situ by a derivation from the initial system.

Example The transitivity rule of implication
\[
(\Rightarrow \Rightarrow) : \frac{\varphi \Rightarrow \chi \quad \chi \Rightarrow \psi}{\varphi \Rightarrow \psi}
\]
is a rule derived from the derivation:
\[
(\Rightarrow \Rightarrow) : \frac{\varphi \Rightarrow \chi \quad [\varphi]_1}{\chi \Rightarrow \psi} \]
\[
(\Rightarrow \Rightarrow) : \frac{\varphi \Rightarrow \chi \quad [\varphi]_1}{\chi \Rightarrow \psi} \]

4.2.4 Abstract syntax of derivations

Derivations have a binding tree structure which resembles that of \( \lambda \)-calculus terms. However, there can be several bindings (0, 1, or more) attached to the same node. To represent a derivation by a \( \lambda \)-term, the set of discharged assumptions must be handled by the application of a single rule: a variable of the \( \lambda \)-calculus is associated with each active assumption and a (bound) variable to each bunch of discharged assumptions. The result is a typed \( \lambda \)-calculus whose types are formulas. Each derivation can be written as a term typed by a formula. This notation is based on the following correspondence in Table 4.2.

The typing rules for the typed \( \lambda \)-calculus have already been given. Judgment
\[
x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_M M : \tau
\]
should be reread as: under assumptions \( \tau_1, \ldots, \tau_n \), the term \( M \) is a derivation of \( \tau \), where each \( x_i \) is a unknown derivation of \( \tau_i \).
<table>
<thead>
<tr>
<th>Typed $\lambda$-calculus</th>
<th>Natural deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$-term type</td>
<td>derivation formula</td>
</tr>
<tr>
<td>type of term free variable</td>
<td>conclusion of derivation active assumption</td>
</tr>
<tr>
<td>bound variable abstraction</td>
<td>bunch of discharged assumptions $\Rightarrow$-introduction $\Rightarrow$-elimination</td>
</tr>
</tbody>
</table>

| Table 4.2 The Curry–Howard correspondence |

**Examples**

- The derivation with one node $\varphi$ is represented by the variable $x^\varphi$.
- The term $x^\varphi \Rightarrow y^\varphi$ represents the derivation

$$ (E \Rightarrow) : \frac{\varphi \Rightarrow \psi}{\psi} \varphi. $$

- The term $\lambda x^\varphi. x$ represents the derivation

$$ (I \Rightarrow) : \frac{[\varphi]}{\varphi \Rightarrow \varphi}. $$

- The term $\lambda f^\varphi \Rightarrow \varphi. x^\varphi. f(x^\varphi)$ represents the derivation

$$ (I^1 \Rightarrow) : \frac{[\varphi] \Rightarrow [\varphi]^2}{\varphi \Rightarrow \varphi} \varphi. $$

$$ (I^2 \Rightarrow) : \frac{\varphi \Rightarrow \varphi}{(\varphi \Rightarrow \varphi) \Rightarrow ((\varphi \Rightarrow \varphi) \Rightarrow (\varphi \Rightarrow \varphi)).} $$

- The term $\lambda x^\varphi((x \Rightarrow \psi) y^\varphi \Rightarrow x^\varphi. xz(yz)$ represents the derivation

$$ (E \Rightarrow) : \frac{[\varphi \Rightarrow (x \Rightarrow \psi)] \Rightarrow [\varphi] \Rightarrow [\varphi]^3}{\varphi \Rightarrow \psi \Rightarrow \psi} \varphi.$$  

$$ (E^2 \Rightarrow) : \frac{(\varphi \Rightarrow \psi) \Rightarrow (\varphi \Rightarrow \psi)}{(\varphi \Rightarrow (x \Rightarrow \psi)) \Rightarrow ((\varphi \Rightarrow x) \Rightarrow (\varphi \Rightarrow x))}. $$

### 4.2.5 Functional interpretation

Only a syntactic justification was given for the term/derivation correspondence. But it is not just a notation system: the correspondence is also semantic.
Natural deduction

121

The basic idea is that a formula’s meaning is the set of its proofs, where the proofs are the objects of an effective calculus. This idea, called BHK\(^2\) interpretation, forms the basis for the constructive approach to logic. By refusing the static notion of truthfulness \(\textit{being true, being false}\), intuitionists looked for a constructive, dynamic meaning for the concept of provable formula: what must one \textit{do} to prove a formula? The interest of such an interpretation for programming is manifest in the following double slogan:

\[
\begin{align*}
\text{proof} &= \text{term} = \text{program} \\
\text{proposition} &= \text{type} = \text{specification}
\end{align*}
\]

In its syntactic form, this idea leads to a typed \(\lambda\)-calculus whose types are the formulas and the terms of type \(\varphi\) are the derivations of \(\varphi\). The logical connectives become type constructors with very standard meanings, corresponding to well-known programming combinators. It is the Curry–Howard correspondence, used in combinatory logic by H. Curry, and made explicit in 1969 by W. Howard [66], which supports several formalisms which link computer science and logic.

Implication is the functional type constructor \(\rightarrow\), conjunction acts as the product \(\times\) and disjunction acts as the union (or sum) of types.

A proof of \(\varphi \Rightarrow \psi\) is a construction associating a proof of \(\psi\) with each proof of \(\varphi\). The introduction of \(\Rightarrow\) is the abstraction \(\lambda x^{\varphi} d\) of a derivation \(d\) of \(\psi\) over a derivation variable of \(\varphi\). The elimination of \(\Rightarrow\) is the application of a derivation of \(\varphi \Rightarrow \psi\) to a derivation of \(\varphi\) to obtain a derivation of \(\psi\). These are the basic operations of the typed \(\lambda\)-calculus. More precisely, the variable \(x^{\varphi}\) bound in \(\lambda x^{\varphi} d\) corresponds exactly to the bunch of assumptions discharged by applying \(I_\Rightarrow\). In fact, the result is exactly the above, proposed, simple notation system for derivations.

A derivation of \(\varphi \land \psi\) is a pair \((d, d')\) formed by a derivation \(d\) of \(\varphi\) and a derivation \(d'\) of \(\psi\), where the introduction of \(\land\) meaning pair formation; the eliminations are the projections \(\pi^1\) and \(\pi^2\). Conjunction then behaves as the product of types.

Disjunction corresponds to a \textit{union} (or sum) of types. A derivation of \(\varphi \lor \psi\) is either a derivation of \(\varphi\) or a derivation of \(\psi\). Each introduction is an injection, written \(\iota^1\) or \(\iota^2\), in a sum. The elimination of \(\lor\) is called case in programming. If \(d\) is a derivation of \(\varphi \lor \psi\), \(d'\) (respectively \(d''\)) a derivation of \(\chi\) under the assumption \(\varphi\) (respectively \(\psi\)), then case\((d, [x^{\varphi}].d', [y^{\psi}].d'')\) is a derivation of \(\chi\). Variables \(x^{\varphi}\) and \(y^{\psi}\) are, respectively, bound in \([x^{\varphi}].d'\) and \([y^{\psi}].d''\). This correspondence can be seen in Table 4.3.

\section*{Examples}

\begin{itemize}
\item \(\lambda x^{\varphi \land \psi}. (\pi^2(x), \pi^1(x))\) is a derivation of \(\varphi \land \psi \Rightarrow \psi \land \varphi\).
\item \(\lambda x^{\varphi \lor \varphi}. \text{case}(x, [x^{\varphi}].x, [x^{\varphi}].x)\) is a derivation of \((\varphi \lor \varphi) \Rightarrow \varphi\).
\end{itemize}

\(^{2}\) for Brouwer, Heyting and Kolmogorov.
<table>
<thead>
<tr>
<th>Typed λ-calculus</th>
<th>Natural deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>pair</td>
<td>conjunction</td>
</tr>
<tr>
<td>pair formation</td>
<td>∧-introduction</td>
</tr>
<tr>
<td>projections</td>
<td>∧-eliminations</td>
</tr>
<tr>
<td>union</td>
<td>disjunction</td>
</tr>
<tr>
<td>injections</td>
<td>∨-introductions</td>
</tr>
<tr>
<td>case</td>
<td>∨-elimination</td>
</tr>
</tbody>
</table>

Table 4.3 The Curry–Howard correspondence: the connectives

In the λ-calculus, a redex corresponds to the combined use of an introduction and an elimination. Redex contraction defines reductions over derivations, for which there are strong normalization and confluence results, due to D. Prawitz [125].

For example, the derivation

\[
\begin{align*}
\text{[\varphi]} \\
\vdots \\
\left( I_{\Rightarrow} \right): & \quad \frac{\psi}{\varphi \Rightarrow \psi} \\
\left( E_{\Rightarrow} \right): & \quad \frac{\varphi \Rightarrow \psi}{\psi} \\
\end{align*}
\]

is contracted to

\[
\begin{align*}
\vdots \\
\varphi \\
\psi
\end{align*}
\]

Example

\[
\begin{align*}
\left( E_{\Lambda} \right): & \quad \frac{\varphi \land \psi}{\varphi} \\
\left( I_{\Rightarrow} \right): & \quad \frac{\varphi \land \psi \Rightarrow \varphi}{\varphi} \\
\left( E_{\Rightarrow} \right): & \quad \frac{\varphi \land \psi \Rightarrow \varphi}{\varphi} \\
\end{align*}
\]

\[
\rightarrow \quad \left( E_{\Lambda} \right): \frac{\varphi \land \psi}{\varphi}.
\]

There are contraction rules for the other connectives ∧ and ∨, corresponding to the combination of introduction and elimination. The weak normalization theorem of the typed λ-calculus guarantees the existence of normal proofs in natural deduction (at least for the fragment corresponding to the type system used). Prawitz obtained not only a strong normalization theorem, but also a description of the form of normal derivations. Some remarkable properties of logic follow from this description. For example, the disjunctive property: if \( \Gamma \vdash \varphi \lor \psi \), then \( \Gamma \vdash \varphi \) or \( \Gamma \vdash \psi \). In fact, consider a derivation of \( \varphi \lor \psi \) in normal form. The last rule used is ∨-introduction and its premise is either \( \varphi \) or \( \psi \).

The Curry–Howard correspondence was presented in the framework of minimal propositional logic. It can easily be extended to intuitionistic logic, but does not hold for classical logic. Going to first-order logic consists of adding quantified
types, which is not difficult, but is not very interesting for programming languages. However, second-order quantification (over type variables) is a way of introducing generic types in languages, more general than those of ML: this is what is done in J.-Y. Girard’s system F (1972 [51]), which, although strongly normalizable, has great expressive power, allowing recursive definitions of functions and of data structures. For example, the type for integers is defined by \( \forall p. (p \rightarrow p) \rightarrow (p \rightarrow p) \), where \( p \) is a type variable.

These ideas have led to the design of completely new kinds of programming languages. These languages, completely integrated with logic, allow the simultaneous specification, synthesis and verification of programs. P. Martin-Löf’s Intuitionistic Type Theory [103] is both a programming language and a formal system for constructive mathematics, where constructive proofs are the algorithms. J.-L. Krivine’s Second Order Functional Arithmetic [90] and T. Coquand and G. Huet’s Calculus of Constructions [25] are two other recent formalisms based on these ideas.

4.2.6 The intuitionistic system

Deductions in minimal logic contain only implications, conjunctions and disjunctions. Negation will be introduced by giving meaning to the constant connective \( \bot \) representing ‘false’. There is no introduction rule for \( \bot \), only an elimination rule:

\[
(E_\bot) : \frac{\bot}{\psi}
\]

which means that from nothing, one can infer anything—in Latin, *ex falso quodlibet sequitur*. The Curry–Howard correspondence interprets \( \bot \) as the ‘empty type’, so \( \bot \) needs no proof. For each \( \psi \), rule \( E_\bot \) defines the unique ‘function’ from \( \bot \) to \( \psi \).

The inference system obtained, NJ, by adding this rule to those of minimal logic is the one of propositional intuitionistic logic. Write \( \vdash \) for the deduction relation defined by this system, as well as the one obtained by adding the rules for quantifiers.

**Negation** A new connective is introduced: *negation* (\( \neg \), sometimes written \( \sim \)). 
\( \neg \varphi \) is an abbreviation of \( \varphi \Rightarrow \bot \).

It is surprising that many tautologies containing negation can be derived in minimal logic (without using the rule \( E_\bot \)).

**Example** One of the contraposition laws is \((\varphi \Rightarrow \psi) \Rightarrow (\neg \psi \Rightarrow \neg \varphi)\). It is nothing but a particular case, with \( \omega = \bot \), of \((\varphi \Rightarrow \psi) \Rightarrow ((\psi \Rightarrow \omega) \Rightarrow (\varphi \Rightarrow \omega))\), which is a minimal tautology.

Other theorems must use \( E_\bot \), for example \((\neg \varphi \lor \psi) \Rightarrow (\varphi \Rightarrow \psi)\), whose converse will be proven in classical logic. To reason directly with negation, it will be more
convenient to use appropriate inference rules, which are written without \( \bot \):

\[
\begin{array}{c}
(I_{\rightarrow}) : \frac{\varphi}{\neg \psi} \quad (E_{\rightarrow}) : \frac{\varphi \Rightarrow \bot}{\neg \varphi} \\
(E_{\bot}) : \frac{\varphi \Rightarrow \bot}{\psi} \quad (I_{\bot}) : \frac{\bot}{\varphi}
\end{array}
\]

As negation is a particular case of implication, \( I_{\bot} \) and \( E_{\bot} \) are obtained from \( I_{\Rightarrow} \) and \( E_{\Rightarrow} \). The elimination of \( \Rightarrow \), applied to \( \varphi \) and \( \varphi \Rightarrow \bot \), allows the introduction of \( \bot \). Hence \( E_{\bot} \) follows from \( E_{\Rightarrow} \) and \( E_{\bot} \), and \( I_{\bot} \) is a particular case of \( I_{\Rightarrow} \):

\[
(I_{\Rightarrow}) : \frac{\bot}{\varphi \Rightarrow \bot} \quad (E_{\Rightarrow}) : \frac{\varphi \Rightarrow \bot}{\bot} \quad (E_{\bot}) : \frac{\varphi \Rightarrow \bot}{\psi} \quad (I_{\bot}) : \frac{\bot}{\varphi}.
\]

Note that only \( E_{\bot} \) uses \( E_{\bot} \), which means that \( I_{\bot} \) is a rule derivable in minimal logic.

### 4.2.7 Quantifiers and the logic variable

The NM and NJ systems are extended to the predicate calculus by giving rules to reason over quantifiers:

\[
\begin{array}{c}
(I_{\forall}) : \frac{\varphi(a)}{\forall x \varphi(x)} \\
(E_{\forall}) : \frac{\forall x \varphi}{\varphi[x := t]} \\
(I_{\exists}) : \frac{\varphi[x := t]}{\exists x \varphi} \\
(E_{\exists}) : \frac{\exists x \varphi}{\chi}
\end{array}
\]

where \( \varphi, \chi, \psi \in L[X], x, a \in X \) and \( t \in T[X] \). It is supposed that the bound variables have been renamed outside the name space of the free variables and of the substituted terms (\( a \) and \( t \)), and that \( x \) is free in \( \varphi(x) \).

By transforming these inference rules into derivation construction rules, certain restrictions must be placed on the presence of free variables in assumptions.

\( (DI_{\forall}) \) If \( d \) is a derivation of \( \varphi(a) \), and if all the assumptions of \( d \) in which \( a \) is free are discharged, then

\[
d (I_{\forall}) : \frac{\varphi(a)}{\forall x \varphi(x)}
\]

is a derivation of \( \forall x \varphi(x) \).

\( (DE_{\exists}) \) If \( d \) is a derivation of \( \exists x \varphi(x) \) and \( d' \) a derivation of \( \psi \), and if all the assumptions of \( d \) and those of \( d' \), other than \( \varphi(a) \), in which \( a \) is free, are
discharged, then
\[
\frac{d}{d'} \quad (E_\exists) : \frac{\exists x \varphi(x)}{\psi} \ \psi,
\]

with an arbitrary number of links between the root and the leaves of \(d'\) labeled by \(\varphi(a)\) and not already linked, is a derivation of \(\psi\).

The rules \((DI_\exists)\) and \((DE_\varphi)\) are elementary: no assumption discharge, no restriction. If the restrictions over \((DI_\varphi)\) and \((DE_\exists)\) were not applied, the following unexpected derivations would be obtained:
\[
(E_\exists) : \frac{\exists x(x = 0)}{\psi} \quad (E_{\neg}) : \frac{\neg(a = 0)}{\psi}
\]

(for arbitrary \(\psi\), and
\[
\forall x(x = 0).
\]

Here, on the other hand, is a good derivation. Note that at the moment when the elimination of the \(\exists\) is applied, \(x\) is bound in the active assumptions \(\forall x(\neg \varphi)\) and \(\exists x \varphi\), other than \(\varphi\).

\[
(E_{12}^\exists) : \frac{\exists x \varphi^2 \quad [\varphi]^1 \quad (E_{\neg}) : \frac{\forall x \neg \varphi}{\neg \varphi}}{\neg(\exists x \varphi)}
\]

By discharging the last hypothesis by \(I \Rightarrow\), \(\vdash_I (\forall x \neg \varphi) \Rightarrow \neg (\exists x \varphi)\) would be obtained.

Quantifiers also satisfy a form of commutativity and distributivity modulo proven equivalence:

\[
\vdash_M \forall x \forall y \varphi \iff \forall y \forall x \varphi
\]
\[
\vdash_M \forall x \varphi \iff \varphi, \quad \text{if } x \notin \text{var}(\varphi)
\]
\[
\vdash_M \forall x(\varphi \land \psi) \iff \forall x \varphi \land \forall x \psi
\]
\[
\vdash_M \forall x(\varphi(x) \lor \psi) \iff \forall x \varphi(x) \lor \psi, \quad \text{if } x \notin \text{var}(\psi)
\]

Replacing \(\forall\), \(\land\) and \(\lor\) by \(\exists\), \(\lor\) and \(\land\) would give more proven equivalences.

The logic variable In any comparison between functional and logic programming languages, attention must be paid to the nature of the variables.

A functional variable is unidirectional: passing an argument to a function always occurs in the same direction, and the process is formalized by \(\beta\)-reduction. A
logic variable is bidirectional: it can also receive a value from the invocation of a predicate in which it appears.

This phenomenon must necessarily be explained by the inference rules for quantifiers: they are the only ones with variables. Consider the combination of an elimination and of an introduction.

Let \( \varphi \in L_\Sigma[x_1, \ldots x_m] \) and \( t_1, \ldots, t_m \in T_\Sigma[y_1, \ldots, y_n] \):

\[
\begin{align*}
(E_\varphi) & : \quad \forall \vec{x} \varphi \\
(I_\varphi) & : \quad \forall \vec{y}(\varphi[x_1 := t_1, \ldots, x_n := t_m])
\end{align*}
\]

where, first, the \( m \) eliminations of \( \forall x_i, 1 \leq i \leq m \), are applied, then the \( n \) introductions of \( \forall y_i, 1 \leq i \leq n \). The latter are allowed since the \( y_i \) are not free in the assumption, which is closed.

A derived rule is thus obtained, \( \varphi \) being an arbitrary formula and \( \sigma \) a substitution:

\[
(\forall \forall) : \quad \forall(\forall(\varphi))
\]

Note that the inference of \( \sigma \varphi \) from \( \varphi \) is not normally allowed. Quantifiers here serve as ‘modal’ operators in the context from which one can apply a substitution to a formula, yielding an instance.

In an instantiation, the (bound) variables of \( \varphi \) become more defined under the subsumption preorder over terms. Such an inference gives them a value, taking the environment from \( \left[ x \right] \) to \( \left[ x \sigma(x) \right] \): the formula is therefore weakened as an environment is created. The typical problem in proofs is: ‘prove \( \forall(\varphi) \)’. But one can be interested by a more general problem: ‘find a \( \sigma \) such that \( \forall(\sigma \varphi) \) is derivable’. In the case where \( \forall(\sigma) \) is derivable, all substitutions \( \sigma \) are solutions, according to the rule \( (\forall \forall) \), but the most interesting, because it is the most general, is the identity substitution. In the general case where the identity is not a solution, the most general substitutions, i.e., minimal under a particular preorder, are sought.

For example, if the ‘goal’ is the formula \( \forall xy R(x, y) \), solutions such as substitutions \( [1] \) and \( [2] \) could be obtained, with derivations of \( \forall y R(0, y) \) and \( \forall x R(x, 1) \). This illustrates the behavior of the ‘logic variable’: it can be instantiated during a derivation, and several solutions are possible.

\[4.2.8 \quad \text{The classical system}\]

Adding another inference method, the excluded middle, to intuitionistic logic yields classical logic. The excluded middle, which states that a property is either true or false, can be defined in several ways: by a nullary inference rule explicitly stating the principle, by a unary inference rule for proof by contradiction or by a unary
inference rule stating the involutivity of negation.

\[(EM) : \frac{[\neg \varphi]}{\varphi \lor \neg \varphi}, \quad (C) : \frac{\bot}{\varphi}, \quad (-\neg) : \frac{\neg \neg \varphi}{\varphi}.\]

These rules are equivalent, i.e., derivable from each other. For example, the ex-
cluded middle is derivable in NJ extended by the rule \((C)\) of reasoning with con-
tradiction.

\[\frac{[\neg (\varphi \lor \neg \varphi)]^2}{\bot} \quad (I_{\lor}) : \frac{[\neg \varphi]^1}{\varphi \lor \neg \varphi} \quad \frac{[\neg (\varphi \lor \neg \varphi)]^2}{\bot} \quad (I_{\lor}) : \frac{\varphi}{\varphi \lor \neg \varphi} \quad \frac{\bot}{\varphi \lor \neg \varphi}.\]

The NK system of natural deduction in classical logic is given by Table 4.4. Write
\(\vdash_{C}\) or simply \(\vdash\) for the deduction relation defined by this system.

Proofs using the excluded middle are often more difficult than proofs in minimal
logic. The role of negation will probably be clearer in the sequent calculus. Note,
however, that there are formulas derivable in NK and not in NJ which do not
contain the negation symbol. One might even be tempted to try to prove them
in minimal logic. However, their derivation in NK necessarily uses rule \((C)\), and
so introduces negation (see exercise 16). In the sequent calculus, it will not be
necessary to introduce negation—but sequents must be derived, not formulas.

The connectives, like the quantifiers, are no longer independent. Morgan’s ax-
ioms state the relationship between \(\lor\), \(\land\) and \(\neg\):

\[\vdash_{C} \varphi \lor \psi \iff \neg (\neg \varphi \lor \neg \psi), \quad \vdash_{C} \varphi \land \psi \iff \neg (\neg \varphi \land \neg \psi).\]

Implication is definable from \(\lor\) and \(\neg\):

\[\vdash_{C} (\varphi \Rightarrow \psi) \iff (\neg \varphi \lor \psi).\]

The two quantifiers are exchanged by negation:

\[\vdash_{C} \forall x \varphi \iff \neg \exists x (\neg \varphi).\]

*Proposition 4.1* For any formula \(\varphi\), there exists \(\varphi'\) formed of logical symbols \(\lor, \neg\)
and \(\exists\) such that \(\vdash_{C} \varphi \iff \varphi'\).
(I\Lambda): \frac{\phi}{\phi \land \psi}

(E_1): \frac{\phi \land \psi}{\phi}

(E_2): \frac{\phi \land \psi}{\psi}

(I\Psi): \frac{\psi}{\phi \lor \psi}

(I\exists): \frac{\psi}{\phi \lor \psi}

(E\lor): \frac{\phi \lor \psi}{\phi \lor \psi}

([\varphi]) \quad ([\psi])

(E\Rightarrow): \frac{\psi \Rightarrow \phi}{\phi}

(E\forall): \frac{\forall x \varphi}{\varphi[x := t]}

([\varphi(a)])

(E\exists): \frac{\exists x \varphi}{\phi}

([\neg \varphi]) \quad ([\neg \varphi])

(E\neg): \frac{\neg \varphi}{\psi \lor \neg \psi}

(C): \frac{\psi}{\varphi \lor \neg \psi}

Table 4.4 The NK system

**Proof**  By induction over formulas.

If \( \varphi \) is atomic, let \( \varphi' = \varphi \).

If \( \varphi = \varphi_1 \lor \varphi_2 \), by the inductive hypothesis \( \varphi_i \iff \varphi_i' \), \( i = 1, 2 \), let \( \varphi' = \varphi_1' \lor \varphi_2' \).

If \( \varphi = \varphi_1 \land \varphi_2 \), by the inductive hypothesis \( \varphi_i \iff \varphi_i' \); since \( \vdash \varphi \iff \neg (\neg \varphi_1 \lor \neg \varphi_2) \), let \( \varphi' = \neg (\neg \varphi_1' \lor \neg \varphi_2') \).

If \( \varphi = \varphi_1 \Rightarrow \varphi_2 \), let \( \varphi' = \neg \varphi_1' \lor \varphi_2' \).

Finally, if \( \varphi = \exists x \varphi_1 \), let \( \varphi' = \exists x \varphi_1' \), and if \( \varphi = \forall x \varphi_1 \), let \( \varphi' = \neg \exists x (\neg \varphi_1') \).

This reduction to three logical symbols will be useful in proofs by induction over formulas, since it is easier to treat three cases than six. Other symbol ‘bases’ can be useful, such as the dual basis of the former, consisting of \( \land \), \( \neg \) and \( \lor \).

**Classical logic, intuitionistic logic and computer science**  Adding rule (C) to NJ allows the proof of many more tautologies. However, classical logic is not more ‘powerful’ than intuitionistic logic: it is simply less strict, having lost the constructive character visible through the Curry–Howard correspondence. Gödel discovered a translation explaining the relations between NK and NJ: if \( \varphi \) is a formula, an
intuitionistic formula $\varphi^g$ classically equivalent to $\varphi$ can be constructed, intuitionistically weaker yet derivable in NJ if $\varphi$ is in NK:

$$
\begin{align*}
\bot^g &= \bot, \\
\varphi^g &= \neg\neg\varphi \quad \text{if } \varphi \text{ is atomic, and } \not= \bot, \\
(\varphi \land \psi)^g &= \varphi^g \land \psi^g, \\
(\varphi \lor \psi)^g &= \neg(\neg\varphi^g \land \neg\psi^g), \\
(\varphi \rightarrow \psi)^g &= \varphi^g \rightarrow \psi^g, \\
(\forall x \varphi)^g &= \forall x \varphi^g, \\
(\exists x \varphi)^g &= \neg\forall x \neg\varphi^g.
\end{align*}
$$

**Example**

$$
\varphi \equiv \forall x R(x) \lor \exists x \neg R(x), \\
\varphi^g \equiv \neg(\neg\forall x \neg R(x) \land \neg\exists x \neg\neg R(x)).
$$

Note that $\varphi^g$ has no disjunctions of existential quantifiers, and that every atom appears negated at least once. This transformation allows the elimination of the two logical symbols whose intuitionistic and classical behaviors are very different.

**Lemma 4.2** If $\varphi$ is a formula constructed using $\Rightarrow$, $\land$ and $\forall$ from negations of atoms, then $\Gamma \vdash \varphi \iff \neg\neg\varphi$.

**Proof** By induction on the formation of this class of formulas. There are four cases: $\neg\neg A$, $\psi \Rightarrow \chi$, $\psi \land \chi$ and $\forall x \psi$.

Consider the case $\varphi \equiv \psi \land \chi$. Then $\Gamma \vdash \neg\neg(\neg\neg\psi \land \neg\neg\chi)$. By the inductive hypothesis, $\Gamma \vdash \neg\neg\psi \Rightarrow \psi$ and $\Gamma \vdash \neg\neg\chi \Rightarrow \chi$, hence $\Gamma \vdash \neg\neg(\neg\neg\psi \land \neg\neg\chi) \Rightarrow (\psi \land \chi)$.

The other cases use the following minimal tautologies:

$$
\begin{align*}
\Gamma \vdash \neg\neg\neg\psi &\iff \neg\psi \\
\Gamma \vdash \neg\neg(\psi \Rightarrow \chi) &\Rightarrow (\psi \Rightarrow \neg\neg\chi) \\
\Gamma \vdash \neg\neg\forall x \psi &\Rightarrow \forall x \neg\neg\psi
\end{align*}
$$

The proof of each of these tautologies is left as an exercise.

**Theorem 4.3**

1. $\Gamma \vdash \varphi \iff \varphi^g$;
2. $\Gamma \vdash \varphi$ if and only if $\Gamma^g \vdash \varphi^g$. 
Proof. It is easy to prove (1). For (2), suppose \( \Gamma^g \vdash_M \varphi^g \). Then \( \vdash_M (\Gamma \Rightarrow \varphi)^g \), and so \( \vdash_C (\Gamma \Rightarrow \varphi)^g \). By (1), \( \vdash_C (\Gamma \Rightarrow \varphi) \), i.e., \( \Gamma \vdash_C \varphi \).

The converse is a proof by induction over derivations of \( \varphi \) in NK. The cases where \( \varphi \) is derived by introduction or elimination of \( \Rightarrow \), \( \land \) or \( \forall \) are immediate.

Consider the contradiction rule case. Let \( d \) be a derivation of \( \bot \) from which \( \varphi \) is derived by discharging the assumption \( \neg \varphi \) of \( d \). By the inductive hypothesis, there exists a derivation \( d^g \) of \( \bot \) under the assumption \( \neg \varphi^g \). Furthermore, \( \neg \neg \varphi^g \Rightarrow \varphi^g \) is a minimal tautology (the previous lemma). So, the minimal derivation is

\[
\begin{align*}
\left( \text{I}_{\Rightarrow} \right) : & \quad (\Gamma^g) \\
\left( \text{E}_{\Rightarrow} \right) : & \quad \frac{\bot}{\neg \neg \varphi^g} \quad \frac{\neg \neg \varphi^g}{\varphi^g} \\
\vdash d^g & \quad \vdash \neg \varphi^g \Rightarrow \varphi^g \quad \vdash [\neg \varphi^g]^1
\end{align*}
\]

The cases of the logical symbols \( \lor \) and \( \exists \) do not have to be considered, since in classical logic, they can be defined in terms of \( \Rightarrow \), \( \land \) and \( \forall \).

It is a good idea to be able to recognize those formulas derivable in intuitionistic logic and those which are only classical. The functional interpretation given by minimal logic is useful. Consider the intuitionistic deduction

\[(\varphi \Rightarrow \psi) \lor (\varphi \Rightarrow \chi) \vdash_I \varphi \Rightarrow (\psi \lor \chi).\]

To the left of the \( \vdash_I \), there is either a function which generates a proof of \( \psi \) from a proof of \( \varphi \), or a function which generates a proof of \( \chi \) from a proof of \( \varphi \). One can then create a function which generates a proof of \( \psi \) or a proof of \( \chi \) from a proof of \( \varphi \). The inverse deduction, true in NK, is false in NJ, and even incomprehensible from a constructive point of view: how, starting from a function \( f \) of \( A \) into \( B \cup C \), does one define a function \( g : A \rightarrow B \) or a function \( h : A \rightarrow C \)? One would have to choose an element \( b \in B \) to define \( g(a) = b \) when \( f(a) \notin B \), a choice which is clearly not constructive.

From the classical point of view, limiting the number of connectives does not reduce the expressivity. However, logic is not just used to state properties. The connectives all have an operational aspect, clearly visible in NJ, which is lost in Gödel’s translation. From the constructive point of view of computer science, the meaning of \( \exists x A(x) \), the existence of an element, is stronger than \( \neg(\forall x \neg A(x)) \), the non-existence of a derivation.

Intuitionistic logic is today considered to be the logic corresponding to sequential computation. It is not coincidental that logic programming (pure Prolog), normally defined using classical logic, fits perfectly in the framework of intuitionistic logic.

### 4.2.9 The sequent calculus

The NK system is not as well adapted to classical logic as the NJ system is to intuitionistic logic. The \( I/E \) symmetry is not respected by the contradiction proof
rule. Furthermore, the classical algebraic properties of connectives and quantifiers are not evident: the $\land/\lor$ and $\forall/\exists$ dualities and the involutivity of $\neg$. The excluded middle gives disjunction an opaque character: $\Gamma, \varphi \lor \psi$ no longer means that $\varphi$ is provable or that $\psi$ is provable.

For classical logic, the most appropriate system is Gentzen’s sequent calculus LK, which reestablishes the symmetry broken by rule (C), but at the cost of a certain redundancy of proofs and by the use of ‘undefined’ judgments. Furthermore, the intuitionistic version LJ is naturally adapted to logic programming. These two facts justify a short introduction to the sequent calculus.

Judgments are introduced as sequents, which are lists of symbols of the form

$$\varphi_1, \ldots, \varphi_n : \psi_1, \ldots, \psi_m,$$

stating that from $\varphi_1$ and $\varphi_2$ and ... and $\varphi_n$, one can deduce $\psi_1$ or $\psi_2$ or ... or $\psi_m$. As particular cases,

$$:\psi \quad \text{and} \quad \varphi :,$$

respectively, state that $\psi$ can be proven and that $\varphi$ is refutable.

The intuitionistic case arises when $m$ is 0 or 1: this is the LJ system. Until now, one changed logic by adding or removing rules. Here, the form of judgments is restricted, and the rules are retained. The behavior of Prolog fits naturally in this system: Horn clauses are intuitionistic sequents over atomic formulas.

The rules of the sequent calculus respect a left/right symmetry. There are:

- structural rules,
- logical rules,
- the identity axiom, and
- the cut rule.

Since a sequent is a list—not a set—of symbols, their order and repetitions must be taken into account: this role is fulfilled by the structural rules of weakening, contraction and permutation. The permutation rule is generally applied implicitly (see Table 4.5).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(L_A)$</td>
<td>$\Gamma, \varphi : \Delta \quad \Gamma : \Delta$</td>
</tr>
<tr>
<td>$(R_A)$</td>
<td>$\Gamma : \Delta, \varphi$</td>
</tr>
<tr>
<td>$(L_C)$</td>
<td>$\Gamma, \varphi, \varphi : \Delta \quad \Gamma, \varphi : \Delta$</td>
</tr>
<tr>
<td>$(R_C)$</td>
<td>$\Gamma : \varphi, \varphi, \Delta$</td>
</tr>
<tr>
<td>$(L_F)$</td>
<td>$\Gamma, \varphi, \psi, \Gamma' : \Delta \quad \Gamma, \psi, \varphi, \Gamma' : \Delta$</td>
</tr>
<tr>
<td>$(R_F)$</td>
<td>$\Gamma : \Delta, \varphi, \psi, \Delta'$</td>
</tr>
</tbody>
</table>

Table 4.5 The LK system: structural rules
For each connective or quantifier, there is a left- and a right-introduction rule: these are the logical rules. Negation (¬) is used directly, instead of \( \bot \). There is a restriction in \( L_\exists \) and \( R_\forall \): \( x \) must not occur free in \( \Gamma, \Delta \) (see Table 4.6).

\[
\begin{align*}
(L_\land^\land) & : \quad \Gamma, \varphi \Delta \quad \Gamma, \psi \Delta \quad \frac{\Gamma, \varphi \land \psi \Delta}{\Gamma, \varphi \land \psi \Delta} \\
(L_\lor^\land) & : \quad \Gamma, \varphi \Delta \quad \Gamma', \psi \Delta' \quad \frac{\Gamma, \varphi \lor \psi \Delta, \Delta'}{\Gamma, \varphi \lor \psi \Delta, \Delta'} \\
(L_\rightarrow^\land) & : \quad \Gamma, \varphi \Delta \quad \Gamma', \psi \Delta' \quad \frac{\Gamma, \varphi \Rightarrow \psi \Delta, \Delta'}{\Gamma, \varphi \Rightarrow \psi \Delta, \Delta'} \\
(L_\neg^\land) & : \quad \Gamma, \varphi \Delta \quad \frac{\Gamma, \neg \varphi \Delta}{\Gamma, \neg \varphi \Delta} \\
(L_\forall^\land) & : \quad \Gamma, \varphi[x := t] \Delta \quad \frac{\Gamma, \forall x \varphi \Delta}{\Gamma, \forall x \varphi \Delta} \\
(L_\exists^\land) & : \quad \Gamma, \varphi \Delta \quad \frac{\Gamma, \exists x \varphi \Delta}{\Gamma, \exists x \varphi \Delta} \\
(R_\land^\land) & : \quad \Gamma, \varphi \Delta \quad \Gamma', \psi \Delta' \quad \frac{\Gamma, \varphi \land \psi \Delta, \Delta'}{\Gamma, \varphi \land \psi \Delta, \Delta'} \\
(R_\lor^\land) & : \quad \Gamma, \varphi \Delta \quad \frac{\Gamma, \varphi \lor \psi \Delta}{\Gamma, \varphi \lor \psi \Delta} \\
(R_\rightarrow^\land) & : \quad \Gamma, \varphi \Delta \quad \frac{\Gamma, \varphi \Rightarrow \psi \Delta}{\Gamma, \varphi \Rightarrow \psi \Delta} \\
(R_\neg^\land) & : \quad \Gamma, \varphi \Delta \quad \frac{\Gamma, \neg \varphi \Delta}{\Gamma, \neg \varphi \Delta} \\
(R_\forall^\land) & : \quad \Gamma, \varphi \Delta \quad \frac{\Gamma, \forall x \varphi \Delta}{\Gamma, \forall x \varphi \Delta} \\
(R_\exists^\land) & : \quad \Gamma, \varphi[x := t] \Delta \quad \frac{\Gamma, \exists x \varphi \Delta}{\Gamma, \exists x \varphi \Delta}
\end{align*}
\]

Table 4.6 The LK system: logical rules

A zero arity rule (axiom) is necessary: the identity

\[ \varphi : \varphi \]

corresponds to the derivation with node \( \varphi \) in natural deduction. The cut rule is presented below. The system described up to now is the cut-free LK system.

Derivations in LK have a tree structure, whose leaves are applications of the identity rule and whose root is the conclusion. There is no reasoning 'modulo assumptions', as in natural deduction.

Example A derivation of the sequent : \(((\varphi \Rightarrow \psi) \Rightarrow \varphi) \Rightarrow \varphi\), which is Peirce's axiom, a classical tautology not derivable in intuitionistic logic. Note that some of the sequents constructed during the derivation have several formulas to the right and that the right-contraction rule must be used.

\[
\begin{align*}
(R_\land^\land) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\lor) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(L_\land^\land) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\rightarrow) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\neg) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\forall) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\exists) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\forall) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
(R_\land^\exists) & : \quad \varphi : \varphi \quad \frac{\varphi : \varphi \varphi : \varphi}{\varphi : \varphi} \\
\end{align*}
\]
Treating negation in LK is much simpler than in NK: to negate is to pass to the other side of the ‘:’.

Example A Morgan axiom of classical logic:

\[
\begin{align*}
(R_\wedge) : & \frac{\varphi : \varphi}{\varphi, \neg \varphi} \\
(R_\rightarrow) : & \frac{\psi : \psi}{\psi, \neg \psi} \\
(L_\wedge) : & \frac{\neg (\varphi \wedge \psi) : \neg \varphi, \neg \psi}{\varphi \wedge \psi : \varphi, \psi, \neg \varphi, \neg \psi} \\
(R_\lor) : & \frac{\neg (\varphi \wedge \psi) : \neg \varphi \lor \neg \psi}{\neg \varphi \lor \neg \psi : \neg (\varphi \wedge \psi)}.
\end{align*}
\]

Consider the relations between natural deduction and the sequent calculus in the intuitionistic case. For each derivation of sequent $\Gamma : \varphi$ in LJ, there is a derivation of formula $\varphi$ in NJ under the assumptions $\Gamma$. In fact, they are bunches of assumptions which correspond to formulas of $\Gamma$. This correspondence is more than a simple translation of LJ derivations into NJ derivations. It must be viewed operationally. A derivation in LJ is a ‘program’ to construct a derivation in NJ. The right rules correspond exactly to the introduction rules in natural deduction, and work on the conclusion. The left rules correspond to eliminations, but work at the level of assumptions. For example, the left introduction of conjunction

\[
(L_{\land}^1) : \frac{\Gamma, \varphi : \chi}{\Gamma, \varphi \land \psi : \chi}
\]

allows one to construct, from a derivation of $\chi$ under the assumptions $\Gamma$ and $\varphi$, the following derivation of $\chi$ under the assumptions $\Gamma$ and $\varphi \land \psi$:

\[
\begin{align*}
\Gamma & \\
\frac{\varphi \land \psi}{\vdots} & \\
\frac{\chi}{\vdots} &
\end{align*}
\]

Conversely, a derivation in LJ can be described in a non-unique manner by a derivation of LJ constructed using structural and logical rules and the identity axiom—hence without using cut.

The cut-free sequent calculus has important formal properties. A simple glance at the rules will show that each formula appearing as a conclusion also already appears as a premise. To be more precise, the subformulas of a formula $\varphi$ must be redefined: for connectives, it is a standard concept, but for quantifiers, substitutions must be added.

The set of Gentzen subformulas of a formula $\varphi$ is composed of $\varphi$ and

- if $\varphi$ is $\varphi_1 \land \varphi_2$, $\varphi_1 \lor \varphi_2$ or $\varphi_1 \Rightarrow \varphi_2$, the subformulas of $\varphi_1$ and $\varphi_2$,
• if ϕ is ¬ψ₁, the subformulas of ψ₁, and 
• if ϕ is ∀xψ₁ or ∃xψ₁ the subformulas of ϕ[x := t] for every term t.

Finally, the subformulas of a sequent are the subformulas of its component formulas. The following proposition is immediate, by simple inspection of the rules.

**Proposition 4.4** The cut-free LK system satisfies the subformula property: formulas appearing in a sequent derivation are the subformulas of the sequent.

This proposition has significant consequences in automatic theorem-proving. If one is trying to construct a derivation of a sequent, the search space is restricted to the subformulas of the goal. In the case of the propositional calculus, this space is finite, and the calculus is therefore decidable. With quantifiers, there are an infinite number of subformulas.

The subformula property does not hold for all inference systems. In particular, the cut rule, presented here,

\[
\text{(cut)} : \frac{\Gamma : \psi, \Delta \quad \psi : \Delta'}{\Gamma, \Gamma' : \Delta, \Delta'}
\]

does not respect it, since the formula ϕ is destroyed by inference: it is not possible to reconstitute ϕ from the conclusion of a cut. On the other hand, the cut allows much more intelligent proofs than can be done using only logical rules. In natural deduction, it substitutes a derivation for ϕ for the assumption ϕ in a derivation. It therefore justifies, at the level of LJ, the interpretation of an assumption as an undetermined derivation, just like the free variable of the λ-calculus. The ϕ which is cut off appears as a lemma, a general result which is picked up during the proof.

Despite the increased flexibility, the cut rule does not increase the expressivity of LK. Better still, if a derivation uses cuts, it can be effectively transformed into a cut-free derivation. This is Gentzen’s famous Hauptsatz (cut-elimination theorem, 1934), a result similar to Prawitz’s normalization theorem in natural deduction. Its proof uses the L/R symmetry of the rules and the fact that the only axiom (at the leaves of the derivation tree) is the identity. For an axiom, the cut disappears:

\[
\text{(cut)} : \frac{\Gamma : \psi, \Delta \quad \psi : \psi \quad \Delta}{\Gamma : \Delta, \psi} \rightarrow \quad (R_P) : \frac{\Gamma : \psi, \Delta}{\Gamma : \Delta, \psi}.
\]

In the other cases, the cut is pushed down to the leaves of the derivation, and the formula in which there is a cut becomes smaller. For example, in the following case, the cut in ¬ψ is replaced by a cut in ψ:

\[
\text{(cut)} : \frac{\Gamma, \psi : \Delta}{\Gamma : \neg\psi, \Delta} \quad (L_\neg) : \frac{\Gamma' : \psi, \Delta'}{\Gamma', \neg\psi : \Delta'} \rightarrow \quad \text{(cut)} : \frac{\Gamma' : \psi, \Delta'}{\Gamma' : \Delta', \Delta'} \quad \text{(cut)} : \frac{\Gamma, \psi : \Delta}{\Gamma, \Gamma' : \Delta, \Delta'}.
\]

This theorem does not generalize if other axioms are used (for example, in a theory; see below). In that case a proof with cuts is truly more powerful, and
the rule must be added to LK, which has major consequences. However, cut-elimination is not a confluent reduction, and therefore cannot claim to represent computation in a manner analogous to the Curry–Howard correspondence.

4.2.10 Hilbert systems

The third method, the first historically, is the Hilbert system. Derivations of formulas are made as in natural deduction, but without discharging deductions. Instead of being λ-terms, the derivations are first-order terms, like the derivations used in Chapter 1, to prove that a term is well formed. These derivations, viewed as terms, are formed over a signature containing rules as n-ary functional symbols, the n = 0 case corresponding to axioms.

It was shown in Chapter 2 how to translate λ-terms into applicative terms by choosing S and K as constants. From the Curry–Howard correspondence, one can deduce that a natural-deduction derivation, which has a typed λ-term structure, can be translated into a derivation using only modus ponens and formed from axioms. The derivation-term correspondence is described in Table 4.7.

<table>
<thead>
<tr>
<th>Combinatory logic</th>
<th>Hilbert system</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>formula</td>
</tr>
<tr>
<td>applicative term</td>
<td>Hilbert derivation</td>
</tr>
<tr>
<td>type of term</td>
<td>conclusion of derivation</td>
</tr>
<tr>
<td>application (App)</td>
<td>modus ponens (E∋)</td>
</tr>
<tr>
<td>constant symbol(S, K)</td>
<td>axiom name</td>
</tr>
<tr>
<td>types of the constants</td>
<td>axioms</td>
</tr>
<tr>
<td>variable</td>
<td>assumption name</td>
</tr>
<tr>
<td>type of variable</td>
<td>assumption</td>
</tr>
<tr>
<td>combinatory basis</td>
<td>axiom system</td>
</tr>
</tbody>
</table>

Table 4.7 The derivation-term correspondence

The types of S and K are well known and give the first axioms of Hilbert systems (they are polymorphic constants: they should really be written \(S^{\varphi, \chi, \psi}\), and \(K^{\varphi, \psi}\), and then typed combinatory logic would be used):

\[
S : (\varphi \Rightarrow (\chi \Rightarrow \psi)) \Rightarrow ((\varphi \Rightarrow \chi) \Rightarrow (\varphi \Rightarrow \psi)) \\
K : \varphi \Rightarrow (\psi \Rightarrow \varphi)
\]

The Hilbert system for the (minimal) logic of implication is formed from the two axioms S and K and the single rule, modus ponens, or implication elimination,
which corresponds to implication.

\[
\begin{array}{c}
(\text{App}) : \varphi \Rightarrow \psi \vdash \varphi \\
\end{array}
\]

There is no introduction rule: modus ponens and the axioms can simulate \(I_{\Rightarrow}\) in the same manner in which abstraction in the \(\lambda\)-calculus is simulated using the constants \(S\) and \(K\) in combinatory logic. Write \(\vdash_H\) for the deduction relation defined by this inference system: \(\varphi_1, \ldots, \varphi_n \vdash_H \varphi\) if there exists a derivation of \(\varphi\) whose assumptions are included in \(\varphi_1, \ldots, \varphi_n\). Such a derivation is understood as a typed applicative term constructed over the constants \(S\) and \(K\) and the variables whose types are included in \(\varphi_1, \ldots, \varphi_n\).

The identity \(\lambda xx\) can be defined using the applicative rule \(SKK\). It corresponds to the following term, a derivation of \(\varphi \Rightarrow \varphi\) in a Hilbert system,

\[
\begin{array}{c}
(\text{App}) : (\varphi \Rightarrow (\psi \Rightarrow \varphi) \Rightarrow \varphi) \Rightarrow (\psi \Rightarrow (\varphi \Rightarrow \varphi)) \Rightarrow \psi \Rightarrow \varphi \\
(\text{App}) : \psi \Rightarrow \varphi \\
\end{array}
\]

which can be written as the applicative term

\[
(S^{\varphi, \varphi \Rightarrow \varphi, \varphi}K^{\varphi, \varphi \Rightarrow \varphi})K^{\varphi, \varphi}.
\]

As can be seen, simple tautologies can have Hilbert derivations much more complex that their natural deduction counterparts. It is not by adding axioms (for example \(\varphi \Rightarrow \varphi\)) that one can hope to reduce this complexity, similarly to what was observed when \(\lambda\)-terms were compiled into applicative terms. Gentzen systems yield much more natural proofs than do Hilbert systems, at the cost of more important formal difficulties, such as assumption management in natural deduction, or sequent manipulation. The same problem exists in functional programming where one has the choice of languages based on the \(\lambda\)-calculus, with their concept of free and bound variables, and applicative languages with no variable, where a good number of ‘functionals’ are necessary (see Backus’s FP language).

There is a deduction theorem, but unlike that for natural deduction, it is not immediate: \(\varphi \vdash_H \psi\) if and only if \(\vdash_H \varphi \Rightarrow \psi\). The problem is to transform a derivation of \(\psi\) containing \(\varphi\) as assumption into a closed derivation of \(\varphi \Rightarrow \psi\). This is done by simulating abstraction in applicative terms: if \(M\) is a derivation of \(\psi\) and \(X\) is a variable of type \(\varphi\) (possibly appearing in \(M\)), then \([x]M\) must be computed. The proof of a deduction theorem therefore consists of the abstraction algorithm presented in Chapter 2.

The deduction theorem means that implication ‘internalizes’ deduction: \(\vdash_H\) is replaced by \(\Rightarrow\). This will justify the axioms associated with the other connectives. An axiom is associated with each natural deduction rule, by replacing the bar with implication, and by currying the multiple premises for \((I_{\land})\) and \((E_{\lor})\). The result is shown in Table 4.8.
\[ \pi^1 : \varphi \land \psi \Rightarrow \varphi \]
\[ \pi^2 : \varphi \land \psi \Rightarrow \psi \]
\[ (, ) : \varphi \Rightarrow (\psi \Rightarrow (\varphi \land \psi)) \]
\[ \nu_1 : \varphi \Rightarrow \varphi \lor \psi \]
\[ \nu_2 : \psi \Rightarrow \varphi \lor \psi \]
\[ \text{case} : \varphi \lor \psi \Rightarrow ((\varphi \Rightarrow \chi) \Rightarrow ((\psi \Rightarrow \chi) \Rightarrow \chi)) \]
\[ \bot : \bot \Rightarrow \varphi \]

Table 4.8 A Hilbert system: propositional axioms

The transformation is more delicate for quantifiers. Rules \((I_\exists)\) and \((E_\forall)\) directly yield substitution axioms:

\[
\begin{align*}
(\sigma_\exists) : & \varphi[x := t] \Rightarrow \exists x \varphi \\
(\sigma_\forall) : & \forall x \varphi \Rightarrow \varphi[x := t]
\end{align*}
\]

However, because of the restrictions on free variables, one cannot transform the rules \((I_\forall)\) and \((E_\exists)\) into axioms. In fact, an axiom \(\varphi(a) \Rightarrow \forall x \varphi\) corresponding to \((I_\forall)\) would systematically introduce the free variable \(a\), which is illegal in the derivation of NK. The Hilbert system therefore uses two generalization rules with the same application restrictions:

\[
\begin{align*}
(Gen_\exists) : & \varphi(a) \Rightarrow \chi \\
(Gen_\forall) : & \chi \Rightarrow \varphi(a)
\end{align*}
\]

Variable \(a\) must not be free in \(\chi\).

Finally, for classical logic, the excluded-middle axiom is added:

\[
(EM) : \varphi \lor \neg \varphi
\]

Hilbert systems tend to be favored by mathematicians, who are used to working with axioms, while natural deduction and the sequent calculus, much studied by logicians in proof theory, have deep and promising links with computer science.

### 4.3 Normal forms

In classical logic, the tautologies which were presented show the syntactic redundancy of the first-order languages: the connectives used are not independent and satisfy certain algebraic relations such as commutativity and distributivity. The formulas can then be put into ‘normal form’. It is standard to represent a formula by a tree whose leaves are atoms and whose internal nodes are connectives and quantifiers. The usual normal forms consist of pushing negations towards the leaves and quantifiers towards the root. The \(\land/\lor\) case requires choosing which distributivity to use.
4.3.1 Prenex formulas

Definition 4.3 A formula of the form $Q_1 x_1 \ldots Q_n x_n \varphi^o$, with $n \geq 0$, $Q_i \in \{\forall, \exists\}$, $\varphi^o \in L^c[X]$, i.e., $\varphi^o$ quantifier-free, and $x_1, \ldots, x_n \in \text{var} \, \varphi^o$, is called prenex.

Write $Q\vec{x}$ instead of $Q_1 x_1 \ldots Q_n x_n$ and $Q'\vec{x}$ the prefix obtained by replacing each $\forall$ by $\exists$ and conversely.

Proposition 4.5 For each formula $\varphi$, there exists a prenex formula $\psi$ such that $\vdash \varphi \iff \psi$ and $\text{var}(\varphi) = \text{var}(\psi)$.

The following tautologies are used:

\begin{align*}
\vdash Qx\varphi(x) \lor \psi & \iff Qx(\varphi(x) \lor \psi), \quad \text{if } x \not\in \text{var}(\psi) \\
\vdash Qx\varphi(x) \land \psi & \iff Qx(\varphi(x) \land \psi), \quad \text{if } x \not\in \text{var}(\psi) \\
\vdash \neg Qx\varphi & \iff Q'x(\neg \varphi).
\end{align*}

Proof By induction over formulas. It suffices to distinguish four cases: atoms, $\lor$, $\land$, and $\exists$.

- If $\varphi$ is atomic, $\varphi$ is prenex.
- If $\varphi$ is $\varphi_1 \lor \varphi_2$, with $\vdash \varphi_1 \iff \psi_1$ and $\vdash \varphi_2 \iff \psi_2$, and $\psi_1 = Q\vec{x} \varphi_1^o$ and $\psi_2 = Q\vec{y} \varphi_2^o$ are prenex formulas. Suppose that the bound variables are renamed so that they are distinct and do not intersect with $\text{var}(\varphi)$; then

\begin{align*}
\vdash \varphi & \iff \psi_1 \lor \psi_2 \quad (\text{property of } \iff) \\
\vdash \psi_1 \lor \psi_2 & \iff Q\vec{x}\vec{y}(\psi_1^o \lor \psi_2^o) \quad (\text{tautologies})
\end{align*}

- If $\varphi$ is $\neg \chi$ with $\vdash \chi \iff \psi$ and $\psi = Q\vec{x} \varphi^o$, then $\vdash \varphi \iff Q'\vec{x}(\neg \varphi^o)$.
- If $\varphi$ is $\exists x \chi$ with $\vdash \chi \iff \psi$ and $\psi = Q\vec{y} \varphi^o$, then $\vdash \varphi \iff \exists x Q\vec{y} \varphi^o$.

Remark It is easy to transform the proof of the theorem into a procedure computing $\psi$ from $\varphi$. However, according to the order of use of the prenex transformations, distinct prenex formulas $\psi$ will be obtained. What is important is that they be equivalent to each other as well as with the initial formula. For example, $\forall x \exists y \forall z (A \land B)$ and $\forall x \forall z \exists y (A \land B)$ are two prenex formulas equivalent to $\forall x(\exists y A \land \forall z B)$, where $A$ and $B$ are atomic, $y \not\in \text{var}(B)$ and $z \not\in \text{var}(A)$.

Computing an equivalent prenex formula can be seen as rewriting using the system obtained by replacing the $\iff$ in the above tautologies by $\rightarrow$.  

\[\square\]
4.3.2 Disjunctive and conjunctive formulas

Disjunctive and conjunctive normal forms are useful for the formulas of the propositional calculus, as well as for the quantifier-free formulas of first-order languages.

A literal is an atom or the negation of an atom. If \( l \) is a literal, define the conjugate literal \( l' \) of \( l \) by

\[
l' = \begin{cases} 
\lnot x & \text{if } l \text{ is an atom } x, \\
x & \text{if } l \text{ is the negation of an atom } x.
\end{cases}
\]

Definition 4.4 Let \( \varphi \) be a formula of the propositional calculus: if

\[
\varphi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{p_i} \varphi^{i}_{j} \quad (\text{respectively } \bigvee_{i=1}^{m} \bigwedge_{j=1}^{p_i} \varphi^{i}_{j}),
\]

where the \( \varphi^{i}_{j} \) are literals and \( \varphi \) is called a conjunctive (respectively disjunctive) formula.

Proposition 4.6 For each formula \( \varphi \) of the propositional calculus, there exists a conjunctive formula \( \varphi^c \) and a disjunctive formula \( \varphi^d \) such that

\[
\vdash \varphi \iff \varphi^c \quad \text{and} \quad \vdash \varphi \iff \varphi^d.
\]

Proof Begin by replacing the connectives \( \Rightarrow \) and \( \iff \) appearing in \( \varphi \) with \( \lnot, \lor \) and \( \land \), then reason by induction over formulas:

- If \( \varphi \) is atomic, \( \varphi^c = \varphi^d = \varphi \).
- If \( \varphi = \chi \land \psi \), let \( \varphi^c = \chi^c \land \psi^c \); furthermore, if by induction \( \vdash \chi \iff \chi^d = V_i \chi_i \),

  \( \text{and } \vdash \psi \iff \psi^d = V_j \psi_j \), let \( \varphi^d = V_{i,j}(\chi_i \land \psi_j) \).
- If \( \varphi = \chi \lor \psi \), proceed in a dual (in \( c/d \)) manner.
- If \( \varphi = \lnot \psi \), by induction \( \psi^c = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{p_i} \psi^i_{j} \). Let \( \varphi^d = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{p_i} \psi^i_{j} \), constructing \( \varphi^c \) is analogous, starting from \( \psi^d \).

These transformations are equivalences for \( \vdash \). \( \square \)

Example

\[
(A \land \lnot(B \lor \lnot C)) \Rightarrow \lnot B \Rightarrow (A \land B).
\]

Eliminate the \( \Rightarrow \):

\[
\lnot(A \land \lnot(B \lor \lnot C)) \lor (\lnot B \lor (A \land B)).
\]

Push the \( \lnot \) towards the leaves and eliminate the \( \lnot \lnot \):

\[
(A \lor (B \lor \lnot C)) \lor (B \lor (A \land B)).
\]
Collect the $\lor$:

\[
(\neg A \lor B \lor \neg C \lor B \lor A) \land (\neg A \lor B \lor \neg C \lor B \lor B),
\]

which is a conjunctive formula equivalent to the initial formula.

These transformations can also be written using noetherian but non-confluent rewriting systems. The one computing the conjunctive forms is given by Table 4.9. The proof of termination uses an rpo simplification order (see Chapter 2).

<table>
<thead>
<tr>
<th>$\phi \iff \psi$</th>
<th>$\to$</th>
<th>$(\phi \Rightarrow \psi) \land (\psi \Rightarrow \phi)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi \Rightarrow \psi$</td>
<td>$\to$</td>
<td>$\neg \phi \lor \psi$</td>
</tr>
<tr>
<td>$\neg \neg \phi$</td>
<td>$\to$</td>
<td>$\phi$</td>
</tr>
<tr>
<td>$\neg (\phi \land \psi)$</td>
<td>$\to$</td>
<td>$\neg \phi \lor \neg \psi$</td>
</tr>
<tr>
<td>$\neg (\phi \lor \psi)$</td>
<td>$\to$</td>
<td>$\neg \phi \land \neg \psi$</td>
</tr>
<tr>
<td>$\phi \lor (\psi \land \chi)$</td>
<td>$\to$</td>
<td>$(\phi \lor \psi) \land (\phi \lor \chi)$</td>
</tr>
<tr>
<td>$(\phi \land \psi) \lor \chi$</td>
<td>$\to$</td>
<td>$(\phi \lor \chi) \land (\psi \lor \chi)$</td>
</tr>
</tbody>
</table>

Table 4.9 Computing conjunctive forms

The normal forms of this system are conjunctive formulas, but a formula can be reduced to several conjunctive formulas.

### 4.4 Theories

In this section, we will work with first-order languages. Be it in mathematics, programming or artificial intelligence, one is always working inside a theory which defines 'common knowledge'. The deduction relation $\vdash$ is that of classical logic.

#### 4.4.1 Definitions and examples

A theory is a set of first-order language formulas $L[X]$. For the moment, unless one is going to enumerate the formulas, a good way to define a theory is to generate it using the deduction relation $\vdash$.

Definition 4.5 Given a subset $\Gamma$ of $L[X]$, whose elements are called axioms (of the theory), the theory generated by $\Gamma$ is the set of formulas $\phi$ of $L[X]$, called theorems, such that $\Gamma \vdash \phi$, i.e., closed under $\vdash$.

The predicate calculus

It is the theory without axioms, the set of $\phi \in L[X]$ such that $\vdash \phi$. Its theorems are the tautologies, some of which have already been seen.
Equality theories

Suppose that $\Sigma$ contains a binary relation symbol $E$ (or $=$, if there is no problem of confusion with other uses of $=$, such as for $\beta$-conversion). The set $Ax_E$ of equality axioms is:

\[(Ax_E1) \forall x Exx,\]
\[(Ax_E2) \forall xy Eyx \Rightarrow Eyx,\]
\[(Ax_E3) \forall xyz Eyx \land Eyz \Rightarrow Ezx,\]
\[(Ax_Ef) \forall x_1 \ldots x_n y_1 \ldots y_n Ex_1 y_1 \land \ldots \land Ex_n y_n \Rightarrow Ef x_1 \ldots x_n fy_1 \ldots y_n,\]
\[(Ax_ER) \forall x_1 \ldots x_n y_1 \ldots y_n Ex_1 y_1 \land \ldots \land Ex_n y_n \Rightarrow (Rx_1 \ldots x_n \Rightarrow Ry_1 \ldots y_n),\]

for each functional symbol $f$ and relational symbol $R$ in $\Sigma$.

By convention, for any theory in a language containing an equality symbol $E$, the axioms $Ax_E$ will always be implicit in the statement of the (other) axioms. The signature, the language and the theory are said to be with equality.

Since equality has a very special role compared to the other relational symbols, its properties can be incorporated into the inference system, by changing the axioms of the inference rules. For example, $Ax_E2$ would be transformed into

\[
\begin{align*}
t &= t' \\
t' &= t
\end{align*}
\]

Note that this rule would allow an infinite derivation that would exchange terms on either side of an equality.

Group theory

Over the usual signature $\Sigma = \{e, \cdot, ^{-1}, =\}$, there are three axioms,

\[\forall x (x.e = x),\]
\[\forall x (x.x^{-1} = e),\] and
\[\forall xyz ((x.y).z = x.(y.z)).\]

It is of course assumed here that the system includes the five $Ax_=$ axioms, listed above.

Many other algebraic structures, such as rings, fields and lattices, have an axiomatic definition in a first-order language.

Peano arithmetic

This is one of the most important theories studied by logicians. It is the standard definition for arithmetic and is used by mathematicians as well as computer scientists. The signature is $\Sigma = \{0, s, +, -, \cdot, =\}$, where $s$ is a unary functional symbol for the successor function.
The axioms \((AP)\) are:

\[
\forall x \neg (sx = 0), \\
\forall xy sx = sy \Rightarrow x = y, \\
\forall x x + 0 = x, \\
\forall xy x + sy = s(x + y), \\
\forall x x.0 = 0, \\
\forall xy x.sy = x.y + x, \\
\forall x \neg(x < 0), \\
\forall xy x < sy \iff x < y \lor x = y, \\
\phi(0) \land \forall x(\phi(x) \Rightarrow \phi(sx)) \Rightarrow \forall x \phi(x).
\]

The last line is the induction axiom scheme: there are a countably infinite number of such axioms for each formula \(\phi\) and each variable \(x\) not bound in \(\phi\). It allows proofs by induction.

A weaker theory, i.e., having fewer axioms, than Peano arithmetic is often introduced by eliminating the induction axiom scheme and by adding the axiom \(\forall xy(x < y \lor x = y \lor y < x)\) which can no longer be proved from the other axioms (see exercise 27). This theory is called \textit{elementary arithmetic}, having the advantage of being finitely axiomatizable, which is not the case for Peano arithmetic, according to the Ryll–Nardzewski theorem. Elementary arithmetic was introduced by Mostowski, Tarski and R.M. Robinson to prove undecidability properties.

### 4.4.2 General properties

\textit{Consistency} is the first property to demand of any theory: a subset \(\Gamma\) of \(L_\Sigma\) (or the theory \(T\) which it generates) is consistent if \(\Gamma \not\vdash \bot\), i.e., if there does not exist \(\varphi \in L[X]\) such that \(\Gamma \vdash \varphi\) and \(\Gamma \vdash \neg \varphi\), or equivalently, by rule \(E_\neg\), if \(T \not\models L[X]\). A theory is inconsistent if it is not consistent.

Inconsistency is directly related to the deduction relation.

\textit{Proposition 4.7}

1. \(\Gamma \vdash \varphi\) if and only if \(\Gamma \cup \{\neg \varphi\}\) is inconsistent;
2. \(\Gamma \vdash \neg \varphi\) if and only if \(\Gamma \cup \{\varphi\}\) is inconsistent.

\textit{Proof}

1. If \(\Gamma \vdash \varphi\), then \(\Gamma, \neg \varphi \vdash \varphi\) and \(\Gamma, \neg \varphi \vdash \neg \varphi\) as well: \(\Gamma \cup \{\neg \varphi\}\) is inconsistent. Conversely, by contradiction:

\[
\Gamma \quad \neg \varphi \\
\hline
(C) : \quad 1 \\
\varphi
\]


2. Idem, with rule ($I\neg$) instead of ($C$).

Completeness is an important, strong property, which is not satisfied by all
interesting theories. A theory of axioms $\Gamma$ is complete if for every closed
formula $\varphi$, $\Gamma \vdash \varphi$ or $\Gamma \vdash \neg \varphi$.

Examples Field theory is not complete: one can neither prove $1 + 1 = 0$ nor
$1 + 1 \neq 0$. But the theory of algebraically closed fields of characteristic 0 is
complete. Peano arithmetic is not complete: see Gödel's incompleteness theorem
in Chapter 7.

Decidability is a fundamental property, related to the concept of computability
and at the heart of theoretical problems in automatic theorem-proving: a theory
with axioms $\Gamma$ is decidable if ‘there exists an algorithm which, given a formula $\varphi$,
determines if $\Gamma \vdash \varphi$ or $\Gamma \not\vdash \varphi$’.

A precise definition of decidability will be given in Chapter 7. A theory which
is not decidable is called undecidable.

Examples Arithmetic, group theory and the predicate calculus (with a symbol of
arity $> 1$) are undecidable. Presburger arithmetic, commutative group theory and
algebraically closed fields of characteristic 0 are decidable.

4.4.3 Operations over theories

Theories are objects which can be manipulated. Four operations will be considered:
extension, relativization, implementation and skolemization.

Extensions

Generally, a theory $T'$ is an extension of a theory $T$ if $T \subseteq L_\Sigma[X]$, $T' \subseteq L_{\Sigma'}[X]$,
$\Sigma \subseteq \Sigma'$ and $T \subseteq T'$. There are two ways to extend a theory: add more symbols to
the signature, or add more axioms.

An important mechanism is that of definitional extension. The idea is to intro-
duce a new symbol with an axiom defining it. There are two cases, according to
the functional or relational nature of the symbol. Let $T \subseteq L_\Sigma$.

Relational extension Let $\chi \in L_\Sigma[x_1, \ldots, x_n]$. Form $\Sigma' = \Sigma \cup \{R\}$, where $R$ is a
new $n$-ary relational symbol and $T' = T \cup \{\text{def}(R)\}$ where $\text{def}(R)$ is the axiom
defining $R$:

$$\forall x_1 \ldots x_n (Rx_1 \ldots x_n \iff \chi(x_1, \ldots, x_n)).$$

Note that $R$ does not appear in $\chi$: it is an explicit definition, not ‘recursive’.
Example Add to the signature of arithmetic a unary symbol 'even' with the axiom\def (even):
\[ \forall x (\text{even}(x) \iff \exists y (x = y + y)). \]

Functional extension Let \[ \chi \in L[\Sigma][x_1, \ldots, x_n, y] \]. To state the functional characteristic of \( \chi \), it is supposed that
\[ T \vdash \forall x_1 \ldots x_n \exists y [\chi \land \forall y' (\chi[y := y'] \Rightarrow y = y')], \]
which can be abbreviated by
\[ T \vdash \forall x \exists! y \chi. \]
The \( \exists! \) is read as 'there exists a unique'. In this case, let \( \Sigma' = \Sigma \cup \{f\} \), \( f \) a new \( n \)-ary functional symbol, and \( T' = T \cup \{\text{def}(f)\} \), where \( \text{def}(f) \) is the axiom defining \( f \):
\[ \forall x_1 \ldots x_n y (y = f x_1 \ldots x_n \iff \chi(x_1, \ldots, x_n, y)). \]

Example In arithmetic, 'max' is defined by the axiom:
\[ \forall x_1 x_2 y (y = \max(x_1, x_2) \iff (y = x_1 \lor y = x_2) \land x_1 \leq y \land x_2 \leq y). \]

In each case, there is a translation \( \delta : L[\Sigma'][X] \rightarrow L[\Sigma][X] \). For the relational case, \( \delta(\varphi) \) is obtained by replacing \( R t_1 \ldots t_n \) with \( \chi(t_1, \ldots, t_n) \) in \( \varphi \). For the functional case, it is not as simple since variables and existential quantifiers must be added to the \( t_i \). The properties of this translation, with \( \varphi \in L[\Sigma'][X] \), are
\[ T' \vdash \varphi \iff \delta(\varphi) \quad \text{and} \quad (4.1) \]
\[ T' \vdash \varphi \text{ if and only if } T \vdash \delta(\varphi). \quad (4.2) \]
Evidently, \( \delta(\varphi) = \varphi \) if \( \varphi \in L[\Sigma][X] \).

A theory \( T' \) is a definitional extension of \( T \) if \( T' \) is obtained from \( T \) by a finite number of relational or functional extensions. This type of extension is common place. 'Number theory' is a definitional extension of Peano arithmetic with all the useful functions and relations. Mathematics are, from a truly formalist point of view, a gigantic definitional extension of set theory. In programming, for example in Lisp, new function symbols are added as needed, along with their definitions.

According to the above properties, if \( \varphi \in L[\Sigma], T' \vdash \varphi \) if and only if \( T \vdash \varphi \): new theorems are not added over the initial signature. The theory \( T' \) is a conservative extension of \( T \). This property (weaker than a definitional extension) is important. In particular, if the extension \( T' \) of \( T \) is conservative, then \( T' \) is consistent if \( T \) is. In computer science, one talks rather in this case of 'hierarchical consistency'.
Set theory  Even though axiomatic set theory is of little interest to computer science, there are good reasons for familiarity with it. First, from a technical point of view, it illustrates better than any other theory the definitional extension of a theory. Second, it was in this context that the various paradoxes of the naive theory were eliminated and that some of the most famous problems of logic, such as the independence of the axiom of choice, were solved.

The language of Zermelo-Fraenkel (ZF) set theory includes equality and uses a single binary relation symbol, $\in$. All variables are of the same nature and designate sets, unlike those in the naïve theory which distinguishes set and element. However, the atomic formula $x \in y$ can be read as ‘$x$ is an element of set $y$’.

Due to the poverty of the language (comparable to that of the $\lambda$-calculus), it is necessary to extend it by adding new symbols, along with their definitions. Even the definition of the axioms will be facilitated by the successive introduction of new symbols. For example, the relational symbol $\subseteq$ is defined by $x \subseteq y \iff \forall u(u \in x \Rightarrow u \in y)$. To introduce a functional symbol always requires a previous proof.

Let $\varphi \in L_{ZF}[\vec{x}, y]$, where the ‘parameters’ $\vec{x}$ and the variable $y$ are distinct. The formula

$$\exists ! z \forall y (y \in z \iff \varphi(\vec{x}, y)),$$

stating that ‘the $y$ such that $\varphi(\vec{x}, y)$ form a set’, is abbreviated by $\text{Set}\{y; \varphi(\vec{x}, y)\}$.

If $ZF \vdash \text{Set}\{y; \varphi(\vec{x}, y)\}$ and $\vec{x} = x_1 \ldots x_n$, introduce an $n$-ary functional symbol $F_\varphi$ along with its definition:

$$y \in F_\varphi(\vec{x}) \iff \varphi(\vec{x}, y).$$

The expression $\{y; \varphi(\vec{x}, y)\}$ can be used as a term of the language, instead of $F_\varphi(\vec{x})$. However, writing $\{y; \varphi(\vec{x}, y)\}$ has no meaning so long as $\text{Set}\{y; \varphi(\vec{x}, y)\}$ has not been proven. It is, however, a good way to introduce ‘paradoxes’ of which the simplest ones are $\{x; x \notin x\}$ and $\{x; x = x\}$. This extension is allowed by the axiom of extensionality which follows.

Extensionality  ‘Two sets are equal if they have the same elements:’

$$\forall x(z \in x \iff z \in y) \Rightarrow x = y.$$

Union  ‘Joining a family $x$ of sets is a set $\bigcup(x)$:’

$$\text{Set}\{y; \exists u (u \in x \land y \in u)\}$$

Power set  ‘The set of the subsets of a set $x$ is a set $\mathcal{P}(x)$:’

$$\text{Set}\{y; y \subseteq x\}.$$

Infinite  ‘There exists a set $x$ having an element $z$ which is the empty set, and for which, for any element $m$, there exists an element $n$ whose elements are those of $m$ along with $m$ itself.’
Replacement scheme 'The image of a set by an operation over sets is a set:'

\[ \text{Set}\{y; \exists x(x \in u \land y = F_\varphi(x))\}, \]

for each formula \( \varphi \) such that \( \text{Set}\{y; \varphi(x, y)\} \).

Foundation 'Every non-empty set \( x \) has an element \( y \) distinct from \( x \).'

A few words about the other axioms. The axiom of choice, which states that a family of non-empty sets is non-empty, is crucial to the development of ordinary mathematics: it allows the proof of the facts that every vector space has a basis, that the product of compact spaces is compact and that the union of a countable number of countable sets is countable, among others. The continuum hypothesis states that there exist as many ways of well-ordering \( \mathbb{N} \) as there are real numbers. It is little used, except in 'fine' real analysis. In 1928 Gödel showed the consistency of the axiom of choice and of the continuum hypothesis relative to ZF [56], but it was only in 1963 that Cohen showed that if ZF is consistent, these axioms cannot be theorems of ZF [21].

Relativization

Let \( D \) be a unary relational symbol. The relative quantifiers \( \forall^D \) and \( \exists^D \) are defined by: \( \forall^D \varphi \) is the abbreviation of \( \forall x(D(x) \Rightarrow \varphi) \), and \( \exists^D \varphi \) is the abbreviation of \( \exists x(D(x) \land \varphi) \). They are sometimes written \( (\forall x \in D) \varphi \), and \( (\exists x \in D) \varphi \).

Example Write \( (\forall x > 0) \varphi \) instead of \( \forall x(x > 0 \Rightarrow \varphi) \).

If \( \varphi \) is a proposition, its relativization \( \varphi^D \) is obtained by relativizing the quantifiers in \( \varphi \) under \( D \).

Implementations

A theory \( T \subseteq L_\Sigma \) is to be implemented in another theory \( T' \subseteq L_{\Sigma'} \). A unary relational symbol \( D \) over \( \Sigma' \) is defined as the 'domain' of the implementation. Similarly, each symbol \( f \) or \( R \) (in \( \Sigma \)) is implemented by a symbol \( f' \) or \( R' \) (in \( \Sigma' \)) of the same arity; these symbols can be defined in an extension of \( T' \). This yields a translation \( \gamma \) of \( L_\Sigma \) in \( L_{\Sigma'} \) obtained by replacing \( f \) and \( R \) with \( f' \) and \( R' \), and by relativising the quantifiers under \( D \). For each \( f' \), axioms '\( D \) is non-empty' and '\( f' \) operates over \( D \)' are added:

\[ T' \vdash \exists x D, \]
\[ T' \vdash D(x_1) \land \ldots \land D(x_n) \Rightarrow D(f'(x_1, \ldots, x_n)). \]

If \( T \) is implemented in \( T' \), \( T' \vdash \gamma(\varphi) \) for each axiom of \( T \).

In logic, this mechanism is mainly used in proofs of undecidability. In computer science, conveniently modified, it allows the definition of the implementation of a data structure using another data structure. Extensions and implementations are the two main construction methods for data structures.
Skolemization

Named after the logician T. Skolem, skolemization is a syntactic transformation of formulas, allowing the elimination of one of the two kinds of quantifiers. There therefore exist two kinds of transformation, one which constructs universal formulas, used for proofs by contradiction, the other which constructs existential formulas, used for direct proofs. Their use is widespread in mathematical reasoning, as is illustrated by the two examples of universal skolemization:

- \( \exists x \, P(x) \) was proven; let \( a \) such that \( P(a), \ldots \).
- \( \forall x \exists n \, Q(x, n) \) was proven; let \( N(e) \) such that \( Q(x, N(e)), \ldots \).

In these examples, new symbols were added to the signature: a ‘Skolem constant’ \( a \) and a ‘Skolem function’ \( N \).

To simplify, here is a description of a universal skolemization step for a prenex formula (this is an unnecessary restriction, and is in general a hindrance, which will be lifted below).

Let \( \psi \in L_{\Sigma}[x_1, \ldots, x_k, x_{k+1}] \) and \( \varphi \equiv \forall x_1 \ldots \forall x_k \exists x_{k+1} \psi \in L_{\Sigma} \). Introduce a new \( k \)-ary functional symbol \( F \) and define

\[
\tilde{\Sigma} = \Sigma \cup \{F\}, \\
\tilde{\varphi} = \forall x_1 \ldots \forall x_k \psi[x_{k+1} := Fx_1 \ldots x_k] \in L_{\tilde{\Sigma}}.
\]

If \( \varphi \) is a prenex proposition containing \( p \) existential quantifiers, then \( p \) iterations of the transformation \( \varphi \mapsto \tilde{\varphi} \) yield a prenex universal proposition \( \varphi^u \) over a signature \( \Sigma^u \) extended by \( p \) new symbols.

If \( \varphi \) is an arbitrary (non-prenex) proposition, there does not exist a unique prenex form equivalent to \( \varphi \). Furthermore, certain prenex transformations can create parasitic dependencies in the skolemized form.

**Example** Let \( \varphi = \forall x \, P(x) \land \exists y \, Q(y) \). Both of the prenex formulas \( \forall x \exists y \, (P(x) \land Q(y)) \) and \( \exists y \forall x \, (P(x) \land Q(y)) \) are equivalent to \( \varphi \). Their skolemized forms are, respectively, \( \forall x \, (P \land Q(Fx)) \) and \( \forall x \, (P \land Q(a)) \). Here \( F \) (respectively \( a \)) is a Skolem unary function (respectively constant). Both are consistent propositions, as is \( \varphi \), but the second is harder to test than the first.

It is therefore preferable to compute the skolemized form without using an intermediate prenex form. Instead, the skolemized propositions \( \varphi^u \) (universal) and \( \varphi^e \) (existential) of a proposition \( \varphi \) are defined by the rules in Table 4.10 where \( s \) is \( u \) or \( e \), \( s \mapsto s' \) exchanges \( u \) and \( e \), \( L \) designates a literal, \( \{u_1, \ldots, u_n\} = \text{var}(\chi) \), and \( F \) is a new \( n \)-ary symbol.

Computing \( \varphi^u \) consists of eliminating each positive quantifier \( \forall x \) or negative quantifier \( \exists x \) and substituting in its scope the term \( Fu_1 \ldots u_n \) a \( x \), where \( u_1, \ldots, u_n \) are the universally quantified variables in the scope in which \( x \) is found.
$L^s = L$
$(\chi \land \psi)^s = \chi^s \land \psi^s$
$(\chi \lor \psi)^s = \chi^s \lor \psi^s$
$(\forall x \chi)^u = \forall x \chi^u$
$(\exists x \chi)^u = (\chi[x := F_u \ldots u_n])^u$
$(\forall x \chi)^e = (\chi[x := F_u \ldots u_n])^e$
$(\exists x \chi)^e = \exists x \chi^e$
$(\neg \chi)^s = \neg (\chi^s)$
$(\chi \Rightarrow \psi)^s = \chi^s \Rightarrow \psi^s$

**Table 4.10 Skolemization**

**Example**

Let $\varphi = \forall x (\forall y A(x, y) \Rightarrow \neg \exists z B(x, z))$.

Then $\varphi^u = \forall x (A(x, F_x) \Rightarrow \neg \exists z B(x, z))$

and $\varphi^e = \forall y A(a, y) \Rightarrow \neg B(a, b)$,

where $F$, $a$ and $b$ are symbols introduced by skolemization. □

In Chapter 4, it will be shown semantically that

1. $\vdash \varphi^u \Rightarrow \varphi$ and $\vdash \varphi \Rightarrow \varphi^e$;
2. $\varphi$ is consistent if and only if $\varphi^u$ is consistent; and
3. $\varphi$ is a tautology if and only if $\varphi^e$ is consistent.

**Exercises**

1. Given a signature $\Sigma$, find $\Sigma'$ such that $L_{\Sigma}[X] = T_{\Sigma}$.

2. Show that if $\Gamma \vdash_M \varphi$, there exists a finite $\Gamma' \subseteq \Gamma$ such that $\Gamma' \vdash_M \varphi$.

3. Give the typed $\lambda$-terms corresponding to the minimal logic derivations of the formulas:

   $(\varphi \lor \psi) \Rightarrow (\varphi \Rightarrow \psi) \Rightarrow \psi,$
   $(\varphi \Rightarrow (\psi \Rightarrow \chi)) \Rightarrow ((\varphi \land \psi) \Rightarrow \chi),$
   $((\varphi \Rightarrow \psi) \lor (\varphi \Rightarrow \chi)) \Rightarrow \varphi \Rightarrow (\psi \lor \chi).$

4. State and derive introduction and elimination rules for the connective $\leftrightarrow$.

---

2Hints to the exercises labeled with a $\star$ can be found at the end of the book.
5. State and prove compatibility properties of proven equivalence for $\land$, $\lor$, $\Rightarrow$ and $\neg$.

6. Prove the Morgan axiom of minimal logic:

$$\vdash_M \neg(\varphi \lor \psi) \iff (\neg \varphi \land \neg \psi).$$

7. Prove in minimal logic that $\varphi \Rightarrow \neg \neg \psi$, $\neg \psi \Rightarrow \neg \varphi$ and $\neg \neg \varphi \Rightarrow \neg \neg \psi$ are equivalent formulas. Prove they are all consequences of $\neg \neg (\varphi \Rightarrow \psi)$, which follows from $\varphi \Rightarrow \psi$, which in turn follows from $\neg \neg \varphi \Rightarrow \psi$. Prove that $\neg \neg (\varphi \Rightarrow \psi)$ is a consequence of $\neg \neg \varphi \Rightarrow \neg \neg \psi$ in intuitionistic logic. Prove that all these formulas are equivalent in classical logic.

8. Prove

$$\vdash_M \neg \neg \neg \varphi \iff \neg \varphi,$$
$$\vdash_I \neg \neg (\neg \neg \varphi \Rightarrow \varphi).$$

9. Prove the minimal tautologies

$$\vdash_M \exists x \neg \varphi \Rightarrow \neg (\forall x \varphi),$$
$$\vdash_M \neg (\exists x \varphi) \iff \forall x \neg \varphi,$$
$$\vdash_M \neg \exists x (\neg \varphi) \iff \neg \forall x (\neg \neg \varphi).$$

10. Prove the minimal tautologies

$$\vdash_M \exists x (\varphi \Rightarrow \psi(x)) \Rightarrow (\varphi \Rightarrow \exists x \psi(x)) \quad \text{if } x \notin \text{var } \varphi,$$
$$\vdash_M (\forall x \varphi(x) \lor \psi) \Rightarrow \forall x (\varphi(x) \lor \psi) \quad \text{if } x \notin \text{var } \psi.$$

11.* Build derived rules, analogous to $\forall \forall$, for $\Rightarrow$, $\lor$ and $\land$ (they have no logical interest). Interpret them using the Curry–Howard correspondence. They lead to reduction rules over the typed $\lambda$-terms (the $\eta$ rules, which are important in computation).

12. Derive an instantiation rule for the existential quantifier, analogous to $\forall \forall$, and explain the use of this rule at the level of the assumptions necessary to prove a formula.

13. Prove that the rules $(\neg \neg)$ and $(C)$ are derivable from the axiom of the excluded middle and that $(C)$ is derivable from $(\neg \neg)$.

14.* Prove the classical tautologies

$$\vdash_C \neg (\forall x \varphi) \Rightarrow \exists x (\neg \varphi),$$
$$\vdash_C \neg (\varphi \land \psi) \Rightarrow \neg \varphi \lor \neg \psi.$$
15. Prove the classical tautologies
\[ \vdash_C (\varphi \Rightarrow \exists x \psi(x)) \Rightarrow \exists x (\varphi \Rightarrow \psi(x)) \quad \text{if} \ x \notin \text{var} \varphi, \]
\[ \vdash_C \forall x (\varphi(x) \vee \psi) \Rightarrow (\forall x \varphi(x) \vee \psi) \quad \text{if} \ x \notin \text{var} \psi. \]

16. Prove ‘Peirce’s axiom’ \((\varphi \Rightarrow \psi) \Rightarrow \varphi) \Rightarrow \varphi\) in the NK system.

17.* Derive \(((\varphi \Rightarrow \chi) \Rightarrow \psi) \Rightarrow ((\psi \Rightarrow \varphi) \Rightarrow (\omega \Rightarrow \varphi))\) (‘Lukasiewicz’s axiom’) in the sequent calculus.

18. Define a rewriting system to compute disjunctive forms.

19. Give an example of a formula having two distinct conjunctive normal forms.

20. Prove \(\{Ax_{E1}, \forall x y z (E x y \land E x y \Rightarrow E x z)\} \vdash Ax_{E2} \land Ax_{E3}.\)

21.* State the formulas of \(L_{\Sigma} (\Sigma \text{ with equality})\) expressing the existence of at least (respectively at most, exactly) three elements.

22. Show that the equality axioms are derivable using the two rules
\[
\frac{\text{t = t}}{\alpha[x := t]} \quad \text{and} \quad \frac{\text{t = t'}}{\alpha[x := t']},
\]
where \(t, t'\) are terms and \(\alpha\) is an atomic formula.

23.* Give the axioms of group theory using the signature \(\Sigma' = \{e, \cdot, =\}\), then using \(\Sigma'' = \{e, m\}\), where \(m\) is a ternary relational symbol such that \(m(x, y, z)\) states that \(z = x \cdot y\).

24. Give signatures and axioms for theories for commutative rings, for fields of characteristic \(p, p \geq 0\) [the characteristic of a field is the least integer \(p\) such that \(p.1 = 1 + \ldots + 1\) is null,] and for algebraically closed fields.

25. Give arithmetic formulas expressing that:
- \(x\) and \(y\) are relatively prime;
- \(x\) is the smallest prime strictly greater than \(y\).

26.* Write AE for elementary arithmetic. Prove that
- \(AE \vdash s^m(0) = s^n(0)\), if \(m = n\);
- \(AE \vdash \neg(s^m(0) = s^n(0))\), if \(m \neq n\);
- \(AE \vdash s^m(0) + s^n(0) = s^p(0)\), if \(m + n = p\);
- \(AE \vdash s^m(0) \cdot s^n(0) = s^p(0)\), if \(m \cdot n = p\).

27. Show that \(\forall x y (x < y \lor x = y \lor y < x)\) is a theorem of Peano arithmetic.
28. Let $p$ be an integer $\geq 0$. For which values of $p$ is the theory of commutative fields of characteristic $p$ inconsistent?

29. Prove that $\Gamma$ is consistent if and only if every finite subset of $\Gamma$ is consistent (see exercise 2).

30. Give the axioms of the theory of vector spaces as an extension of field theory.

31. Prove equations 4.1 and 4.2 of a definitional extension in the relational case (with $R$ unary, to simplify).

Bibliographic notes

The Curry–Howard correspondence was made explicit by Howard [66] and forms the framework of Girard et al.'s book [54]. The correspondence is implemented in higher-order systems: Girard's system $F$ [52, 54], and Martin-Löf's intuitionistic type theory [104].

Intuitionistic logic is studied by van Dalen in [154], and, more deeply, in the more recent book by Troelstra and van Dalen [150]. Natural deduction was analyzed by Prawitz in [125]. The sequent calculus and its links with computer science are treated in [47, 54].
Chapter 5

Models

5.1 The semantics of first-order logic

Interpretation schemes are given for each of the syntactic objects introduced in first-order logic: signatures, variables, terms, formulas and derivations.

5.1.1 $\Sigma$-algebras

Let $\Sigma$ be a signature, i.e., a set with an arity function $\text{ar} : \Sigma \to \mathbb{N}$.

Functional signatures
A symbol of arity $n$ is interpreted as a function of $n$ arguments.

Definition 5.1 A $\Sigma$-algebra $A$ consists of a set $A$, and for each $f \in \Sigma$ of arity $n$, of a mapping $f^A : A^n \to A$. $A = (A, (f^A)_{f \in \Sigma})$ is called an interpretation of $\Sigma$. The domain of interpretation $A$ is $A$.

In particular, a constant symbol is interpreted by an element of $A$.

Example A singleton $\{a\}$ is the domain of a trivial $\Sigma$-algebra, each constant symbol being interpreted by $a$ and the function symbols by constant functions of value $a$. A trivial interpretation equates everything. On the other hand, the free interpretations which follow equate nothing.

If $A$ and $B$ are $\Sigma$-algebras, a $\Sigma$-morphism $m : A \to B$ is a mapping from $A$ to $B$ such that

$$m(f^A(\bar{a})) = f^B(m(\bar{a}))$$

for each $f \in \Sigma$ of arity $n$ and each $n$-tuple $\bar{a}$ of elements of $A$. The category\footnote{Recall that a category is defined by a bunch of objects, and, for each pair $(A, B)$ of objects, of a set of morphisms $f : A \to B$, with a morphism composition operator (if $f : A \to B$ and} of $\Sigma$-algebras and $\Sigma$-morphisms is written $\text{Alg}_\Sigma$. Note that if $\Sigma \subseteq \Sigma'$, every $\Sigma'$-algebra
$A$ is automatically a $\Sigma$-algebra, written $A \upharpoonright \Sigma$, obtained by interpreting only the elements of $\Sigma$.

**Term algebras**

If $X$ is an arbitrary set of variables, the set $T_{\Sigma}[X]$ of terms has a canonical $\Sigma$-algebra structure $T$ defined by

$$f^T(t_1, \ldots, t_n) = ft_1 \ldots t_n$$

for $f \in \Sigma$, $ar(f) = n$, and $t_1, \ldots, t_n \in T_{\Sigma}[X]$.

This $\Sigma$-algebra is also a free object in the category-theoretic sense, as free monoids and free groups. For every $\Sigma$-algebra $A$ and every mapping $\xi : X \to A$, there exists a unique $\Sigma$-morphism $\hat{\xi}_\Sigma : T_{\Sigma}[X] \to A$ such that $\hat{\xi}_\Sigma(x) = \xi(x)$ for every $x \in X$:

![Diagram](https://via.placeholder.com/150)

$\hat{\xi}_\Sigma$ is defined by induction over terms:

$$\hat{\xi}_\Sigma(x) = \xi(x),$$

$$\hat{\xi}_\Sigma(ft_1 \ldots t_n) = f^A(\hat{\xi}_\Sigma(t_1), \ldots, \hat{\xi}_\Sigma(t_n)).$$

$T_{\Sigma}[X]$ is called the free $\Sigma$-algebra generated by $X$.

In particular, this property applies to $T_{\Sigma}$, the algebra of closed terms, when $X = \emptyset$. The above property is more easily written as: for every $\Sigma$-algebra $A$, there exists a unique $\Sigma$-morphism $i_A : T_{\Sigma} \to A$, defined by:

$$i_A(ft_1 \ldots t_n) = f^A(i_A(t_1), \ldots, i_A(t_n)).$$

Because of this property, $T_{\Sigma}$ is called the initial $\Sigma$-algebra.

Another important application of this property occurs when $A = T$. A unique endomorphism $\xi : T_{\Sigma}[X] \to T_{\Sigma}[X]$ extending $\xi : X \to T_{\Sigma}[X]$ is obtained. The domain of $\xi : X \to T_{\Sigma}[X]$, or of $\hat{\xi}$, is the set

$$\text{dom}(\xi) = \{ x \in X; \xi(x) \neq x \}.$$  

A previously defined concept reappears: if $\text{dom}(\xi)$ is finite, $\xi$ (or $\hat{\xi}$) is a substitution.

**Algebras in normal form**

Let $R$ be a confluent and normalizing rewriting system, i.e., every term has a unique normal form. Consider the set $N$ of normal terms for $R$, and write $t \downarrow \in N$ for the normal form of $t$. Let $f \in \Sigma$, $ar(f) = n \geq 0$. If $t_1, \ldots, t_n \in N$ let

$$f^N(t_1, \ldots, t_n) = (ft_1 \ldots t_n) \downarrow .$$

A $\Sigma$-algebra structure has been defined over the set $N$. 

$g : B \to C$, then $g \circ f : A \to C$ is defined) which is associative $(f \circ (g \circ h) = (f \circ g) \circ h$, for composable morphisms) and has a neutral element $1_A : A \to A$. See, for example, [5].
Logical connectives
Consider a signature of connectives, for example $\Sigma_c = \{ \land, \lor, \bot, \rightarrow \}$. The set $\text{Prop}[A]$ of propositions over $A$ is defined as the set $T_{\Sigma_c \cup \Delta}$. To give the Boolean meaning of the symbols in the signature, it is not an arbitrary $\Sigma_c$-algebra which must be used, but, rather, a Boolean algebra, and in particular the simplest among them, which is $2 = \{0, 1\}$. A semantics with two truth values will be given: true (for 1) and false (for 0). The operations min and max interpret $\land$ and $\lor$; $\bot$ is interpreted by 0; and $\rightarrow^2(a, b) = \max(1 - a, b)$.

Signatures of the predicate calculus
And now for first-order logic. Let $\Sigma = \Sigma_f \cup \Sigma_r$ be a predicate calculus signature. The relational symbols of the signature must be interpreted by relations. A $\Sigma$-algebra $A$ consists of a non-empty set $A$ along with the following:

- an element $c^A \in A$ for each constant symbol $c \in \Sigma$;
- a mapping $f^A : A^n \rightarrow A$ for each $n$-ary functional symbol $f \in \Sigma$;
- a relation $R^A : A^n \rightarrow \{0, 1\}$ (i.e., $R^A \subseteq A^n$) for each $n$-ary relational symbol $R \in \Sigma$.

The algebra $A = (A, c^A, \ldots, f^A, \ldots, R^A, \ldots)$ is called an interpretation of $\Sigma$. The set $A$ is its domain.

In the definition of a $\Sigma$-morphism between two $\Sigma$-algebras $A$ and $B$, the condition

$$m(R^A(\bar{a})) \subseteq R^B(m(\bar{a}))$$

is added for every $R \in \Sigma$ of arity $n$ and every $n$-tuple $\bar{a}$ of elements of $A$. The symbol $\text{Alg}_{\Sigma}$ continues to be used for the category of $\Sigma$-algebras.

Examples
For $\Sigma = \{0, s, +, -, \leq, =\}$, the set $\mathbb{N}$ with the usual $0, s : n \mapsto n + 1, +, -, \leq$ and $=$ is a $\Sigma$-algebra. But there are many others, since no property is imposed on $f^A$ or $R^A$.

If $\Sigma$ is the signature consisting of the binary relational symbol $R$, a $\Sigma$-algebra is a simple graph: its domain is the set of vertices.

Recall that if $\Sigma \subseteq \Sigma'$, i.e., $\Sigma_f \subseteq \Sigma'_f$ and $\Sigma_r \subseteq \Sigma'_r$, every $\Sigma'$-algebra $A$ is automatically a $\Sigma$-algebra written $A\upharpoonright\Sigma$, obtained by interpreting only the elements of $\Sigma$: it is the restriction of $A$ to $\Sigma$. A $\Sigma$-algebra $A$ whose restriction to $\Sigma_f$ is the initial $\Sigma_f$-algebra is called a Herbrand $\Sigma$-algebra. The set $T_{\Sigma}$ underlying all Herbrand $\Sigma$-algebras is sometimes called the Herbrand universe. Note that the relational symbols can be interpreted in any manner. A Herbrand algebra $\mathcal{H}$ is defined by specifying a value in 2 for each atom $Rt_1 \ldots t_n$ or by the set of atoms $Rt_1 \ldots t_n$ such that $R^H(t_1, \ldots, t_n) = 1$. Herbrand $\Sigma$-algebras can therefore be equated with subsets of the set $\mathcal{A}_\Sigma$ of closed terms, called the Herbrand base.
5.1.2 Terms

Variables are interpreted as ‘varying’ in a $\Sigma$-algebra. Write $A^X$ for the set of mappings from $X$ to $A$. The elements of $A^X$, written $\xi, \eta, \ldots$, are called valuations. In the semantics of programming languages, a valuation represents a memory state.

A term $t \in T_\Sigma[X]$ is interpreted (or denoted) by a mapping $t^A : A^X \to A$ defined by induction over terms by:

1. $x^A(\xi) = \xi(x)$, if $x \in X$;
2. $c^A(\xi) = c^A$, if $c \in \Sigma$ is a constant symbol;
3. $(f t_1 \ldots t_n)^A(\xi) = f^A(t_1^A(\xi), \ldots, t_n^A(\xi))$, if $f \in \Sigma$ is an $n$-ary function symbol.

If $\operatorname{var}(t) \subseteq \{x_1, \ldots, x_n\} \subseteq X$, $t^A(\xi)$ depends only on $\xi(x_1), \ldots, \xi(x_n)$. So the mapping can also be written $t^A : A^n \to A$, whose value in $(a_1, \ldots, a_n) \in A^n$ is obtained by ‘substituting’ $a_1, \ldots, a_n$ for $x_1, \ldots, x_n$ and $f^A$ for $f$, $f \in \Sigma$. In particular, a closed term $t \in T_\Sigma$ is interpreted by an element $t^A$ of $A$.

Rational expressions \ Let $\Gamma$ be an alphabet. Consider the signature $\Sigma$ having:

- the elements of $\Gamma$ along with the symbols $0, 1 \notin \Gamma$ as constants;
- $\ast$ (unary, as superscript), $\cdot$ (binary, also written by simple juxtaposition) and $+$ (binary) as functional symbols.

Consider the closed terms over $\Sigma$, called rational expressions over the alphabet $\Gamma$. For example, for $\Gamma = \{a, b, c\}$, $(c + ab)^*c$ is a closed term. Let $\mathcal{M}$ be the $\Sigma$-algebra $\mathcal{P}(\Gamma^*)$ where $\Gamma^*$ is the free monoid generated by $\Gamma$, with:

- $a^\mathcal{M} = \{a\}$, for $a \in \Gamma$;
- $0^\mathcal{M} = \emptyset$;
- $1^\mathcal{M} = \{\epsilon\}$, where $\epsilon$ is the empty string, the neutral element of $\Gamma^*$;
- $+$ is implemented in $\mathcal{M}$ by union and $\cdot$ by concatenation, i.e,

$\quad \quad LL' = \{uu'; u \in L, u' \in L'\}$;

- $\ast$ is implemented by Kleene closure, i.e.,

$\quad \quad L^* = \{u_1 \ldots u_n; n \geq 0, u_1, \ldots, u_n \in L\}$.

For $t = (c + ab)^*c$, $t^\mathcal{M} = \{c, cc, abc, ccc, cabc, ababc, \ldots\}$. The set $t^\mathcal{M}$ is called the rational language denoted by the rational expression $t$. The study of these languages forms one of the classical branches of formal language theory.
5.1.3 The propositional calculus

Since the language Prop[A] of propositions formed over the set A of atoms is a set of terms, the classical semantics of the propositional calculus is a particular case of the above. The standard Boolean interpretation 2 is imposed on the signature \( \Sigma_c \) of connectives. An interpretation of Prop[A] is a \( \Sigma_c \cup A \)-algebra whose restriction to \( \Sigma_c \) is the standard interpretation in 2. It is therefore completely defined by an element \( I \) of \( 2^A \), associating with each atom \( a \in A \) its ‘truth value’ \( I(a) \). This mapping \( I \) is extended to Prop[A] with,

\[
I(\varphi \lor \psi) = \max(I(\varphi), I(\psi)), \\
I(\varphi \land \psi) = \min(I(\varphi), I(\psi)), \\
I(\varphi \implies \psi) = \max(1 - I(\varphi), I(\psi)).
\]

If \( I(\varphi) = 1 \), then \( I \models \varphi \), read ‘\( I \) is a model of \( \varphi \).’

**Example** Let \( I(p) = 1 \) and \( I(q) = 0 \). Then \( I \models (p \land q) \implies p \), since

\[
I((p \land q) \implies p) = \max(1 - \min(I(p), I(q)), I(p)) \\
= \max(1 - \min(1, 0), 1) \\
= 1.
\]

If \( A \) has \( n \) elements, \( n \) finite, there are only \( 2^n \) interpretations of \( A \), so there exists an exhaustive algorithm to decide if a formula is a semantic tautology, i.e., valid for all interpretations. This algorithm is the well-known and inefficient truth table.

The chosen connectives have simple meanings, close to that of everyday language (although few people accept that ‘false implies false’ is ‘true’!). A \( k \)-ary connective \( c \) is interpreted in the Boolean algebra 2 as a mapping \( c^2 : 2^k \to 2 \). Nothing prevents a new \( k \)-ary connective from being defined simply by giving a function from \( 2^k \) to 2, without worrying about the meaning of the connective. It is not clear by looking at NK that there is a bound to the number of possible binary connectives. The semantics given here provides an exact bound: there are exactly \( 2^{2^k} \) \( k \)-ary connectives, in particular, 2 0-ary connectives (\( \top \) and \( \bot \)), 4 unary connectives (only \( \neg \) was used), and 16 binary connectives. An interesting question is, are there any other interesting connectives and do they have the same ‘Boolean’ characteristics that the usual connectors do? It was seen that some of the introduced connectives were sufficient to define all of them (\( \perp, \top, \neg, \land, \lor, \implies, \iff \)) in classical logic. This is the case for \( \lor \) and \( \land \) and \( \implies \), among others.

Let \( \Sigma_c \) be a fixed connective signature. A formula \( \varphi \in \text{Prop}[p_1, \ldots, p_k] \) has an interpretation \( I(\varphi) \in 2 \). Let \( \bar{\varphi}(I) = I(\varphi) \), thereby defining a mapping \( \bar{\varphi} : 2^k \to 2 \), hence a \( k \)-ary connective \( c \) such that \( c^2 = \bar{\varphi} \).
$\Sigma_c$ is called a connective basis if for every $n$ and every mapping $f : 2^n \to 2$, there exists a formula $\varphi$ such that $f = \tilde{\varphi}$ ($f$ is then said to be defined by $\varphi$).

**Proposition 5.1** $\Sigma_c = \{\neg, \land, \lor\}$ is a connective basis.

**Proof** For $n = 1$, there are six functions from 2 to 2, defined by the formulas $p$, $\neg p$, $p \lor \neg p$, $p \land \neg p$.

Let $n \geq 1$ and $f : 2^{n+1} \to 2$. Define $f_i(x_1, \ldots, x_n) = f(x_1, \ldots, x_n, i)$ for $i \in 2$.

By the inductive hypothesis, there exist formulas $\varphi_0$ and $\varphi_1$ such that $f_i = \tilde{\varphi}_i$. Define $\varphi = (p_{n+1} \land \varphi_1) \lor (\neg p_{n+1} \land \varphi_2)$. It is clear that $\tilde{\varphi} = f$, which proves that $\Sigma_c = \{\neg, \land, \lor\}$ is a connective basis. \qed

It is easily deduced that $\{\neg, \lor\}$ and $\{\neg, \land\}$ are also bases, by using Morgan's laws, whose algebraic expression is

$$
\begin{align*}
\min(x, y) &= 1 - \max(1 - x, 1 - y), \\
\max(x, y) &= 1 - \min(1 - x, 1 - y),
\end{align*}
$$

as well as $\{\bot, \Rightarrow\}$, since

$$\max(x, y) = \Rightarrow^2 (\Rightarrow^2 (x, \bot^2), y).$$

There are also bases with one element. Let $\text{nand}$ (also written $\mid$, Sheffer's bar) and $\text{nor}$ (also written $\dagger$, Peirce's dagger) be defined by the formulas $\neg(p_1 \land p_2)$ and $\neg(p_1 \lor p_2)$ respectively. The connectives are used in electronics to design logical gates: for example, $\text{nor}$ tests if its two entries are simultaneously null, which is easy to implement.

### 5.1.4 Formulas

Closed formulas are interpreted by truth values (0 or 1) in an analogous manner to the propositional calculus. Because of variables, valuations must be used, as was done for terms, to interpret formulas as relations.

A formula $\varphi \in L_\Sigma[X]$ is interpreted by a mapping $\varphi^A : A^X \to 2$, i.e., by a subset of $A^X$, whose definition by induction over formulas is given by Table 5.1.

In particular, for a proposition $\varphi \in L_\Sigma$, $\varphi^A$ is the constant 0 ("false") or 1 ("true") of 2. In a given interpretation, a formula is either true or false. Note that if $\varphi \in L_\Sigma[x_1, \ldots, x_n]$, $\varphi^A(\xi)$ depends only on $\xi(x_1), \ldots, \xi(x_n)$ and hence defines a mapping, also written $\varphi^A : A^n \to 2$.

Consider once again the extension $\hat{\xi}$ from $T_\Sigma[X]$ to itself of a valuation $\xi : X \to T_\Sigma[X]$. Write $\hat{\xi}$ also for the mapping $\hat{\xi} : T_\Sigma[X] \to T_\Sigma[X]$ defined for an atomic formula $Rt_1 \ldots t_n$ by

$$\hat{\xi}(Rt_1 \ldots t_n) = R\hat{\xi}(t_1) \ldots \hat{\xi}(t_n).$$

The extension of $\hat{\xi}$ to formulas without quantifiers is immediate. But to extend $\hat{\xi}$ from atoms to quantified formulas, the domain of $\xi$ must be finite, to allow the
Table 5.1 Interpretation of formulas

\[
\begin{align*}
(Rt_1 \ldots t_n)^A(\xi) &= R^A(t_1^A(\xi), \ldots, t_n^A(\xi)) \\
(\varphi \lor \psi)^A(\xi) &= \max(\varphi^A(\xi), \psi^A(\xi)) \\
(\varphi \land \psi)^A(\xi) &= \min(\varphi^A(\xi), \psi^A(\xi)) \\
(\neg \varphi)^A(\xi) &= 1 - \varphi^A(\xi) \\
(\exists x \varphi)^A(\xi) &= \sup_\eta \varphi^A(\eta) \quad \text{with } \eta(y) = \xi(y) \text{ for } y \neq x \\
(\forall x \varphi)^A(\xi) &= \inf_\eta \varphi^A(\eta) \quad \text{with } \eta(y) = \xi(y) \text{ for } y \neq x
\end{align*}
\]

renaming of the bound variables outside of dom(\xi). So \(\xi\) is restricted to being a substitution.

5.1.5 Derivations

The syntactic objects remaining to be interpreted are the derivations NK.

Let \(d \in D_E\) be a derivation with conclusion \(\varphi \in L_\Sigma[X]\): it is a (binding) tree with nodes labeled by formulas, and with root labeled by \(\varphi\). By substituting for these formulas their interpretations in a \(\Sigma\)-algebra \(A\), then for each \(\xi \in A^n\), by taking their values at \(\xi\), a tree \(d^A(\xi)\) labeled by truth values is obtained. The tree \(d^A(\xi)\) indicates the local behavior of \(d\) in \(A\):

\[
\frac{\varphi_1 \quad \psi_2}{\chi} \quad \Rightarrow \quad \frac{\varphi^A(\xi) \quad \psi^A(\xi)}{\chi^A(\xi)}
\]

Lemma 5.2 If the unbound leaves of \(d^A(\xi)\) are labeled by 1, then the conclusion is \(\varphi^A(\xi) = 1\).

Proof By induction over derivations, thirteen rules must be examined. Consider the case where the conclusion \(\varphi\) follows from rule \((E_\lor)\):

\[
(E_\lor) : \frac{d_0 \quad d_1[\chi] \quad d_2[\psi]}{\varphi \quad \varphi}
\]

where \(d_1\) and \(d_2\) are derivations of \(\varphi\) containing, respectively, \(\chi\) and \(\psi\) among their assumptions. The active assumptions of \(d\) are:

- those of \(d_0\);
- those of \(d_1\) without \(\chi\); and
- those of \(d_2\) without \(\psi\).
Suppose that the interpretations of the active assumptions of \( D \) all equal 1 at a given \( \xi \in A^X \). The inductive hypothesis applied to \( d_0 \) shows that \((\chi \lor \psi)^A(\xi) = 1\). By the definition of the interpretation of \( \lor \), \( \chi^A(\xi) = 1 \) or \( \psi^A(\xi) = 1 \). Look at the first case, \( \chi^A(\xi) = 1 \), and consider \( d_1 \). The active assumptions of \( d_1 \) also contain \( \chi \), and hence are all equal to 1 at \( \xi \). The inductive hypothesis can then be applied to \( d_1 \) to deduce its conclusion, which is also the conclusion of \( d \), that \( \varphi^A(\xi) = 1 \).

\[ \square \]

### 5.2 Models of a theory

Now that interpretation schemes for all syntactic objects have been adopted, it is time to introduce the notion of model and the semantic properties of soundness and completeness.

#### 5.2.1 The satisfiability relation \( \models \)

Let \( A \) be a \( \Sigma \)-algebra and \( \varphi \in L_\Sigma[X] \). The algebra \( A \) satisfies \( \varphi \), written \( A \models \varphi \), if \( \varphi^A(\xi) = 1 \) for all \( \xi \in A^X \), i.e., if the formula \( \varphi \) is ‘true in \( A \) by substituting any elements of \( A \) for the free variables’.

If \( A \models \varphi \) for every \( \Sigma \)-algebra \( A \), \( \varphi \) is valid or is a semantic tautology, written \( \models \varphi \). Since numerous tautologies can be written using \( \iff \), note that \( A \models \varphi \iff \psi \) if and only if \( \varphi^A = \psi^A \).

Let \( \Gamma \) be a theory, i.e., a subset of \( L_\Sigma[X] \). The algebra \( A \) is called a model of \( \Gamma \), written \( A \models \Gamma \), if \( A \models \varphi \) for every \( \varphi \in \Gamma \). Write \( \text{Mod}(\Gamma) \) for the ‘class’ of models of \( \Gamma \).

**Example** A model of the axioms of group theory is a group! See, however, the next section on equality.

A model of a theory which is also a Herbrand algebra is called a Herbrand model of that theory. The importance of these models is shown by Herbrand’s theorem and by its applications.

#### 5.2.2 Equality

Let \( \Sigma \) be a signature with equality, the symbol \( E \) standing for equality. A \( \Sigma \)-algebra with equality is a \( \Sigma \)-algebra \( A \) such that \( E^A \) is the equality of \( A \), i.e., the diagonal of \( A \times A \).

The equality axioms \( Ax_E \) were given in Chapter 3. A \( \Sigma \)-algebra with equality obviously satisfies \( Ax_E \), but not conversely. If \( A \models Ax_E \), then \( E^A \) is simply an equivalence relation over \( A \) (by \( Ax_E 1, 2, 3 \)), compatible with \( f^A \) and \( R^A \) (by the
following axioms). The quotient \( A' \) can be defined by \( A' = A/E^A \) and \( A' \) is then a \( \Sigma \)-algebra with equality, the projection \( \pi : A \rightarrow A' \) being a \( \Sigma \)-morphism. By induction over formulas, one can prove that for \( \varphi \in L_\Sigma[X] \), and \( \xi \in A^X \), the value \( \varphi^A(\xi) \) only depends on \( \pi \circ \xi \), hence that \( \varphi^A \) is the quotient of \( \varphi^A \) by \( E^A \).

\[
\begin{array}{ccc}
X & \xrightarrow{\xi} & A \\
& \searrow_{\xi'} & \downarrow \pi \\
& & A'
\end{array}
\]

The following proposition has therefore been demonstrated:

**Proposition 5.3** Let \( \Gamma \subseteq L_\Sigma[X] \) with equality. \( \Gamma \) has a model with equality if and only if \( \Gamma \cup Ax_E \) has a model.

**Convention** If \( \Sigma \) is with equality and \( \Gamma \subseteq L_\Sigma \), the phrase ‘model of \( \Gamma \)’ refers to a model with equality of \( \Gamma \).

### 5.2.3 Soundness of deduction

Let \( \Gamma \subseteq L_\Sigma[X] \) be the theory generated by \( \Delta \subseteq L_\Sigma[X] \),

\[
\Gamma = \{ \varphi \in L_\Sigma[X] ; \Delta \vdash \varphi \}.
\]

As \( \Delta \subseteq \Gamma \), it is clear that \( \text{Mod}(\Gamma) \subseteq \text{Mod}(\Delta) \), but is there equality?

**Lemma 5.4** Let \( \Delta \vdash \varphi \), \( A \) a \( \Sigma \)-algebra and \( \xi \in A^X \). If \( \psi^A(\xi) = 1 \) for every \( \psi \in \Delta \), then \( \varphi^A(\xi) = 1 \).

**Proof** It is a reformulation of lemma 5.2, since \( \Delta \vdash \varphi \) means that \( \Delta \) contains the set of active assumptions of a derivation of \( \varphi \). \( \square \)

Write \( \Delta \vdash \varphi \) if for every \( \Sigma \)-algebra \( A \) and for every \( \xi \in A^X \), \( \Delta^A(\xi) = 1 \) implies \( \varphi^A(\xi) = 1 \). It is a new relation over \( \mathcal{P}(L_\Sigma[X]) \times L_\Sigma[X] \) (like \( \vdash \)), called *semantic consequence*. In the particular case of closed formulas, i.e., if \( \Gamma \subseteq L_\Sigma \) and \( \varphi \in L_\Sigma \), the following definition is obtained: \( \Gamma \vdash \varphi \) if and only if for every \( \Sigma \)-algebra \( A \), \( A \vdash \Gamma \) implies \( A \vdash \varphi \). The lemma becomes the soundness theorem:

**Theorem 5.5 (Soundness)** If \( \Gamma \vdash \varphi \) then \( \Gamma \vdash \varphi \).

**Corollary 5.6** The models of a theory are those of its axioms.

For example, the models of group theory (and not only of its axioms) are the groups.
5.2.4 Completeness of deduction

The completeness theorem is the converse of the soundness theorem. With these two theorems, the \( \vdash \) and \( \models \) relations are identical, which gives two perspectives for the same problems. For example, the following theorem equates consistency (the deductive point of view) with the existence of a model (the semantic point of view): it will be used to prove the completeness theorem.

**Theorem 5.7 (Gödel completeness)** A theory has a model iff it is consistent.

**Proof**

Necessary condition: immediate since \( \varphi \) and \( \neg \varphi \) cannot have the same model.

Sufficient condition: this is the difficult part. An outline of L. Henkin's (1949 [60]) proof is given.

(1) An attempt is made to construct a 'syntactic model'. Suppose that \( \Sigma \) contains a constant, so that \( T \Sigma \neq \emptyset \). If \( \Gamma \subseteq L \Sigma[X] \), let \( \mathcal{H}_\Gamma \) be the canonical Herbrand \( \Sigma \)-algebra of \( \Gamma \), which interprets the relational symbols over domain \( T \Sigma \) by:

\[
R^{\mathcal{H}_\Gamma}(t_1, \ldots , t_n) = 1 \quad \text{if and only if} \quad \Gamma \vdash Rt_1 \ldots t_n.
\]

Hence, for a closed atomic formula \( \alpha \in At_\Sigma \),

\[
\mathcal{H}_\Gamma \models \alpha \quad \text{if and only if} \quad \Gamma \vdash \alpha.
\]

But it does not hold for an arbitrary formula, since

(a) there are in general not enough closed terms to satisfy them: if \( \Gamma \vdash \exists x \psi(x) \), there should exist \( t \in T \Sigma \) such that \( \psi^{\mathcal{H}_\Gamma}(t) = 1 \);

(b) there can exist \( \varphi \in L_\Sigma \) such that \( \Gamma \nvdash \varphi \) and \( \Gamma \nvdash \neg \varphi \) while, necessarily, \( \mathcal{H}_\Gamma \models \varphi \) or \( \mathcal{H}_\Gamma \models \neg \varphi \).

(2) Assume that \( \Gamma \) satisfies the following properties:

(a) Henkin property: if \( \Gamma \vdash \exists x \psi(x) \in L_\Sigma \), then there exists \( t \in T_\Sigma \) such that \( \Gamma \vdash \psi[x := t] \).

(b) Consistency and completeness: for every \( \varphi \in L_\Sigma \), \( \Gamma \vdash \varphi \) or else \( \Gamma \vdash \neg \varphi \).

The canonical \( \Sigma \)-algebra is the sought model: for every \( \varphi \in L_\Sigma \), \( \mathcal{H}_\Gamma \models \varphi \) iff \( \Gamma \vdash \varphi \). This is shown by induction over propositions. The case of atomic formulas is given by (1). There are three other cases: \( \neg \psi \), \( \varphi_1 \lor \varphi_2 \) and \( \exists x \psi \).

For negation, \( \mathcal{H}_\Gamma \models \neg \psi \) if and only if \( \mathcal{H}_\Gamma \models \neg \psi \) if and only if, by the inductive hypothesis, \( \Gamma \nvdash \psi \) if and only if, by assumption (b), \( \Gamma \vdash \neg \psi \).

If \( \mathcal{H}_\Gamma \models \varphi_1 \lor \varphi_2 \), then \( \mathcal{H}_\Gamma \models \varphi_1 \) or \( \mathcal{H}_\Gamma \models \varphi_2 \), hence by the inductive hypothesis, \( \Gamma \vdash \varphi_1 \) or \( \Gamma \vdash \varphi_2 \), where by \( \lor \) introduction, \( \Gamma \vdash \varphi_1 \lor \varphi_2 \). Conversely, if \( \mathcal{H}_\Gamma \nvdash \varphi_1 \lor \varphi_2 \), then \( \mathcal{H}_\Gamma \nvdash \varphi_1 \) and \( \mathcal{H}_\Gamma \nvdash \varphi_2 \), hence by the inductive hypothesis, \( \Gamma \nvdash \varphi_1 \) and \( \Gamma \nvdash \varphi_2 \). By completeness, it follows that \( \Gamma \vdash \neg \varphi_1 \) and
\( \Gamma \vdash \neg \varphi_2 \), hence by \( \land \)-introduction, \( \Gamma \vdash \neg \varphi_1 \land \neg \varphi_2 \), and by a Morgan law, \( \Gamma \vdash \neg (\varphi_1 \lor \varphi_2) \). By consistency, it follows that \( \Gamma \not\models \varphi_1 \lor \varphi_2 \).

Consider, finally, a proposition \( \varphi = \exists x \psi \). By the definition of \( \models \) and \( \mathcal{H}_\Gamma \), \( \mathcal{H}_\Gamma \models \varphi \) is the same as \( \psi^{\mathcal{H}_\Gamma}(t) = 1 \) for some \( t \in T_\Sigma \), i.e., \( \mathcal{H}_\Gamma \models \psi[x := t] \) for some \( t \in T_\Sigma \). By the inductive hypothesis, this is equivalent to \( \Gamma \vdash \psi[x := t] \) for some \( t \in T_\Sigma \). Hence if \( \mathcal{H}_\Gamma \models \varphi \), then by \( \exists \)-introduction, it follows that
\[
\Gamma \vdash \exists x \psi,
\]
\( i.e., \Gamma \vdash \varphi \). Conversely, if \( \Gamma \vdash \varphi \) by assumption (a) on \( \Gamma \), there exists \( t \in T_\Sigma \) such that \( \Gamma \vdash \psi[x := t] \), where by what has preceded, \( \mathcal{H}_\Gamma \models \varphi \).

(3) Suppose only that \( \Gamma \) is consistent. Then a construction of an extension \( \Gamma' \subseteq L_{\Sigma'}[X] \) of \( \Gamma \), \( \Sigma \subseteq \Sigma' \), satisfying conditions (a) and (b) of (2) is needed.

First, a \textit{conservative} extension is constructed: \( \Gamma \subseteq L_{\Sigma'}[X] \) is an extension of \( \Gamma \) satisfying (a) by 'iteratively' adding new constants \( c_\psi \) to the signature and axioms using these constants. Start with \( \Sigma_0 = \Sigma \) and \( \Gamma_0 = \Gamma \), and construct \( \Sigma_{n+1} \) from \( \Sigma_n \) by adding a constant symbol \( c_{3 \psi} \), and \( \Gamma_{n+1} \) by adding to \( \Gamma_n \) an axiom \( \exists \psi \Rightarrow \psi[x := c_3 \psi] \) for each proposition \( \exists \psi \) of \( L_{\Sigma_n} \). Let \( \Sigma' = \bigcup_{n \geq 0} \Sigma_n \) and \( \Gamma = \bigcup_{n \geq 0} \Gamma_n \).

All that is needed now is a consistent and complete extension \( \Gamma' \) of \( \Gamma \) over the same language \( L_{\Sigma'}[X] \). The existence (not the construction!) of this extension follows from the application of Zorn's lemma to the set of consistent extensions of \( \Gamma \). This set is non-empty, since \( \Gamma \) is consistent, and an increasing chain of consistent theories is consistent. There therefore exists a maximal element \( \Gamma' \) in the set. It is a consistent extension of \( \Gamma \), and if it were not complete, it could be extended in a consistent manner by adding a proposition \( \varphi \) such that \( \Gamma' \not\models \varphi \) and \( \Gamma' \not\models \neg \varphi \).

\( \Gamma \subseteq \Gamma' \) and \( \Gamma' \) satisfy (a), since \( \Gamma \subseteq \Gamma' \subseteq L_{\Sigma'}[X] \), and (b). So, \( \mathcal{H}_{\Gamma'} \) is a model of \( \Gamma' \), hence also of \( \Gamma \).

\[ \square \]

\textit{Theorem 5.8} If \( \Gamma \models \varphi \) then \( \Gamma \vdash \varphi \).

\textit{Proof} It is known that \( \Gamma \vdash \varphi \) if and only if \( \Gamma \cup \{ \neg \varphi \} \) is inconsistent. So if \( \Gamma \not\models \varphi \), \( \Gamma \cup \{ \neg \varphi \} \) is consistent and has a model \( \mathcal{A} \) such that \( \mathcal{A} \models \Gamma \) and \( \mathcal{A} \not\models \neg \varphi \), i.e., for every \( \xi \in A_X \), \( A^\mathcal{A}(\xi) = 1 \) and \( \varphi^\mathcal{A}(\xi) = 0 \), which proves \( \Gamma \not\models \varphi \).

\[ \square \]

\textit{Remark} This proof is non-constructive because it does not construct the derivation of \( \varphi \) from \( \Gamma \). Once the equivalence of \( \Gamma \models \varphi \) and \( \Gamma \vdash \varphi \) is assumed, two possible types of 'proof' are possible:

- deductive, by constructing \textit{one} derivation of \( \varphi \) with assumptions in \( \Gamma \);
- semantic, by showing that \( \varphi \) is true in \textit{all} the models of \( \Gamma \).

\[ \square \]

Deductive concepts such as the completeness of a theory can be studied semantically:
Example Field theory is consistent since $Q$ is a field—one only need know one model to claim its consistency. It is not complete since $1 + 1 = 0$ is false in $Q$ and $1 + 1 \neq 0$ is false in $Z/2Z$. Either one of these two propositions can be added to field theory to give a stronger theory (more theorems) which remains consistent.

Exercises 6 and 7 are examples of very simple syntactic conditions (on the form of axioms) leading to semantic properties: universal propositions and subalgebras, universal–existential propositions and increasing chains. The converses of these properties are also true, but more difficult to prove: they are the Loś–Tarski and Chang–Loś–Suszko theorems.

True formulas and provable formulas According to the soundness theorem, every formula provable in Peano arithmetic (PA) is true in $N$. Conversely, every formula true in all models of PA is provable in PA. But it is not sufficient that it be true in $N$. Although ‘standard’, $N$ is only one of many models of PA. It would seem reasonable that there be true formulas in $N$ which are not provable in PA, in the same manner that there are formulas (for example, $\forall xy(x + y = y + x)$) true in the group $(Z, +)$ and not provable in group theory. The existence of such formulas is a consequence of Gödel’s incompleteness theorem, proven in Chapter 7. There are, however, sufficiently simple formulas which are provable as soon as they are true: this is a property of closed atomic formulas, such as $s(0) + s(0) = s(s(0))$, propositions without quantifiers or with bounded quantifiers.

An arithmetic formula uses bounded quantifiers if all of its quantifiers are of the form $(\forall x < t)\varphi$ and $(\exists x < t)\varphi$, where $x \notin \text{var}(t)$. Recall that relative quantifiers are, respectively, abbreviations of $\forall x(x < t \implies \varphi)$ and $\exists x(x < t \land \varphi)$. These formulas describing properties of bounded, hence finite, sets are sufficiently simple that they are provable as soon as they are true, and they are even provable in elementary arithmetic (EA):

**Proposition 5.9** If $\varphi$ is a proposition with bounded quantifiers, then

1. if $N \models \varphi$, then $EA \models \varphi$;
2. if $N \not\models \varphi$, then $EA \models \neg \varphi$.

**Proof** By induction over propositions with bounded quantifiers.

The case of atoms $t_1 < t_2$ and $t_1 = t_2$ is left as an exercise.

If $N \models \psi \land \chi$, then $N \models \psi$ and $N \models \chi$, hence by the inductive hypothesis, $EA \models \psi$ and $EA \models \chi$, from which $EA \models \psi \land \chi$. If $N \not\models \psi \land \chi$, then $N \not\models \psi$ or $N \not\models \chi$, hence by the inductive hypothesis, $EA \models \neg \psi$ or $EA \models \neg \chi$, from which $EA \models \neg \psi \lor \neg \chi$, and by a Morgan axiom, $EA \models \neg (\psi \land \chi)$.

Let $t$ be a closed term $m = t^N$. If $N \models (\exists x < t)\psi$, then $N \models \psi[x := s^t(0)]$ for an $i$, $0 < i < m$, hence by the inductive hypothesis, $EA \models \psi[x := s^t(0)]$. Now, $EA \models s^t(0) < s^m(0)$ and $EA \models t = s^m(0)$, so by the axioms of equality, $EA \models s^t(0) < t \land \psi[x := s^t(0)]$, and by $\exists$-introduction, $EA \models \exists x(x < t \land \psi(x))$. 


If $\mathbb{N} \not\models (\exists x < t)\psi$, then $\mathbb{N} \not\models \psi[x := s^i(0)]$ for every $i$, $0 < i < m$, hence by the inductive hypothesis, $EA \vdash \neg \psi[x := s^i(0)]$ for each $i < m$. Now, $EA \vdash t = s^m(0)$ and $EA \vdash x < s^m(0) \iff x = 0 \lor \ldots \lor x = s^{m-1}(0)$. So, by $\lor$-elimination and by the equality axioms, $EA \vdash \forall x (x < t \Rightarrow \neg \psi(x))$, hence by the Morgan axioms, $EA \vdash \neg \exists x (x < t \land \psi(x))$.

The other cases are treated in an analogous manner.

5.2.5 A few applications of model theory

Model theory has undergone substantial development. Some results are presented, particularly consequences of the compactness theorem, which give an overview of semantic phenomena. The essentially syntactic proof of the compactness theorem assumes the completeness theorem.

Theorem 5.10 (Compactness) A theory $\Gamma$ has a model if and only if every finite subset of $\Gamma$ has a model.

Proof Follows from the theorem of the existence of models and from the finitary character of derivations: if $\Gamma \vdash \bot$, there exists a finite subset $\Gamma'$ of $\Gamma$ such that $\Gamma' \vdash \bot$.

Exercises 10 and 11 are classical applications of this result to field theory and to the existence of non-standard integers. In this manner one can show the existence of models, e.g., of arithmetic, of arbitrarily large cardinality. In the other direction, the following proposition shows that the consistent theories used here always have countable models. For example, set theory has a countable model, which seems paradoxical, since there are an uncountable number of sets.

Proposition 5.11 If $\Gamma \subseteq L_\Sigma[X]$, $\Sigma$ at most countable, then $\Gamma$ has a countable model.

Proof Recall the proof of the model existence theorem. If $\Sigma$ is countable then $T_\Sigma$ is countable (easy). But, the constructed model $\mathcal{H}_\Sigma$ has as domain $T_\Sigma$. It suffices to prove that $\Sigma'$ is countable, which follows from the manner in which it was constructed: starting from $\Sigma$, a countable number of constants is added (since $L_\Sigma[X]$ is countable) iteratively.

This proposition is a strengthening of Gödel’s theorem. There are other results on the cardinality of models, by Skolem–Löwenheim, Vaught, and others. Exercise 12 is one of those results.

A class $\mathcal{E}$ of $\Sigma$-algebras is called axiomatizable, (respectively finitely axiomatizable) or elementary if there exists a subset (respectively a finite subset) $\Gamma$ of $L_\Sigma[X]$ such that $\mathcal{E} = \text{Mod}(\Gamma)$.

An important concept in model theory is that of elementary equivalence: two $\Sigma$-algebras $\mathcal{A}$, $\mathcal{B}$ are elementarily equivalent, written $\mathcal{A} \equiv \mathcal{B}$, if for every proposition
$\varphi \in L_\mathcal{E}$, $\mathcal{A} \models \varphi$ if and only if $\mathcal{B} \models \varphi$. This concept formalizes the intuitive idea of indistinguishability under observations: here, an observation is the value of an interpretation of a proposition. Exercise 15 allows this concept to be linked to the completeness of a theory.

A finer concept of elementary equivalence is that of elementary morphism or of elementary subalgebra. A morphism $m : \mathcal{A} \rightarrow \mathcal{B}$ of $\Sigma$-algebras is elementary if for every formula $\varphi$ and every valuation $\xi \in A^X$, $\varphi^\mathcal{A}(\xi) = \varphi^\mathcal{B}(m \circ \xi)$. If $m$ is inclusion, $\mathcal{A}$ is called an elementary subalgebra of $\mathcal{B}$, characterized by exercise 16: a formula with parameters in $\mathcal{A}$, satisfiable in $\mathcal{B}$ by an element of $\mathcal{B}$, is also satisfiable by an element of $\mathcal{A}$.

5.2.6 Kripke interpretations

The semantics of classical logic is very simple: it reduces to an algebraic computation in a finite Boolean algebra. The semantics of intuitionistic logic is much more complex. Only the propositional case will be considered here. From a constructive point of view, knowing that a formula is true cannot be reduced to the result of the computation of a truth value: a result is never an explanation. But if the sequentaility of the computation can be controlled, then a constructive aspect can be given.

It is supposed that the computation of the truth value of a formula $\varphi$ is done in several steps $u, v, \ldots$. Let $U$ be the (finite) set of these steps, called a frame. A notation is introduced to say that at step $u$, a formula $\varphi$ has been verified. Let $I(p)$ be the finite set of steps where the atomic formula $p$ is verified. This function $p \mapsto I(p)$ is called a Kripke interpretation. A relation $I, u \models \varphi$ is defined stating that in the interpretation $I$, at step $u$, the formula $\varphi$ is verified. If $I$ is fixed, this relation is abbreviated to $u \models \varphi$ (u forces $\varphi$).

To take account of the sequential character of the verification process, one must suppose that $U$ is ordered, although not necessarily (totally) by time, since certain steps can be given to separate 'verifiers'. A natural condition is the monotonicity of the process: a verified fact is not put into question by a subsequent step, as is denoted at the atomic formula level by

$$\text{if } u \in I(p) \text{ and } u \preceq v, \text{ then } v \in I(p).$$

In the inductive relation which follows from relation $\models$, the cases of conjunction, disjunction and false are analogous to classical logic. However, extending the monotonicity condition to all formulas imposes a particular form on implication. The semantics requires that modus ponens be allowed at each step: if $u \models \varphi \Rightarrow \psi$ and $v \geq u$, then by monotonicity $v \models \varphi \Rightarrow \psi$, and if $v \models \varphi$, by 'local modus ponens at $v$', $v \models \psi$ is obtained. The following definition of $\models$ satisfies this condition.

1. $u \models p$, if $p$ is an atom and $u \in I(p)$;
2. \( u \models \varphi \land \psi \) if \( u \models \varphi \) and \( u \models \psi \);
3. \( u \models \varphi \lor \psi \) if \( u \models \varphi \) or \( u \models \psi \);
4. \( u \models \varphi \Rightarrow \psi \) if for every \( v \geq u \), if \( v \models \varphi \) then \( v \models \psi \);
5. \( u \models \bot \) is not true for any \( u \).

Since \( \neg \varphi \) is an abbreviation for \( \varphi \Rightarrow \bot \), it follows that

\( (5') \quad u \models \neg \varphi \) if \( v \models \varphi \) for no \( v \geq u \).

Interpretation \( I \) can be extended to all formulas by

\[
I(\varphi) = \{ u \in U; I, u \models \varphi \}.
\]

\( I \) satisfies \( \varphi \), written \( I \models \varphi \), if \( I, u \models \varphi \) for each step \( u \in U \), i.e., if \( I(\varphi) = U \). Formulas \( \varphi \) is true in frame \( U \), written \( U \models \varphi \), if all the interpretations with values in \( U \) satisfy \( \varphi \). Finally, \( \varphi \) is intuitionistically valid (\( \models \varphi \)) if \( U \models \varphi \) for every frame \( U \). Soundness and completeness theorems can be proven, as for the classical case.

It is easy to disprove a formula by finding a Kripke interpretation which does not satisfy it.

**Example** Let \( \varphi \equiv p \lor \neg p \), and consider the interpretation \( I \) defined by \( U = \{ a, b \} \), \( a \leq b \), \( I(p) = \{ b \} \). At step \( a \), \( p \) is not verified since \( a \notin I(p) \), nor is \( \neg p \) since \( p \) is verified in \( b \geq a \).

In classical logic, the interpretation \( I(\varphi) \) of a formula is an element of the Boolean algebra \( \mathcal{P}(U) \). Here \( I(\varphi) \) is a subset of \( U \).

One might think that the interpretation \( I \) has values in the Boolean algebra \( \mathcal{P}(U) \). However, \( I(\varphi) \) is always an open subset of \( U \), i.e., if \( u \in I(\varphi) \) and \( u \leq v \), then \( v \in I(\varphi) \) (the open denomination effectively corresponds to open subsets of a topological space). Let \( \mathcal{O}(U) \) be the set of open subsets. A Kripke interpretation is therefore a function with values in \( \mathcal{O}(U) \), which while not being a Boolean algebra, does have an interesting algebraic structure, called Heyting algebra. In general, the set of open sets of a topological space \( X \) is an example of a Heyting algebra. The complement of an open set not being open in general, its interior must be taken: define \( \sim G = (X \setminus G)° \). Negation is interpreted by this operation and \( \sim \sim G = G \) is not always true (only \( G \subseteq \sim \sim G \) is guaranteed), which illustrates the behavior of intuitionistic negation. The role of a semantics is to offer intuitive views, as is done here.

Kripke interpretations are close to those defining the semantics of modal, temporal and epistemic logics, where the set \( U \) of steps is interpreted by a set of worlds, instants or knowledge states. These logics are used to study certain properties of the execution of sequential and concurrent programs such as termination, liveness and fairness. Artificial intelligence also uses modal logics.
5.3 Herbrand’s theorem

Herbrand’s theorem essentially consists of doing semantics using syntactic means. It is therefore the semantic technique most suited to computer science. However, since syntactic means are more restricted than general interpretations, semantics will be given through ‘approximations’: a formula of the predicate calculus will be approximated by its expansions, which are formulas of the propositional calculus.

5.3.1 Formula expansions

The problem is how to treat quantifiers. In semantics, they are interpreted by computing the least or greatest upper bound over an infinite set of elements, not a computable operation. For example, Fermat’s theorem has the truth value, in N,

\[
\inf_{n \geq 3} \inf_{x,y,z > 0} (\text{if } x^n + y^n = z^n \text{ then } 0 \text{ else } 1),
\]

and no one knows how to compute the value. When the domain of an interpretation is finite, a universal quantifier (respectively existential) behaves as a conjunction (respectively disjunction). The quantifiers are ‘interpreted’ internally, i.e., by only working at the syntactic level of the formula, replacing the quantifiers with the connectives \( \land \) and \( \lor \).

Let \( D \) be a finite subset of the set \( T_\Sigma \) of closed terms. To the formula \( \varphi \in L_\Sigma[\mathcal{X}] \) is associated its expansion under \( D \), \( \text{Exp}(\varphi, D) \), which is the formula defined by induction in Table 5.2.

| \text{Exp}(\text{R}t_1 \ldots t_n, D) | = | \text{R}t_1 \ldots t_n |
| \text{Exp}(\varphi_1 \land \varphi_2, D) | = | \text{Exp}(\varphi_1, D) \land \text{Exp}(\varphi_2, D) |
| \text{Exp}(\varphi_1 \lor \varphi_2, D) | = | \text{Exp}(\varphi_1, D) \lor \text{Exp}(\varphi_2, D) |
| \text{Exp}(\varphi_1 \Rightarrow \varphi_2, D) | = | \text{Exp}(\varphi_1, D) \Rightarrow \text{Exp}(\varphi_2, D) |
| \text{Exp}(\forall x \varphi, D) | = | \land_{d \in D} \text{Exp}(\varphi[x := d], D) |
| \text{Exp}(\exists x \varphi, D) | = | \lor_{d \in D} \text{Exp}(\varphi[x := d], D) |
| \text{Exp}(\neg \psi, D) | = | \neg \text{Exp}(\psi, D) |

Table 5.2 The expansion under \( D \), \( \text{Exp}(\varphi, D) \)

It follows immediately that \( \text{Exp}(\varphi, D) \) is an open (quantifier-free) formula having the same free variables as \( \varphi \).
Example  Let $\varphi \equiv \forall x (x > 0 \Rightarrow \exists y (x = sy))$ and $D = \{0, ss0\}$. Then

$$\operatorname{Exp}(\varphi, D) = \quad (0 > 0 \Rightarrow (0 = s0) \lor (0 = ss0))$$
$$\quad \land (ss0 > 0 \Rightarrow (ss0 = s0) \lor (ss0 = sss0)).$$

What is the relationship between a formula and its expansions? Consider the simple case of an atom with one quantified variable. By applying rules $E_\forall$ and $I_\land$ in the universal case and rules $I_3$ and $E_\forall$ in the existential case, the result is

$$\forall x \, Rx \vdash Ra_1 \land \ldots \land Ra_n,$$
$$Ra_1 \lor \ldots \lor Ra_n \vdash \exists x \, Rx.$$

Hence, an expansion is an ‘approximation from above’ in the first case, ‘from below’ in the second. It depends on the quantifier. In the case of an arbitrary formula, the place of quantifiers must be taken into account, and an analogous result is achieved only when the quantifiers all have the same force. A formula is called *universal* if all of its occurrences of $\forall$ are positive and all of its occurrences of $\exists$ are negative. Exchanging ‘positive’ and ‘negative’ gives the definitions of *existential* formulas.

In the preceding example, $\varphi$ is neither universal nor existential, since the occurrences of $\forall$ and of $\exists$ are both positive. A *prenex* formula is universal (respectively existential) if all of its quantifiers are $\forall$ (respectively $\exists$). The transformation of a universal formula into an equivalent prenex formula is immediate: it suffices to place all the quantifiers at the front by replacing the $\exists$, necessarily negative, by $\forall$. One can then continue to ‘normalize’ the prenex form, for example by computing its conjunctive normal form.

Lemma 5.12 If $\varphi$ is a universal (respectively existential) formula, then for all $D$,

$$\vdash \varphi \Rightarrow \operatorname{Exp}(\varphi, D) \quad \text{(respectively} \quad \vdash \operatorname{Exp}(\varphi, D) \Rightarrow \varphi).$$

Rather than give a proof, consider the computation of the expansion and its behavior. Represent $\varphi$ by a tree whose leaves are atoms and nodes are the connectives or quantifiers. Each node defines a subformula of $\varphi$. The expansion of $\varphi$ is done in an ascending manner, i.e., by successively computing the expansions of its subformulas and by following the ‘direction’ of the implication between a formula and its expansion. The leaves hold atoms $\alpha$ and $\vdash \alpha \iff \operatorname{Exp}(\alpha, D) = \alpha$. When passing through a $\neg$ node, the direction of the implication is reversed. When passing through a $\land$ or $\lor$ binary node, the direction is preserved if it is the same for the two children. For an $\Rightarrow$ node the direction of the right-hand node is preserved if the directions of the children are different. When passing through $\forall$ (respectively $\exists$), the direction is preserved, if the direction is $\Rightarrow$ (respectively $\Leftarrow$). In the other cases there is no ascending transmission of relations between subformulas and their expansions. As for the sign of the subformula, it is transmitted from the
root, where it is 0, towards the leaves (changing along the way on a \( \rightarrow \) or to the left of a \( \Rightarrow \)).

If \( \varphi \) is universal, the ascending propagation is possible and satisfies the invariant: the direction of implication is \( \Rightarrow \) (respectively \( \leftarrow \)) for a positive (respectively negative) subformula. It follows that at the root, which is a positive subformula, 
\[ \vdash \varphi \Rightarrow \exp(\varphi, D). \]

In the terminology of attribute grammars, the sign of a formula is an inherited attribute while its expansion is a synthesized attribute.

To conclude, in the case of a universal or existential formula, its expansion is a syntactically simpler approximation for \( \vdash \), from above or below.

### 5.3.2 Herbrand's theorem

In the following discussion, only closed formulas will be considered. If \( \varphi \) is closed, the expansion \( \exp(\varphi, D) \) is both open and closed, i.e., a formula of the propositional calculus over the set \( \text{At}_B \) of closed atoms. The expansion \( \exp(\varphi, D) \) has a value in 2 when the closed terms are interpreted in 2, which means considering a Herbrand \( \Sigma \)-algebra. Recall that a Herbrand algebra is a subset \( B \) of \( \text{At}_\Sigma \) composed of the atoms of value 1 in this interpretation.

Let \( A \) be a \( \Sigma \)-algebra. Associate the Herbrand \( \Sigma \)-algebra \( \mathcal{H}_A \) defined by the set of closed atoms \( \alpha \) such that \( A \models \alpha \). Write \( \text{Th}_\varphi(A) \) for the set of universal closed formulas \( \varphi \) such that \( A \models \varphi \).

**Proposition 5.13**

1. The Herbrand algebra \( \mathcal{H}_A \) is a model of theory \( \text{Th}_\varphi(A) \).
2. A universal theory has a model if and only if it has a Herbrand model.

**Proof**

(1) The Herbrand interpretation defined for the closed atoms can be extended to all the open formulas by

\[ (\varphi^o)^{\mathcal{H}_A}(\eta) = (\varphi^o)^A(i_A \circ \eta), \]

where \( i_A \) is the unique (initial) \( \Sigma_f \)-morphism from \( \mathcal{H}_A \) to \( A \).

Let \( \varphi \in \text{Th}_\varphi(A) \), assumed prenex to simplify: \( \varphi = \forall \exists \varphi^o \), where \( \varphi^o \) is open. By assumption, \( \varphi^A = 1 \). Compute \( \varphi^{\mathcal{H}_A} \):

\[
\begin{align*}
\varphi^{\mathcal{H}_A} &= \inf_{\eta \in \text{Th}_\varphi^A} (\varphi^o)^{\mathcal{H}_A}(\eta) \\
&= \inf_{\eta \in \text{Th}_\varphi^A} (\varphi^o)^A(i_A \circ \eta) \\
&\geq \inf_{\xi \in A^A \mathcal{H}_A} (\varphi^o)^A(\xi) \\
&= 1,
\end{align*}
\]
i.e, $\mathcal{H}_A \models \varphi$.

(2) Let $\Gamma$ be a theory formed of universal propositions. If $\Gamma$ has a model $A$, then $\mathcal{H}_A$ is a model of $\Gamma$, since $\mathcal{H}_A \models Th_\mathcal{V}(A)$ and $\Gamma \subseteq Th_\mathcal{V}(A)$. The converse is trivial. □

**Theorem 5.14 (Herbrand)** A universal proposition $\varphi$ is unsatisfiable if and only if there exists an expansion $\text{Exp}(\varphi, D)$ which is unsatisfiable.

**Proof** To simplify the proof, suppose that $\varphi$ is prenex, $\varphi = \forall x \varphi^o$. In that case,

$$\text{Exp}(\varphi, D) = \bigwedge_{\theta : \text{var}(\varphi^o) \rightarrow D} \theta(\varphi^o)$$

is a conjunction of propositions taken from the set

$$\text{Exp}(\varphi) = \{\theta(\varphi^o); \theta : \text{var}(\varphi^o) \rightarrow T_\Sigma\}$$

of all the closed instances of $\varphi^o$.

Suppose that every expansion is satisfiable. The theory $\text{Exp}(\varphi)$ will be shown to be satisfiable. In fact, let $E = \{\theta_1(\varphi^o), \ldots, \theta_n(\varphi^o)\}$ be an arbitrary finite subset of $\text{Exp}(\varphi)$ and define $D = \bigcup_{i=1}^n \text{Im}(\theta_i)$, so that the $\theta_i$ take values in $D$. By assumption, $\text{Exp}(\varphi, D)$ has a model, which satisfies in particular each of the elements of $E$. Therefore, every finite subset of $\text{Exp}(\varphi)$ is satisfiable, which implies, by the compactness theorem, that $\text{Exp}(\varphi)$ is satisfiable. But it is a universal theory, which therefore has a Herbrand model, $\mathcal{H}$, from the preceding proposition. It remains to be shown that $\mathcal{H}$ is a model of $\varphi$, which follows from the following computation:

$$\varphi^\mathcal{H} = \inf_{\eta : \text{var}(\varphi^o) \rightarrow T_\Sigma} (\varphi^o)^\mathcal{H}(\eta)$$

$$= \inf_{\eta : \text{var}(\varphi^o) \rightarrow T_\Sigma} (\eta(\varphi^o))^\mathcal{H}$$

$$= \inf_{\psi \in \text{Exp}(\varphi)} \psi^\mathcal{H}$$

$$= 1.$$

Conversely, $\varphi$ being universal, $\vdash \varphi \Rightarrow \text{Exp}(\varphi, D)$, so every model of $\varphi$ is a model of $\text{Exp}(\varphi, D)$. □

The 'dual' version of the Herbrand theorem, for an existential proposition $\varphi$, can be deduced from the universal case (exercise 19):

1. $\varphi$ is valid if and only if $\varphi$ is true in every Herbrand algebra;
2. $\varphi$ is valid if and only if there exists a valid expansion of $\varphi$, i.e., for $\varphi = \exists x \varphi^o$, there exist substitutions $\theta_i : \text{var} \varphi^o \rightarrow T_\Sigma$, $i = 1, \ldots, n$, such that $\bigvee_i \theta_i(\varphi^o)$ is a tautology of the propositional calculus.
The real Herbrand theorem is in fact a much stronger result: it states that the closed substitutions \( \theta_i \) can be effectively computed from a derivation of \( \varphi \) and that, conversely, a derivation of \( \varphi \) can be effectively constructed from a derivation of \( \forall_i \theta_i(\varphi^o) \), and that these constructions have a 'limited' complexity. The substitutions \( \theta_i \) constitute a supplementary information on the truth of a formula: the Herbrand theorem allows the transformation of a validity proof into a 'computation' of the closed terms. A systematic implementation of this idea is used in logic programming.

### 5.3.3 Skolemization

Since the Herbrand theorem cannot be applied to an arbitrary formula, the formulas will be translated by skolemization before applying the theorem. Properties of the transformation, presented in Chapter 3, will be proven here.

To simplify the following proof, only prenex formulas will be used. Let \( \psi \in \mathcal{L}_\Sigma[x_1, \ldots, x_k, x_{k+1}] \) and \( \varphi \equiv \forall x_1 \ldots \forall x_k \exists x_{k+1} \psi \in \mathcal{L}_\Sigma \).

\[
\tilde{\Sigma} = \Sigma \cup \{F\},
\]
\[
\tilde{\varphi} = \forall x_1 \ldots \forall x_k \exists x_{k+1} \psi[x_{k+1} := Fx_1 \ldots x_k] \in \mathcal{L}_{\tilde{\Sigma}}
\]

were defined, \( F \) a new \( k \)-ary function symbol.

The following lemma will be used to prove the Herbrand–Skolem theorem (next page).

**Lemma 5.15**

1. \( \models \tilde{\varphi} \Rightarrow \varphi \).
2. Every \( \Sigma \)-algebra \( \mathcal{A} \) can be extended to a \( \tilde{\Sigma} \)-algebra \( \tilde{\mathcal{A}} \) such that \( \varphi^{\tilde{\mathcal{A}}} = \varphi^\mathcal{A} \).
3. \( \varphi \) is satisfiable if and only if \( \tilde{\varphi} \) is satisfiable.

**Proof**

(1) It suffices to prove that for every \( \tilde{\Sigma} \)-algebra \( \mathcal{A} \), \( \tilde{\varphi}^\mathcal{A} \leq \varphi^\mathcal{A} \) under order \( 0 \leq 1 \) over \( \{0, 1\} \):

\[
\tilde{\varphi}^\mathcal{A} = \inf_{a_1, \ldots, a_k} \psi^\mathcal{A}(a_1, \ldots, a_k, F^\mathcal{A}(a_1, \ldots, a_k)) \\
\leq \inf_{a_1, \ldots, a_k} \sup_{a_{k+1}} \psi^\mathcal{A}(a_1, \ldots, a_k, a_{k+1}) \\
= \varphi^\mathcal{A}.
\]

(2) Let \( \mathcal{A} \) be a \( \Sigma \)-algebra. If \( \varphi^\mathcal{A} = 0 \), then for every extension \( \mathcal{B} \) of \( \mathcal{A} \), \( \tilde{\varphi}^\mathcal{B} = 0 \), from (1). If \( \varphi^\mathcal{A} = 1 \), it is because for every \( a_1, \ldots, a_k \in \mathcal{A} \), there exists \( a_{k+1} \in \mathcal{A} \) such that \( \psi^\mathcal{A}(a_1, \ldots, a_k, a_{k+1}) = 1 \). Define \( F^\mathcal{A}(a_1, \ldots, a_k) = a_{k+1} \) and let \( \tilde{\mathcal{A}} \) be the extension of \( \mathcal{A} \) so defined. Then \( \tilde{\varphi}^\mathcal{A} = 1 \).
(3) If \( \varphi \) is satisfiable, so is \( \bar{\varphi} \), from (2). Conversely, every model of \( \bar{\varphi} \) is a model of \( \varphi \), from (1).

\[ \square \]

**Theorem 5.16** Let \( \varphi \) be a proposition, \( \varphi^u \) (respectively \( \varphi^e \)) the universal (respectively existential) skolemized form of \( \varphi \). Then

1. \( \vdash \varphi^u \Rightarrow \varphi \) and \( \vdash \varphi \Rightarrow \varphi^e \);
2. \( \varphi \) is satisfiable if and only if \( \varphi^u \) is satisfiable;
3. \( \varphi \) is sound if and only if \( \varphi^e \) is sound;
4. \( \varphi \) is unsatisfiable if and only if there exists an unsatisfiable expansion of \( \varphi^u \);
5. \( \varphi \) is sound if and only if there exists a sound expansion of \( \varphi^e \).

#### 5.3.4 Problems in automatic theorem-proving

Automatic theorem-proving is an applied area of computer science, in which algorithms from logic are used to solve problems involving deduction.

It has been shown that the statements ‘\( \Gamma \vdash \varphi \)’, ‘\( \Gamma \vDash \varphi \)’, ‘\( \Gamma \cup \{\neg \varphi\} \) inconsistent’ and ‘\( \Gamma \cup \{\neg \varphi\} \) unsatisfiable’ are equivalent, as are those when \( \Gamma \) is finite: ‘\( \vdash \varphi \)’, ‘\( \vDash \varphi \)’, ‘\( \neg \varphi \) inconsistent’ and ‘\( \neg \varphi \) unsatisfiable’. For these kinds of problems, of validity or unsatisfiability, typical methods are oriented towards proving inconsistency, i.e., work towards the negation \( \neg \varphi \): \( \bot \) is derived from \( \neg \varphi \). The problem of satisfiability, ‘does \( \Gamma \) have a model?’, is the ‘dual’ of the preceding one.

The essential step is skolemization. Equivalence-preserving algebraic transformations are then made to facilitate deductive methods of refutation, in particular the so-called **resolution**. To a proposition \( \varphi \) is associated a set \( S(\varphi) \) of clauses with the property that \( \varphi \) is satisfiable if and only if \( S(\varphi) \) is. Herbrand’s theorem is applied to \( S(\varphi) \). A **clause** of a first-order language is a universally quantified disjunction of literals.

1. For every \( \varphi \in L_\Sigma \), a universal \( \varphi^u \) can be constructed such that \( \varphi \) is satisfiable if and only if \( \varphi^u \) is satisfiable.
2. \( \varphi^u \) is immediately transformed into an equivalent prenex formula \( \forall \vec{x} \varphi^o \) by moving all the quantifiers to the head, the \( \exists \) being replaced by \( \forall \).
3. For every open formula \( \varphi^o \), a conjunctive formula \( \varphi^c \) can be constructed such that \( \vdash \varphi^o \iff \varphi^c \).
4. If \( \varphi^c = \varphi_1 \wedge \ldots \wedge \varphi_m \), let

\[ S(\varphi) = \{ \forall \vec{x} \varphi_1, \ldots, \forall \vec{x} \varphi_m \}. \]
$S(\varphi)$ is a set of clauses, and

$$\vdash \varphi^u \iff \bigwedge_{C \in S(\varphi)} C.$$  

Careful! The first transformation only preserves satisfiability, while the last three produce a formula equivalent to $\varphi^u$.

**Proposition 5.17** Let $\varphi$ be a proposition. Then:

1. $S(\varphi) \vdash \varphi$.
2. If $\varphi$ is satisfiable, $S(\varphi)$ is as well.

Herbrand’s theorem is the basis for unsatisfiability proofs: test the $\text{Exp}(\varphi^u, D)$ for all the possible $D \subseteq T_\Sigma$. It suffices to construct an increasing sequence of finite subsets $D_n$ whose union is $T_\Sigma$ to obtain a semi-decision procedure, called Herbrand’s procedure, for the unsatisfiability problem of $\varphi$, since for every $n$ there is a decision procedure for the unsatisfiability of $\text{Exp}(\varphi^u, D_n)$, for example using truth tables. Let

\[
\begin{align*}
D_0 &= \Sigma_0, \\
D_{n+1} &= \{ f t_1 \ldots t_p; f \in \Sigma_p, t_1, \ldots, t_p \in D_n \} \\
S(\varphi) &= \{ C_1, \ldots, C_m \}, \\
u_p &= \text{card } \Sigma_p, \quad 0 \leq p \leq l, \\
u_i &= \text{card } \text{var}(C_i), \quad 1 \leq i \leq m, 
\end{align*}
\]

where $\Sigma_p$ is the set of $p$-ary functional symbols of $\Sigma$ and $l$ is the maximum arity of symbols in $\Sigma$. For $n \geq 0$,

\[
\begin{align*}
d_n &= \text{card } D_n, \\
h_n &= \text{card } \text{Exp}(\varphi^u, D_n),
\end{align*}
\]

must be evaluated, where this expansion is seen as a set of $m$-clauses. Then (see exercise 22)

\[
\begin{align*}
d_0 &= u_0, \\
d_{n+1} &= u_0 + u_1 d_n + u_2 (d_n)^2 + \ldots + u_l (d_n)^l, \\
h_n &= (d_n)^{u_1} + \ldots + (d_n)^{u_m}.
\end{align*}
\]

In general the tremendous size of the sets of propositional clauses to test necessitates the use of finer methods.
5.4 The semantics of computation

The method taken with first-order logic was first to define a syntax (signature, terms, variables, formulas), then to define an interpretation scheme for the syntactic objects, associating values (algebras, valuations, interpretations) to these objects. The Boolean algebra 2 was used, and the connectives were interpreted as Boolean operations.

For computation, the same method must be used. The syntactic objects are programs written in programming languages. For the purposes of this book, they are the terms of the \( \lambda \)-calculus, of rewriting systems or of recursive definitions. The values associated with them will not be from a Boolean algebra, but, rather, from a complete partial order. Before introducing the interpretation schemes, it will be shown how an order relation can give meaning to a computation.

The ideas presented here are due to D. Scott, whose work came forty years after the formalization of computability by Church, Gödel and Turing, discussed in Chapter 7, and Tarski’s semantic approach to logic. The originality of Scott’s contribution cannot be denied. His work led to the first model of the \( \lambda \)-calculus. In computer science, he devised, along with C. Strachey, the denotational semantics of programming languages, which today constitutes a reference for other semantic approaches. Since the work of Scott, the natural framework for denotational semantics is that of complete partial orders.

Scott’s main idea was to describe computation using a relation of information approximation. This idea is quite familiar in numerical analysis, but there it is more related to the nature of the objects themselves and the quality of the result, rather than the process itself in the computation.

5.4.1 Partial functions and complete orders

One of the essential operational problems of computation, such as it is practised in programming, or analyzed in the \( \lambda \)-calculus, is termination. The non-termination of computations will be interpreted using partial functions. A partial function from \( A \) to \( B \) is written \( f : A \rightarrow B \). The same function is a (total) function from a subset \( D(f) \subseteq A \), called the domain of \( f \), which takes values in \( B \). The notation \( f(a) \downarrow \) means that the partial function \( f \) is defined at \( a \), i.e., \( a \in D(f) \). Otherwise, write \( f(a) \uparrow \).

The following discussion uses partial functions from \( \mathbb{N}^k \) to \( \mathbb{N} \). Write \( \mathcal{F}(\mathbb{N}^k, \mathbb{N}) \), or simply \( \mathcal{F} \), for their set.

The graph of \( f : \mathbb{N}^k \rightarrow \mathbb{N} \) is the set

\[
G(f) = \{(\bar{m}, n) \in \mathbb{N}^k \times \mathbb{N}; f(\bar{m}) \downarrow \text{ and } f(\bar{m}) = n\}.
\]

Its projection into \( \mathbb{N}^k \) is the domain \( D(f) \) of \( f \). Write \( f \subseteq g \) if \( G(f) \subseteq G(g) \): this relation means that \( g \) is an extension of \( f \), i.e., that if \( f(\bar{m}) \downarrow \) then \( g(\bar{m}) \downarrow \) and
\( g(\bar{m}) = f(\bar{m}) \). Information is gained in passing from \( f \) to \( g \). Each of the lower bounds of \( g \) constitutes partial information about \( g \).

Consider the ordered set \( (\mathcal{F}, \subseteq) \). There is a smallest element \( \emptyset \), the partial function with empty domain. The total functions are the maximal elements. Two arbitrary functions do not necessarily have a common upper bound, but should they do so, then they have a least upper bound.

More generally, a non-empty subset \( E \) of an ordered set \( X \) is consistent (respectively directed) if two arbitrary elements have a common upper bound in \( X \) (respectively in \( E \)).

**Definition 5.2** A complete partial order (cpo) is an ordered set with a smallest element, generally written \( \bot \), such that every directed subset \( E \) has a least upper bound \( \sup E \).

Note that every increasing sequence in a cpo has a least upper bound.

For approximation to have meaning in computer science, it is normal to work in a complete space. It is very difficult in mathematics to do analysis seriously (in particular functional analysis) in non-complete metric spaces. For the example of \( \mathcal{F} \) and others which will follow, the order relations model an intuitive notion of information approximation. So it seems natural that computational objects be elements of a complete partial order. The set \( \mathcal{F} \) of partial functions is an important example (see exercise 23). Others are

- complete lattices, in particular the set \( \mathcal{P}(X) \) of subsets of a set;
- the flat orders \( X_\bot = X \cup \{\bot\} \), where \( \bot \notin X \), under the order \( x \leq y \) if \( x = y \) or \( x = \bot \);
- the sets of formal terms \( T_{E,U}(\Omega) \).

The framework of complete partial orders is still too big. The set \( \mathcal{F} \) satisfies other important properties, in particular, a concept of finiteness. To interpret the effective character of computation, the subject of Chapter 7, finite approximations are used (since only the finite is representable in extension; certain infinite sets are representable by intension, by formulas or by programs, and a semantics must be given to those non-elementary representations). The approximations of a partial function \( f \) are all the partial functions of which \( f \) is the extension. There are some which are finite: they are the partial functions of finite domain, which can be effectively described extensionally using a finite sequence of pairs \( (\bar{m}, n) \). It is clear that a partial function is the least upper bound of its finite approximations. To compute a partial function is to reach it through its finite approximations.

The concept of finite object can be formalized without referring to the cardinality of a set. In a complete partial order, an element \( x \) is called compact, or isolated, if for every directed subset \( E \), \( x \leq \sup E \) implies the existence of \( x_0 \in E \) such that \( x \leq x_0 \).

A complete partial order \( X \) is algebraic if for every \( x \in X \), the set of its compact lower bounds is directed and has \( x \) as least upper bound. The set \( \mathcal{F} \) is algebraic (see
exercise 24). Suppose, furthermore, that the set of compact elements is countable, as always for reasons of computability. This is evidently the case for \( \mathcal{F} \).

**Definition 5.3** A (Scott) domain is a complete, algebraic, partial order in which the set of compact elements is countable.

A computation is described using domains. How is a function \( f : X \to Y \) computed in this framework? For each \( x \in X \), \( f(x) \) is approximated from an approximation of \( x \), so a definition of continuity over \( f \) is needed.

A function \( f : X \to Y \) between complete ordered sets is continuous if for every directed subset \( E \) of \( X \), \( f(\text{sup } E) = \text{sup } f(E) \). In particular, a continuous function is monotone (take for \( E \) a pair \( \{a, b\} \) with \( a \leq b \)). In the case where \( X \) is an algebraic order, a function is continuous if for every \( x \in X \), \( f(x) \) is the least upper bound of the set of images of the compact lower bounds of \( x \): it suffices to compute over ‘finite’ objects.

The following result, along with the slogan

\[
\text{recursive definition} = \text{fixpoint equation}
\]

explains the utility of this modeling by itself.

**Theorem 5.18** Let \( X \) be a complete ordered set and \( \Phi : X \to X \) a continuous function. Then \( \Phi \) has a least fixpoint, written \( \mu \Phi \).

**Proof** Since \( \Phi \) is monotone, and since \( \bot \) is the smallest element of \( X \), the sequence \( \bot, \Phi(\bot), \Phi^2(\bot), \ldots, \Phi^n(\bot), \ldots \) is increasing, and hence directed. By the completeness of \( X \), this sequence has a least upper bound \( \mu \Phi \), and by the continuity of \( \Phi \),

\[
\Phi(\mu \Phi) = \Phi(\text{sup } \Phi^n(\bot))
\]

\[
= \text{sup } \Phi(\Phi^n(\bot))
\]

\[
= \text{sup } \Phi^n(\bot)
\]

\[
= \mu \Phi.
\]

Hence \( \mu \Phi \) is a fixpoint of \( \Phi \). If \( b \) is another fixpoint, \( \bot \leq b \) implies \( \Phi^n(\bot) \leq \Phi^n(b) = b \), hence \( \mu \Phi \leq b \): \( \mu \Phi \) is the least fixpoint.

Consider the classical example of recursive definitions, to be studied in more detail in the next section. Consider the definition

\[
f(x, y) = \begin{cases} y & \text{if } x = 0 \\ f(x - 1, y + 1) & \text{otherwise.} \end{cases}
\]
Interpret this definition in \( \mathcal{F} \). It is a fixpoint equation for the operator \( \Phi : \mathcal{F} \to \mathcal{F} \) defined by

\[
\Phi : f \mapsto (x, y) \mapsto \begin{cases} 
  y & \text{if } x = 0 \\
  f(x - 1, y + 1) & \text{otherwise.}
\end{cases}
\]

\( \Phi \) is a continuous operator. The least fixpoint of \( \Phi \) is the least upper bound of the sequence \( \Phi^n(\emptyset) \).

### 5.4.2 Flat orders, strict and sequential functions

The preceding approach using partial functions is not satisfactory, compared to the semantics of first-order logic, where every formula receives a value (0 or 1) in a given interpretation, be it provable or not. If \( f \) is a partial function, and \( n \in D(f) \), the value \( f(n) \) exists. But if \( n \notin D(f) \), \( f(n) \) does not exist. A fairer treatment for both cases consists of introducing a value \( \bot \notin \mathbb{N} \) to define \( f(n) = \bot \) if \( n \notin D(f) \).

For every set \( X \), define the flat ordered set \( X_\bot = X \cup \{ \bot \} \), where \( \bot \notin X \), under the order \( x \leq y \) if \( x = y \) or \( x = \bot \). This ordered set is complete, having as directed subsets only the \( \{ \bot, x \} \) for \( x \in X \) and the one-element subsets.

The symbol \( \bot \), the smallest element of \( X_\bot \), is the minimal information about a computation, expressing an undefined result, the result of a non-terminating computation, or, better, one which has not yet terminated. In the same way that the elements of \( \mathcal{F} \) are partial functions, the elements of \( \mathbb{N}_\bot \) can be considered as partial integers: an ordinary integer is a ‘total’ integer, completely computed, while \( \bot \) is a (very) incompletely computed integer.

A function \( u : X_\bot \to Y_\bot \) is monotone if for every \( x \in X \), \( \bot \leq x \) (the only possible inequality in \( X_\bot \)) implies \( u(\bot) \leq u(x) \), hence \( u(\bot) = u(x) \) or \( u(\bot) = \bot \). In the first case \( u \) is constant, in the second, \( u \) is strict. In the case of flat orders, a monotone function is necessarily continuous.

The set of continuous functions from \( E \) to \( F \) is written \([E \to F]\). It is given the pointwise order: \( f \leq g \) if for every \( x \in E \), \( f(x) \leq g(x) \). Now, it is \([\mathbb{N}_\bot \to \mathbb{N}_\bot]\) which plays the role of \( \mathcal{F}(\mathbb{N}, \mathbb{N}) \). Fixpoint equations can also be solved here, because of the following property:

**Proposition 5.19** \([E \to F]\) is complete if \( E \) and \( F \) are complete partial orders.

**Proof** The constant function of value \( \bot_F \) is the smallest element of \([E \to F]\). Furthermore, if \( D \) is a directed subset of \([E \to F]\), then the set of \( f(d) \) for \( d \in D \) is also directed, and its least upper bound is computed pointwisely: \( \sup D)(x) = \sup_{f \in D} f(x) \).

A recursive definition

\[
f(x) = \begin{cases} 
  1 & \text{if } x = 0 \\
  x \ast f(x - 1) & \text{else}
\end{cases}
\]
is interpreted as a fixpoint equation for the operator $\Phi : [N_\perp \to N_\perp] \to [N_\perp \to N_\perp]$:

$$f \mapsto \forall x \in N_\perp. \text{if } x = 0 \text{ then } 1 \text{ else } x \ast f(x - 1).$$

The successive approximations $f_n = \Phi^n(\bot_{N_\perp - N_\perp})$ of $\mu \Phi$ are easy to compute:

$$f_0(x) = \bot \quad \text{for every } x \in N_\perp,$$

$$f_1(x) = \begin{cases} 1 & \text{if } x = 0 \\ \bot & \text{otherwise,} \end{cases}$$

$$f_n(x) = \begin{cases} x! & \text{if } x < n \\ \bot & \text{otherwise.} \end{cases}$$

Scott’s induction principle is generally used to prove properties about the least fixpoint:

**Proposition 5.20** Let $D$ be a complete order, $P$ be a predicate over $D$ stable under least upper bounds, i.e., if $E \subseteq D$ is directed and $E$ satisfies $P$, then $\sup E$ satisfies $P$, and $\Phi : D \to D$ be a continuous function. If

1. $P(\bot)$, and
2. $P(x) \Rightarrow P(\Phi(x))$ for every $x \in D$,

then $P(\mu \Phi)$.

**Proof** From (1) and (2), and from the monotonicity of $\Phi$, by induction over the integers, $P(\Phi^n(\bot))$ for every $n \geq 0$. Since $P$ is stable under sup, it follows that $P(\sup_{n \geq 0} \Phi^n(\bot))$. If $\Phi$ is continuous, $\mu \Phi = \sup_{n \geq 0} \Phi^n(\bot)$ therefore satisfies $P$. □

There is a correspondence between partial functions and strict functions. If $f : X \to Y$ is a partial function between two sets $X$ and $Y$, define the (total) function $f^* : X_\perp \to Y_\perp$, called the strict extension of $f$, by:

$$f^*(x) = \begin{cases} f(x) & \text{if } x \in D(f) \\ \bot & \text{otherwise.} \end{cases}$$

Not all functions from $X_\perp$ to $Y_\perp$ are strict extensions of partial functions. For example, there are two kinds of constant function: strict extensions $c^*$ of constant functions from $X$ to $Y$, and ‘real’ ones.

By replacing partial functions with their extensions over flat orders, new objects must now be considered, such as those non-strict constant functions. It is easy to give an operational meaning to such objects. A strict function can be evaluated using call by value: its argument is first evaluated, and if the value is defined, the result of the function is as well. However, the argument of a non-strict constant function must not be evaluated, since the result can be given immediately, and the evaluation of the argument is unnecessary, possibly dangerous if it does not terminate. Call by name must be used. Abstract interpretation, described below,
can be used to determine when a function can be called by value without any danger.

If in the case of flat orders, non-strict monotone functions seem to be the exception (only the constant functions), there are many others in more ‘rich’ complete orders. Consider the case of the Cartesian product: if $X$ and $Y$ are complete orders, then the Cartesian product $X \times Y$, with the order $(x, y) \preceq_{X \times Y} (x', y')$ if $x \preceq_X x'$ and $y \preceq_Y y'$, is also a complete order.

**Definition 5.4** Let $D_i$ be a complete order, and $f : D_1 \times \ldots \times D_n \to D_0$ a continuous function. Then $f$ is strict in its $i$-th argument, written strict($i$), if

$$f(x_1, \ldots, x_{i-1}, \bot_{D_i}, x_{i+1}, \ldots, x_n) = \bot_D.$$ 

It is important to be able to know that a function is strict in an argument: the evaluation of that argument can then be done safely, i.e., the non-termination of that evaluation will not prevent a value from being produced. The arithmetic operations of programming languages are generally strict in each of their arguments. The two arguments of an addition must be evaluated, and the result of an addition diverges as soon as one of the arguments diverges ($x + \bot = \bot + y = \bot$, for all $x, y \in \mathbb{N}_\bot$).

In most programming languages, where parameter passing is done by value, the functions defined by a program are strict. However, certain predefined functions, or special forms in the Lisp sense, have a different evaluation mode and are not strict.

For example, the conditional (if _ then _ else _ in ML, (_ _ ? _ _ : _) in C) is strict in its first argument, which is always evaluated, and non-strict in the two last arguments. Only one will be evaluated, depending on the value of the first argument.

Each Boolean operator has a strict version and several non-strict versions. The or of Pascal always evaluates its two arguments, but the or of Lisp (as well as the ‘short-cut’ or else of Ada) always evaluates its first argument, and should it yield the value true, returns true without evaluating its second argument. The and does not evaluate its second argument if the first argument is false, which allows one to write expressions such as if $x \neq 0$ and ($1/x < 1$) ... . The Boolean operations therefore do not have the same semantics in all languages. Let $2_\bot = \{\bot, 0, 1\}$ be the flat order over the Booleans 0 and 1. Every function from $2^n$ to $2$ has a unique strict extension and several other interesting extensions. Consider the case of conjunction. The strict extension is defined by $\bot \land^x \bot = \bot$: it is the least defined extension. The extension corresponding to the and form of Lisp, called left sequential extension, satisfies

$$\bot \land^l x = \bot,$$

$$0 \land^l \bot = 0,$$

$$1 \land^l \bot = \bot.$$

It is strict only in its first argument. It is an example of a sequential function.
Definition 5.5 A function \( f : D_1 \times \ldots \times D_n \rightarrow D_0 \) is sequential in \( x \in D_1 \times \ldots \times D_n \) if there exists an index \( i \) such that for every \( y \geq x \) for which \( y_i = x_i \), \( f(x) = f(y) \). Then \( f \) is sequential if it is for all \( x \).

We show that \( \wedge^i \) is sequential. It is sequential in \( (\bot, x) \) in the index 1, since for every \( y \), \( \bot \wedge^i x = \bot \wedge^i y (= \bot) \); in \((0, x), (1, 0)\) and \((1, 1)\) in indices 1 and 2, since the result in 2 cannot be more defined for those arguments; and in \((1, \bot)\) in index 2.

One can design an extension of conjunction which is more defined than \( \wedge^p \); it is the parallel extension of conjunction: \( \bot \wedge^p 0 = 0 \wedge^p \bot = 0 \) and \( \bot \wedge^p 1 = 1 \wedge^p \bot = \bot \wedge^p \bot = \bot \). It is not sequential since in \((\bot, \bot)\) the result can be improved in two different manners. This extension is not implemented in sequential languages. A parallel implementation consists of evaluating the two arguments in parallel and stopping as soon as one of them has the value 0. A sequential simulation would evaluate the arguments in turn, which would ensure termination should one of the arguments obtain the value 0.

More generally, for every function \( f : 2^n \rightarrow 2 \), there is an extension \( f^p \), strict in each of its arguments, which is the minimal continuous extension of \( f \), and a maximal continuous extension \( f^p \). To compute \( f^p(x_1, \ldots, x_n) \) where \( x_1, \ldots, x_n \in 2_{\bot} \), compute the set \( F(x_1, \ldots, x_n) \) of all the \( f(x'_1, \ldots, x'_n) \), where \( x'_i = x_i \) if \( x_i \in 2 \), and \( x'_i \in 2 \) if \( x_i = \bot \), and let

\[
f^p(x_1, \ldots, x_n) = \begin{cases} 
a & \text{if } F(x_1, \ldots, x_n) = \{a\} \\
\bot & \text{if } \text{card } F(x_1, \ldots, x_n) > 1.
\end{cases}
\]

Another useful construction is that of the sum of two domains. Let \( D_1 \) and \( D_2 \) be two domains whose smallest elements are \( \bot_1 \) and \( \bot_2 \). Define \( D = D_1 + D_2 = D_1 \cup D_2 \cup \{\bot\} \), by extending the orders over \( D_1 \) and \( D_2 \) by \( \bot < \bot_1 \) and \( \bot < \bot_2 \). There are then several layers of partial information: if \( d = \bot \), nothing is known about \( d \); if \( d = \bot_1 \), it is known that \( d \in D_1 \). This operation is useful in programming, to define concrete data types and matching functions:

\[
\text{datatype } D = \text{ one of } D_1 \mid \text{ two of } D_2 ;
\]

Among the strict functions over \( D \), the ‘discriminating’ functions only need to know the constructor of the argument:

\[
\text{fun } f (\text{one } _\bot) = 1 \mid (\text{two } _\bot) = 2 ;
\]

\[
f(\bot) = \bot, \text{ but } f(\bot_1) = 1 \text{ and } f(\bot_2) = 2.
\]

5.4.3 Infinite objects and approximations

Constructing a flat order introduced partial elements, which play an important role in Cartesian products. For other constructions, it is necessary to introduce infinite elements: the most important example is that of infinite lists.

Scott's methods are useful since domains can be constructed for all the usual data structures; a typical example is that of lists. (Finite) lists are finite terms
over a many-sorted signature formed of a constant \( \bot \) and a binary constructor \( :: \) (cons). The concept of strict function is not precise enough. For example, the function \( l \) computing the length of a list is strict in the sense that \( l(\bot) = \bot_n \), but it only requires a limited evaluation of its argument. Only its skeleton need be known:

\[
l(\bot :: x) = 1 + l(x).
\]

The hd function only requires the evaluation of the head of a list, and the tl function the rest of the list.

These remarks are of interest only if cons does not evaluate its arguments systematically. Programming languages have therefore been conceived where cons, unlike the predefined arithmetic operations, does not evaluate its arguments (or, at least, does not evaluate its second argument): a call \( \text{cons} a 1 \) can, for example, create a pair of pointers to closures containing the unevaluated arguments and the current environment, which are awoken subsequently by hd or tl. These are lazy pairs.

Suppose that the function cons is non-strict, and consider the function \( x \mapsto 1 :: x \) from \( D \) to \( D \). It is a continuous function as soon as the binary constructor :: is continuous. It has a fixpoint, the limit of the increasing sequence \( \bot, 1 :: \bot, 1 :: 1 :: \bot, 1 :: 1 :: 1 :: \bot, \ldots \). If the operation :: were strict in its second argument, then it would be the constant sequence \( \bot \). Otherwise the limit represents the infinite list \( 1 :: 1 :: 1 :: \ldots \).

This problem will be examined in the more general framework of infinite terms. These were introduced in Chapter 1, by removing the finiteness restriction on the tree domain which had been imposed to ensure the description of finite terms. Let \( \Sigma \) be a signature. In this section, the elements of \( T_\Sigma \) are called \textit{finite terms}, and those of \( T_\Sigma^\infty \) \textit{formal terms}. Two possible definitions of approximation were given in Chapter 1: using a distance or using an order relation by introducing a constant \( \Omega \). The resulting structures are a complete metric space and a complete ordered set. Each of these structures allows the modeling of a computation and the proof of, for example, fixpoint theorems, which is important in studying the algebraic semantics of rewriting systems, and program schemes.

The principle of structural induction allows only proofs of properties for finite terms. To be able to work with infinite terms, the principle of \textit{continuous induction} is used. It combines structural induction with finite partial terms with a passage to the limit, made possible by the algebraic character of \( T_\Sigma^\infty \cup \{\Omega\} \). Depending on the case, the true properties of all formal terms, finite or not, or simply of non-finite terms are of interest. In the second case, structural induction will be used over a subsignature formed of symbols of arity \( \geq 1 \) (since a constant is not a "base case" for infinite terms!).

\textit{Proposition 5.21} Let \( P \) be a predicate over \( T_\Sigma^\infty \cup \{\Omega\} \), stable under least upper bound, and \( \Sigma' \subseteq \Sigma \). If

1. \( P(\Omega) \),
2. \( P(M_1), \ldots, P(M_n) \) imply \( P(fM_1 \ldots M_n) \) for every \( f \in \Sigma \)'s of arity \( n \), then \( P(M) \) for every \( M \in T^\infty_{\Sigma \cup \{ \Omega \}} \).

Proof By structural induction, \( P \) is true over \( T_{\Sigma \cup \{ \Omega \}} \). But every formal term \( M \in T^\infty_{\Sigma \cup \{ \Omega \}} \) is the least upper bound of its finite approximations \( \tau_k(M) \in T_{\Sigma \cup \{ \Omega \}} \). Since \( P \) is stable under least upper bound, it follows that \( P(M) \) for every \( M \in T^\infty_{\Sigma \cup \{ \Omega \}} \).

Example Consider the following program for the concatenation of two lists:

\[
\begin{align*}
\text{fun} & \quad \text{append} \; 1 \; 1' \\
& = \text{if} \ (\text{null} \; 1) \ \text{then} \ 1' \ \text{else} \ (\text{hd} \; 1) :: (\text{append} \ (\text{tl} \; 1) \ 1') \\
\end{align*}
\]

It will be shown that if \( :: \) is not strict, then \( \text{append} \; 1 \; 1' = 1 \) if \( 1 \) is an infinite list. Structural induction is used over partial lists, i.e., lists of the form \([a;b;c;\ldots;\Omega] \): there are two cases, the base case for \( \Omega \), and the induction step for \( 1 \) to \( a :: 1 \). First, if \( \text{if} \) is strict in its first argument, and \( \text{null} \) is strict; it follows that \( \text{append} \; \Omega \; 1' = \Omega \). Furthermore, if \( \text{append} \; 1 \; 1' = 1 \), then \( \text{append} \ (a :: 1) \ 1' = a :: (\text{append} \ 1 \ 1') = a :: 1 \). On the other hand, it is not true that \( \text{append} \; 1' = 1' \), so this property does not hold for (real) non-partial finite lists.

5.4.4 Models

As for first-order terms and formulas, the interpretation schemes are defined from the syntactic structure of expressions. We will examine two important cases, the \( \lambda \)-calculus and program schemes.

\( \lambda \)-calculus

The first attempt to give meaning to \( \lambda \)-calculus terms would be to interpret abstractions as functions and applications as applications of a function to an argument. Unfortunately, set theory does not allow the application of a function to itself, and so there can be no interpretation of a term such as \( \lambda x.x \).

The situation is simpler in the typed \( \lambda \)-calculus where self-application is not allowed by the syntax. It is easy to imagine an interpretation where each type \( \tau \) is interpreted by a set \( D_\tau \), where \( D_{\tau_1 \to \tau_2} \) is an appropriate set of functions from \( D_{\tau_1} \) to \( D_{\tau_2} \). For example, \( D_{\text{num}} = N \), \( D_{\text{num} \to \text{num}} = N^N \), or else \( D_{\text{num}} = N_\perp \), \( D_{\text{num} \to \text{num}} = [N_\perp \to N_\perp] \).

In the case of the pure \( \lambda \)-calculus, the syntax must be followed more closely. Discussion will begin with combinatory logic, a variant of the \( \lambda \)-calculus with no bound variables.

When \( \Sigma \) is an applicative signature, formed of a unique binary symbol \( \text{App} \) and of a certain number of constant symbols, a \( \Sigma \)-algebra is called an applicative algebra. The applicative notation will still be used, omitting the \( \text{App} \) symbol, and
associating parentheses to the left. In an applicative algebra $\mathcal{A}$, the ‘product’ $ab$ of two elements $a$ and $b$ of $\mathcal{A}$ represents the application of the ‘function’ $a$ to the ‘argument’ $b$. In the case of combinatory logic, the constants are $S$, $K$ and $I$ and the first-order axioms are

$$Sxyz = x(yz),$$
$$Kxy = x,$$
$$Ix = x.$$ 

A model of this theory, in the sense of first-order logic, in which the constants $S$ and $K$ are interpreted by two distinct elements, is called a combinatory algebra. The existence of these algebras is a consequence of the Church–Rosser theorem, which implies the consistency of the equational theory (two distinct normal forms are not interconvertible, by the Church–Rosser property). It is easy to construct a combinatory algebra in a standard manner by taking the quotient of the set of terms under an equivalence relation (see Chapter 5). This model does not tell us anything about the calculus: if in this model, the self-application $a(a)$ has a meaning, nothing says that the left $a$ acts as a function.

The first model of the $\lambda$-calculus, $D_\infty$, was constructed by Scott in 1969 (unpublished). A second, simpler model, $P\omega$, is due to Plotkin (1972 [121]).

Free variables are treated as usual using an environment. Write $[M]\rho$ for the value of a term $M$ in the environment $\rho$. This notation is standard in denotational semantics, and corresponds to the $M^D(\rho)$ used in first-order logic. The value $[M]\rho$ is defined using the syntax of the terms:

$$[x]\rho = \rho(x)$$
$$[[MN]]\rho = ([M]\rho)([N]\rho)$$
$$[\lambda x M]\rho = \lambda d \in D. [M](\rho; (x = d))$$

where $\lambda d \in D. foo$ is the $\lambda$-notation for a function $d \mapsto foo$ (it is not a $\lambda$-term!), and $\rho; (x, d)$ designates the environment $\rho$ extended by the binding $(x = d)$.

Here is the construction of the model $P\omega$ of Plotkin. Write $P\omega$ for the set of subsets of $\mathbb{N}$, ordered by set inclusion. It is a complete order (also a complete lattice) and even algebraic, a compact element being a finite subset of $\mathbb{N}$. A continuous function can therefore be computed using finite approximations: $f : P\omega \rightarrow P\omega$ is continuous if and only if for every $x \in P\omega$,

$$f(x) = \bigcup\{f(a); a \text{ finite } \subseteq x\}.$$ 

In other words, $n \in f(x)$ if there exists a finite subset $a$ of $x$ such that $n \in f(a)$ (in Chapter 1, a function $f : \mathcal{P}(X) \rightarrow \mathcal{P}(X)$ having this property was called finitary). One can then represent $f$ by the set of pairs $(a, n)$ such that $n \in f(a)$. But these pairs are finite objects which can be coded by integers in the standard manner (these encodings are defined and used in Chapter 7). A finite subset $\{i_1, \ldots, i_n\}$
is coded by the integer \( m = \sum_{k=1}^{n} 2^{ik} \). Let \( a = e_m \), where the elements of \( e_m \) are the positions occupied by 1's in the binary representation of \( m \). There is also an encoding \( \langle \cdot \rangle : N^2 \rightarrow N \) which codes the pair \( (e_m, n) \) by the pair of integers \( (m, n) \), then by the integer \( \langle m, n \rangle \), and this encoding is bijective.

Hence, a continuous function from \( P\omega \) to \( P\omega \) is represented by a set of integers, i.e., by an element of \( P\omega \): if \( f \in [P\omega \rightarrow P\omega] \), define

\[
G(f) = \{ \langle m, n \rangle; n \in f(e_m) \}.
\]

It is now easy to interpret application by a binary operation over \( P\omega \): if \( a, b \in P\omega \), define

\[
a b = \{ n \in N; (\exists e_m \subseteq b) \langle m, n \rangle \in a \}.
\]

In other words, \( n \in ab \) if there exists an element \( \langle m, n \rangle \) of \( a \)—seen as a function—allowing the computation of \( a(b) \), i.e., in which \( m \) is a finite approximation of \( b \).

From this binary operation is derived a 'homothety' \( H \): for \( a \in P\omega \), \( H(a) \) is the function \( x \mapsto ax \). Conversely, the application is defined from \( H \) by \( ab = H(a)(b) \).

It is easily shown that \( H(a) \) is continuous, that

\[
H : P\omega \rightarrow [P\omega \rightarrow P\omega]
\]

are continuous, and that

\[
(H \circ G)(f) = f
\]

for every \( f \in P\omega \). It is now possible to interpret \( \lambda \)-terms in \( P\omega \):

\[
[x]_\rho = \rho(x), \\
[M N]_\rho = ([M]_\rho)[[N]_\rho], \\
[\lambda x M]_\rho = G(\lambda a \in P\omega . [M]_\rho; (x = a)) \\
= \{ \langle m, n \rangle; n \in [M]_\rho; (x = a) \}.
\]

Here are a few examples of interpretation:

\[
[\lambda x x] = \{ \langle m, n \rangle; n \in e_m \}, \\
[\lambda x y . x] = \{ \langle m_1, \langle m_2, n \rangle \rangle; n \in e_{m_1} \}, \\
[\lambda x y z . x z(y z)] = \{ \langle m_1, \langle m_2, \langle m_3, n \rangle \rangle \rangle; n \in e_{m_1} e_{m_2} e_{m_3} \}.
\]

Two examples will be used to show how this model \( P\omega \) can be used to study the consistency of extensions of the \( \lambda \)-calculus.

Unsolvable terms What is the meaning of a non-normalizable term? It is tempting to identify 'non-normalizable' with 'undefined'. However, there do exist non-normalizable terms which have a useful and well-defined operational behavior, such as the fixpoint combinators. It is, furthermore, inconsistent to equate all non-normalizable terms. Suppose the two non-normalizable terms \( M \equiv \lambda x . x K \Omega \) and
$N \equiv \lambda x . xS\omega$ are equated, where $\omega \equiv \omega \omega$, with $\omega \equiv \lambda x . xx$. If $M = N$, then $MK = NK$, i.e., $K = S$, which implies (exercise) that $x = y$ for arbitrary variables $x$ and $y$.

The concept of a normalizable term is much too syntactic. A semantic condition characterizing ‘undefined’ terms is needed. In the model $P\omega$, an appropriate condition would be $[M] = \varnothing$, the smallest element of $P\omega$. Exercise 31 shows that this is the case for $\Omega$. It is easy to show (see exercise 30) that if $[M] = \varnothing$, then $[MN] = [\lambda x M] = \varnothing$. This is clearly a different concept from non-normalizability, since $MN$ can be normalizable without $M$ being normalizable (see $Y(KI)$ in the following example). Barendregt and Wadsworth introduced a finer concept:

**Definition 5.6** A closed term is **solvable** if there exist terms $N_1, \ldots, N_p$ such that $MN_1 \cdots N_p = I$. An arbitrary term $M$ is solvable if the closed term $(\lambda x_1 \cdots x_n . M)$ is solvable, with $\{x_1, \ldots, x_n\} = \text{var}(M)$. A term is **unsolvable** if it is not solvable.

**Example** $K$ is solvable since $KIX = I$. The term $Y \equiv \lambda f . (\lambda x . f(xx))(\lambda x . f(xx))$ is solvable since $Y(KI) = KI(Y(KI)) = I$. The term $\Omega$ is unsolvable since $\Omega N$ can only yield by $\beta$-reduction a term of the form $\Omega N'$.

From exercise 30, and since $[I] \neq \varnothing$, it is clear that $M$ is unsolvable if $[M] = \varnothing$: a proof of unsolvability is obtained from $\Omega$ by semantic means, without considering reductions. The converse, whose proof can be found in [8, theorem 19.1.10], is more difficult.

**Theorem 5.22** $M$ is unsolvable if and only if $[M]^{P\omega} = \varnothing$.

There is no risk, as for the non-normalizables, of equating all terms, by equating the unsolvable terms: this equivalence relation is consistent with the $\lambda$-calculus. The unsolvable terms can be considered as undefined terms, all carrying the same amount of information, which is minimal. They also have a characterization with respect to $\beta$-reduction.

**Definition 5.7** A term $M$ is in **head normal form (hnf)** if it is of the form

$\lambda x_1 \ldots x_p . yM_1 \ldots M_q$,

with $p, q \geq 0$. Variable $y$ is called the head variable. Terms which are not in hnf are necessarily of the form $\lambda x_1 \ldots x_p . (\lambda y M_0)M_1 \ldots M_q$. The redex $(\lambda y M_0)M_1$ is called the head redex. A term $M$ has a hnf if there exists an $M'$ in hnf such that $M = M'$.

A term is solvable if and only if it has a hnf [8, theorem 8.3.14]. Furthermore, the hnf of a term, if it exists, can be attained by a head reduction, i.e., by contracting the head redex at each step. For example, $Y$ is not a hnf, having the head redex $(\lambda x . f(xx))(\lambda x . f(xx))$. Contracting yields $\lambda f . f((\lambda x . f(xx))(\lambda x . f(xx)))$, a hnf with $f$ as head variable.

So, if unsolvable terms are considered to have no value, it suffices to attempt to reduce a term to hnf.
The $\lambda$-calculus with pairs  As a second application of these semantic methods, pairs are added to the $\lambda$-calculus. Pairing combinators have already been defined:

$$[M, N] \equiv \lambda z . z M N,$$

$$\pi^1 \equiv \lambda p . p T,$$

$$\pi^2 \equiv \lambda p . p F.$$

So

$$\pi^1 [M, N] \beta \rightarrow M, \quad \pi^2 \beta \rightarrow N,$$

but it is not true that $[\pi^1 P, \pi^2 P] \not\rightarrow P$ for an arbitrary term $M$. One might ask what is the ‘meaning’ of the first projection of a term which is not a pair.

As a combinator satisfying the three preceding reduction properties does not exist (Barendregt 1984 [8]), the $\lambda$-calculus is extended by adding constant symbols $\pi, \pi^1,$ and $\pi^2$ with the axioms

$$\pi^1 (\pi x y) = x, \quad \pi^2 (\pi x y) = y, \quad \pi (\pi^1 x)(\pi^2 x) = x.$$

Is the resulting theory, $\lambda \pi$, consistent, i.e., do these axioms imply the equality of all terms? The simplest proof of the consistency of the pure $\lambda$-calculus uses the Church–Rosser theorem: two new distinct terms cannot be interconverted. Unfortunately, the natural extension of $\beta$-reduction by the three rules

$$\pi^1 (\pi x y) \rightarrow x, \quad \pi^2 (\pi x y) \rightarrow y, \quad \pi (\pi^1 x)(\pi^2 x) \rightarrow x,$$

is not confluent (see Klop [84]). A proof of consistency by semantic methods is needed. (A proof of syntactic consistency is, however, possible; see Klop and de Vrijer [85]).

In the same way that the axiom $1 + 1 = 0$ is proven to be consistent with field theory by showing that it is satisfied in the field $\mathbb{Z}/2\mathbb{Z}$, the consistency of $\lambda \pi$ is proven by showing that $P \omega$ is a model. Let

$$f(a, b) = \{2m; m \in a\} \cup \{2m + 1; m \in b\},$$

$$f_1(c) = \{m; 2m \in c\},$$

$$f_2(c) = \{m; 2m + 1 \in c\}.$$  

These three functions satisfy

$$f_1 (f(a, b)) = a, \quad f_2 (f(a, b)) = b, \quad f(f_1(c), f_2(c)) = c.$$  

However, to interpret the constants $\pi, \pi^1$ and $\pi^2$, elements of $P \omega$ are needed, while it was functions over $P \omega$ which were constructed. In fact, by a continuity argument, it can be shown that these functions are representable by elements of $P \omega$, which establishes the consistency of $\lambda \pi$.

The introduction of pairs in the $\lambda$-calculus is done more naturally using a type system. The development of categorical combinatory logic, due to P.-L. Curien [31, 70], and its interest for the compilation of CAML have shown that pairs constitute an essential data structure for the ‘internal’ manipulation of semantic objects linked to the evaluation of terms, such as environments and stacks.
Recursive definitions

Let \( \Sigma \) be a signature formed of primitive function symbols \( u, v, w, \ldots \). Typically, for programs over the integers, \( \Sigma \) contains 0, the successor \( s \), the + and * operations, and a ternary conditional \( \text{cond} \). A recursive definition of \( \Delta \) over \( \Sigma \) is a set of equations (one for each symbol of \( \Delta \)),

\[
fx_1 \ldots x_r = M_f,
\]

where \( f \in \Delta, r = \text{ar}(f), x_1, \ldots, x_r \in X, M_f \in T_{\Sigma\Delta}[x_1, \ldots, x_r] \).

In the case of the \( \lambda \)-calculus, the fact that \( P\omega \) (as for \( D_\infty \)) is a complete partial order does not appear to be fundamental, since \( \lambda \)-calculus programs are terms, not equations: the fixpoint operators allow the resolution of equations in an internal manner. However, the semantics of recursive definitions is based on the resolution of fixpoint equations: complete orders become natural.

\( \Sigma \) is interpreted in a complete partial order \( D \), each symbol \( u \in \Sigma \) of arity \( n \geq 0 \) being interpreted by a continuous function \( u^D \in [D^n \rightarrow D] \). By definition, \( D \) is thus a complete \( \Sigma \)-algebra. Each term \( M \in T_\Sigma[X] \) has an interpretation \( M^D(\rho) = [M]\rho \) in the environment \( \rho : X \rightarrow D \).

To simplify, consider \( \Delta = \{f\} \), a symbol of arity \( r \). If \( h \) is an arbitrary continuous function of \( D^r \) to \( D \), the complete \( \Sigma \)-algebra \( D \) can be extended into a complete \( \Sigma \cup \{f\} \)-algebra \( D' \) by interpreting \( f \) by \( h \). Each term \( M \in T_{\Sigma\cup\{f\}}[X] \), in an environment \( \rho \), therefore has an interpretation \([M]'\rho \) in this extended algebra. If \( x_1, \ldots, x_n \) are variables of \( M \), a continuous function \( \tilde{M}(h) : D^r \rightarrow D \) is derived by

\[
\tilde{M}(h)(d_1, \ldots, d_n) = [M]'[x_1 := d_1; \ldots; x_n := d_n].
\]

Let this construction be applied to a recursive definition of \( f \) over \( \Sigma \), where \( f(x_1, \ldots, x_r) = M_f \). Let

\[
\Phi : [D^r \rightarrow D] \rightarrow [D^r \rightarrow D],
\]

\[
h \mapsto \tilde{M}_f(h).
\]

The recursive definition is then interpreted as the fixpoint equation \( \Phi(h) = h \). Since \( \Phi \) is continuous, this equation has a least solution \( f^D = \mu\Phi \in [D^r \rightarrow D] \), which is, by definition, the interpretation of \( f \) computed in \( D \) by the recursive definition.

The complete \( \Sigma \)-algebra \( D \) is therefore extended into a complete \( \Sigma \cup \{f\} \)-algebra. So each term \( M \in T_{\Sigma\cup\{f\}} \) can be interpreted in \( D \), and so also can programs of the form

\[
\text{let } f(x_1, \ldots, x_r) = M_f \text{ in } M \end{end} ;
\]

Among all the interpretations, a particular role is played by the complete initial \( \Sigma \)-algebra. Since \( T_\Sigma \) is in a standard way a \( \Sigma \)-algebra, the set \( H = T_\Sigma[\Omega] \) of formal partial terms, with the same interpretation of symbols, is a complete \( \Sigma \)-algebra, initial in the category of complete \( \Sigma \)-algebras. The initiality morphism
$i_D$ from $T_{\Sigma U}(\Omega)$ to a $\Sigma$-algebra $D$ is constructed as in the case of $T_E$, by adding $i_D(\Omega) = \bot_D$.

Computing the fixpoint of $\Phi : [H^r \rightarrow H] \rightarrow [H^r \rightarrow H]$ can be done formally in $T_{\Sigma U}(\Omega)[X]$; $\mu \Phi$ is the least upper bound of the increasing sequence $(F_k)_{k \geq 0}$ of finite partial terms, called the Kleene sequence:

$$F_0 = \bot_{[H^r \rightarrow H]}(x_1, \ldots, x_r) = \Omega,$$

$$F_{k+1} = \Phi(F_k)(x_1, \ldots, x_r).$$

The limit $F_\infty = \sup F_k$ is a formal term, which is by definition the formal interpretation of $f$. The initiality property allows the verification that for every complete $\Sigma$-algebra $D$, $i_D(F_\infty) = f^D$.

**Example** Consider the recursive definition of $f$ over $\Sigma = \{a, u, v\}$:

$$f(x) = u(x, a, f(v(x))).$$

The associated operator $\Phi : [H \rightarrow H] \rightarrow [H \rightarrow H]$ is defined by

$$\Phi(h)(M) = u(M, a, h(v(M)))$$

and the first elements of the Kleene sequence are

$$F_0 = \bot_{[H \rightarrow H]}(x) = \Omega,$$

$$F_1 = \Phi(F_0)(x) = u(x, a, \Omega),$$

$$F_2 = \Phi(F_1)(x) = u(x, a, u(v(x), a, \Omega)),$$

$$F_3 = \Phi(F_2)(x) = u(x, a, u(v(x), a, u(v(v(x)), a, \Omega))), \ldots.$$

\[\square\]

### 5.4.5 Abstract interpretations

A model of the $\lambda$-calculus must satisfy the theory of $\beta$-conversion, i.e., if two terms $M$ and $N$ are $\beta$-convertible ($M =_\beta N$), then $\llbracket M \rrbracket_\rho = \llbracket N \rrbracket_\rho$ for every environment $\rho$ ($D \models M = N$). The result is that the reduction mechanism is interpreted by equality. It means a considerable impoverishment of the operational content of the calculus—much as the deductive content of a logic disappears in its interpretations—hence a simplification from which one might hope to derive powerful methods. In fact, the interesting aspect of Scott's method is not to interpret everything in a standard way (in $\mathcal{F}$, $\mathcal{N}_1$, $T_{\Sigma U}(\Omega)$ or $P\omega$). Rather, a semantic approach is taken, and, as for first-order logic, the power of this approach lies in the liberty of interpretation. Hence, other domains can be used, for example finite domains. The method of abstract interpretation fits in this framework.
it is often used for optimizing programs. An example from non-strict functional
language compilation is examined here: strictness analysis of a function, due to
A. Mycroft [113]. The case of recursive definitions is studied here.

Let \( D \) be the domain under which the standard interpretation of programs
is applied, for example \( \mathbb{N}_\perp \) and \( [\mathbb{N}_\perp \to \mathbb{N}_\perp] \). A recursive definition of \( f \) defines an
element \( f^D = \mu \Phi \), or \( f \), in \( [D^k \to D] \), as the least fixpoint of a continuous operator
\( \Phi \). Since it is impossible to decide in a static manner if \( f^D \) is strict, abstract
interpretations can be used to find sufficient conditions.

An abstract domain with two elements, \( \mathcal{D}^\# = \{0,1\} \), is introduced. Ordered
by \( 0 < 1 \), it is ‘abstract’ since it is much simpler than \( D = \mathbb{N}_\perp \). The abstraction
function \( \text{abs} : D \to D^\# \) is defined by

\[
\text{abs}(d) = \begin{cases} 
0 & \text{if } d = \perp \\
1 & \text{otherwise.}
\end{cases}
\]

\( D^\# \) must be made into a complete \( \Sigma \)-algebra. For \( u \in \Sigma \), write \( u^\# \) instead of \( u^{D^\#} \)
for the interpretation of \( u \), and \( [M]^\# \) for the interpretation of a term \( M \in T_\Sigma \). A
recursive definition yields a continuous operator \( \Phi^\# : [D^\# \to D^\#] \to [D^\# \to D^\#] \)
whose least fixpoint \( \mu \Phi^\# \) is written \( f^\# \).

The abstract interpretation is correct if for every \( i \), \( i \) is a strict index of \( f^D \) as
soon as \( i \) is a strict index of \( f^\# \):

\[
f^\#(a_1, \ldots, 0, a_{i+1}, \ldots, a_r) = 0 \Rightarrow f^D(d_1, \ldots, \perp, d_{i+1}, \ldots, d_r) = \perp.
\]

Conditions over \( D^\# \) and \( \text{abs} \) ensuring this correctness property are needed.

A condition to impose is

\[
\text{abs}(u(x_1, \ldots, x_n)) \leq_{D^\#} u^\#(\text{abs}(x_1), \ldots, \text{abs}(x_n))
\]

for every \( u \in \Sigma \), of arity \( n \) and \( x_1, \ldots, x_n \in D \).

This condition is always verified if \( u^D \) is strict (it is not necessary to choose \( u^\# \)
strict). However, it imposes conditions on \( u^\# \) if \( u^D \) is not strict. Consider the usual
sequential conditional

\[
\text{cond}(\perp, x, y) = \perp,
\]

\[
\text{cond}(\text{true}, X, \perp) = x,
\]

\[
\text{cond}(\text{false}, \perp, y) = y.
\]

The second equality implies \( x \leq \text{cond}^\#(1, x, 0) \) for \( x \in D^\# \), hence \( \text{cond}^\#(1, 1, 0) = 1 \)
and the third equality implies \( \text{cond}^\#(1, 0, 1) = 1 \). The other values of \( \text{cond}^\# \) can
be arbitrary.

This property implies

\[
\text{abs}([M]_\rho) \leq_{D^\#} [M]^\#(\text{abs} \circ \rho)
\]
for every term $M \in T_\Sigma[X]$ and $\rho : X \to D$ (the proof is easy by induction). In particular,
\[
\text{abs}(\llbracket F_k \rrbracket(x_i := d_i)) \leq_D^\# \llbracket F_k \rrbracket^\#(x_i := \text{abs}(d_i))
\]
for all the elements of the Kleene sequence of $f$. If it is also supposed that $\text{abs}$ is -
continuous, more exactly

\[
\text{abs}(\text{sup} \ d_n) \leq \sup \ \text{abs}(d_n)
\]
for every increasing sequence $(d_n)$ of $D$, it follows by going to the limit that

\[
\text{abs}(f^D(d_1, \ldots, d_r)) \leq_D^\# f^\#(\text{abs}(d_1), \ldots, \text{abs}(d_r)),
\]
and more generally that

\[
\text{abs}(\llbracket M \rrbracket \rho) \leq_D^\# \llbracket M \rrbracket^\#(\text{abs} \circ \rho),
\]
for every term $M \in T_{D \cup f}[X]$ and every environment $\rho : X \to D$.

So

\[
\text{abs}(f^D(d_1, \ldots, \perp, d_{i+1}, \ldots, d_r)) \leq f^\#(\text{abs}(d_1), \ldots, 0, \text{abs}(d_{i+1}), \ldots, \text{abs}(d_r)) = 0
\]
if $f^\#$ is strict in $i$ and if $\text{abs}(\perp) = 0$. If it is also supposed that $\text{abs}(d) = 0$ implies $d = \perp$, it follows that

\[
f^D(d_1, \ldots, \perp, d_{i+1}, \ldots, d_r) = \perp,
\]
which proves that $f^D$ is strict in $i$, and establishes the correctness of the abstract interpretation. So the following has been shown:

**Proposition 5.23** Let $D$ and $D^\#$ be two interpretations of $\Sigma$ and $\text{abs} : D \to D^\#$ a mapping satisfying:

1. $\text{abs}(d) = \perp^\#$ if and only if $d = \perp$;
2. $\text{abs}(u(x_1, \ldots, x_n)) \leq_D^\# u^\#(\text{abs}(x_1), \ldots, \text{abs}(x_n))$,
   where $u \in \Sigma$ and $x_1, \ldots, x_n \in D$;
3. $\text{abs}(\text{sup} \ d_n) \leq \sup \ \text{abs}(d_n)$ for every increasing sequence in $D$.

Then the interpretation $D^\#$ is correct, i.e., for every $i$, $i$ is a strict index of $f^D$ as soon as $i$ is a strict index of $f^\#$.

The problem is to find the best abstract interpretation which is correct. Obviously, the best would be to choose $D^\# = D$: no information would be lost, but then no static calculus could be done! In the usual case where $D^\# = \{0, 1\}$, $u^\#$ is chosen as follows:

\[
u^\#(a_1, \ldots, a_r) = \sup\{\text{abs}(u(d_1, \ldots, d_r)) ; \forall i \ \text{abs}(d_i) \leq a_i\}.
\]

Consider abstract interpretations in $D^\# = \{0, 1\}$. By equating $D^\#$ with the set of Booleans, a continuous function is represented by a combination of the connectives $\land$ and $\lor$, and each constant by $1$. The conditional is then interpreted by $(b, x, y) \mapsto b \land (x \lor y)$, and the arithmetic operations (which are strict) by $\land$. The conditions of the preceding proposition are verified.
Examples  Consider the (non-recursive) definition
\[ f(x, y) = \text{if } (x = 0) \text{ then } x + y \text{ else } x. \]

\[ f^\#(x, y) = (x \land 1) \land ((x \land y) \lor x) = x, \text{ hence } f^\#(0, y) = 0 \text{ and } f^\#(x, 0) = x. \]
So \( f^\# \) is strict(1), and non-strict(2). So the only deduction is that \( f \) is strict(1).

One can interpret the recursive definition of the factorial
\[ f(x) = \text{if } (x = 0) \text{ then } 1 \text{ else } x \ast f(x - 1) \]
in \( D^\# = \{0, 1\} \). So \( f^\#(x) = (x \land 1) \land (1 \lor (x \land f^\#(x \land 1))) \). This equation can be simplified to \( f^\#(x) = x \land f^\#(x) \). The least fixpoint of this equation is computed by iteration and is obtained on its first iteration: it is \( f^\#(x) = 0 \), which proves that \( f \) is strict. \[ \square \]

Exercises

1. Show that for every closed term \( t \), \( t^A = \iota_A(t) \), and that for every term \( t \),
   \( t^A(\xi) = \xi_B(t) \).

2. * Show that \{nor\} and \{nand\} are bases for connectives, and that they are
   the only bases formed of a unique binary connective.

3. * Show that \{\top, \to\} and \{\land, \lor\} are not bases for connectives.

4. Show using \( \models \) that the formula \( \exists y \forall x (R(y, x) \iff \neg R(x, x)) \) does not have a
   model. An application is Russell's barber problem.

5. Show that \( \models \exists x (\phi(x) \land \psi) \iff \exists x \phi(x) \land \psi \text{ if } x \notin \text{var}(\psi) \).

6. If \( A \) is a sub-\( \Sigma \)-algebra of \( B \) (define this concept, using as an example
   the definition of subgroups), and if \( \varphi \) is a universally quantified prenex
   proposition (i.e., with only \( \forall \)), then \( B \models \varphi \) implies \( A \models \varphi \). Deduce that if \( \Gamma \)
   is formed of universal propositions, a subalgebra of a model of \( \Gamma \) is a model
   of \( \Gamma \). Does this result apply to ring theory? to field theory?

7. If \( (A_n)_{n \geq 0} \) is an increasing sequence (for inclusion) of \( \Sigma \)-algebras, and if \( \Gamma \)
   is formed of prenex propositions of the form \( \forall \vec{x} \exists \vec{y} \psi \), \( \psi \) without quantifier,
   then \( A = \bigcup_{n \geq 0} A_n \) is the domain of a model \( A \) of \( \Gamma \). Does this result apply
   to field theory? to the theory of discrete ordered sets?

8. * Construct Kripke interpretations which do not satisfy the formulas
   \( (p \Rightarrow q) \lor (q \Rightarrow p) \), \( (p \Rightarrow (q \lor r)) \Rightarrow ((p \Rightarrow q) \lor (p \Rightarrow r)) \), \( (p \Rightarrow q) \Rightarrow (\neg p \lor q) \),
   'Peirce's axiom'.

---

2Hints to the exercises labeled with a * can be found at the end of the book.
9. Show that a Kripke frame \( U \) satisfies \( U \models p \lor \neg p \) for every interpretation in \( U \) if and only if the order over \( U \) is discrete, i.e., \( u \leq v \) implies \( u = v \).

10.* Let \( \varphi \) be a proposition of the language of fields. Show that if \( k \models \varphi \) for all the fields \( k \) of characteristic 0, then there exists an integer \( p \) such that \( k \models \varphi \) for any field \( k \) of characteristic greater than \( p \).

11. A constant \( c \) is added to the signature of arithmetic, as are the axioms \( 0 < c, \ s(0) < c, \ s(s(0)) < c, \ldots \) to those of (elementary) arithmetic. Show, using the compactness theorem, that the resulting theory has a model different from the standard model \( \mathbb{N} \).

12. Show that if \( \Gamma \) is a theory with arbitrarily large models, then \( \Gamma \) has an infinite model. Is the class of finite groups axiomatizable?

13. Show that a class \( \mathcal{E} \) of algebras is finitely axiomatizable if and only if there exists \( \varphi \) such that \( \mathcal{E} = \text{Mod}(\{ \varphi \}) \).

14.* Show that \( \mathcal{E} \) is finitely axiomatizable if and only if \( \mathcal{E} \) and \( \text{Alg}_{\Sigma} \setminus \mathcal{E} \) are axiomatizable.

15.* Show that a theory \( \Gamma \) is complete if and only if two of its arbitrary models are elementarily equivalent.

16. Show that \( \mathcal{A} \) is an elementary subalgebra of \( \mathcal{B} \) if and only if for every formula \( \varphi \in L_{\Sigma}[x, y_1, \ldots, y_n] \), if \( \varphi^B(b, a_1, \ldots, a_n) = 1 \) with \( b \in B \) and \( a_1, \ldots, a_n \in A \), there exists \( a \in A \) such that \( \varphi^A(a, a_1, \ldots, a_n) = 1 \).

17. If \( (\mathcal{A}_n)_{n \geq 0} \) is an increasing elementary sequence of \( \Sigma \)-algebras, i.e., each \( \mathcal{A}_n \) is an elementary subalgebra of \( \mathcal{A}_{n+1} \), then each \( \mathcal{A}_n \) is an elementary subalgebra of \( \mathcal{A} = \bigcup_{n \geq 0} \mathcal{A}_n \).

18. Prove lemma 5.12 for the universal case, starting from a prenex formula in conjunctive form.

19. Deduce from the universal case the 'dual' version of Herbrand's theorem, for an existential proposition \( \varphi \).

20.* Compute the set of clauses \( S(\varphi) \) for the formula \( \varphi \)

\[ \forall \varepsilon (\varepsilon > 0 \Rightarrow \exists \delta (\delta > 0 \land \forall x (|x - a| < \delta \Rightarrow |f(x) - f(a)| < \varepsilon))). \]

21. The same for

\[ \exists x \exists y (\forall w (\exists y (w = z + y) \Rightarrow (\exists u (u \cdot w = z) \Rightarrow w = 0 \lor w = 1)) \land \exists y (z = x + y)). \]
22. Prove equations 5.1–5.3 of page 173.

23.* Show that every consistent subset of the ordered set $F$ of partial functions has a least upper bound. Show as well that each subset has a greatest lower bound.

24. Show that the compact elements of $F$ are the partial functions of finite domain.

25.* Find all the continuous solutions (including the least fixpoint) in $[[\mathbb{N} \rightarrow \mathbb{N}) \rightarrow [\mathbb{N} \rightarrow \mathbb{N}]]$ of the equations

\[
\begin{align*}
f(x) &= f(x), \\
f(x) &= f(x + 1), \\
f(x) &= \text{if } x = 0 \text{ then } 1 \text{ else } f(x + 1).
\end{align*}
\]

26. Find the least element and the directed subsets of the Cartesian product of two flat orders.

27. Show that a function which is strict in each of its arguments, a constant function and the conditional function are all sequential.

28. Show that for $M, N \in T_{\mathbb{E}(\Omega)}$, $M \preceq_\Omega N$ if and only if for every occurrence $u$ of $M$, $M(u) = N(u)$ or $M(u) = \Omega$. Deduce that $M \preceq_\Omega N$ is equivalent to the existence of $u_1, \ldots, u_k \in \mathcal{O}(M)$ and of terms $P_1, \ldots, P_k$ such that $M(u_i) = \Omega$ and $N = M[u_1 \leftarrow P_1, \ldots, u_k \leftarrow P_k]$.

29. Show that every consistent subset of $T_{\mathbb{E}(\Omega)}^\omega$ has a least upper bound. Prove as well that every subset has a greatest lower bound, and that a partial term is compact if and only if it is finite.

30. Compute $\mathbb{N}x$ and $\varnothing x$, for $x \in P\omega$. In $P\omega$, if $[M] = \varnothing$, show that $[MN] = [\lambda x M] = \varnothing$.

31. Compute the interpretation of $\Omega$ in $P\omega$: 

- Show that if $n \in [\Omega]$, there exists a sequence of integers $p_i, i \geq 0$, such that $\langle p_{i+1}, n \rangle \in e_{p_i}$ and $e_{p_{i+1}} \subseteq e_{p_i}$.
- Deduce that if $n \in [\Omega]$, there exists $p$ such that $\langle p, n \rangle \in e_p$.
- Show using the standard encodings that this implies $[\Omega] = \varnothing$. 

Bibliographic notes

The semantics of classical logic is considered in, among others, [154]. Kripke interpretations are studied in [154, 150]. The semantics of the $\lambda$-calculus is treated in [8] and in the recent book by Krivine [90]. Stoy's book [143] presents the denotational semantics of programming languages. [59] discusses recursive definitions of programs. Lazy pairs in functional programming are introduced in [46]. The abstract interpretation method is due to P. and R. Cousot [30]. Many applications of this method are presented in [2].
6.1 Algebras and equations

Σ -algebras are the basic objects of semantics, as they occur in the predicate calculus, equational logic, logic programming, and even, with supplementary continuity conditions, in the semantics of computing.

The algebraic structures of mathematicians, such as (monoid, group, ring, ...), and the data structures (list, stack, record, ... ) of computer scientists can all be formalized as Σ-algebras, also called structures by logicians.

These algebras are studied in universal algebra, which examines the general properties of such structures from the same vantage point. The group concept avoids repeating a proof of the same property, for example \((xy)^{-1} = y^{-1}x^{-1}\), for \(\mathbb{Z}, S_3,\) or \(SU(2)\): not only is this property true for all groups, but the proof is the same regardless of the chosen group. Similarly, the algebra concept avoids the need to repeat proofs of certain structural properties, such as the kernel of a morphism (factorization of a morphism through a quotient, ... ) for groups, vector spaces or rings. To a great extent, the study of these general properties has been unified even more by the use of categories and functors (universal objects, adjunction, ... ), the vocabulary of which will sometimes be used here.

As new levels of abstraction are created, generalities with little depth can be ignored, so that one can concentrate on the particular cases, richer in content and difficulties. It is in fact in universal algebra that the first undecidable problems affecting the average mathematician were first discovered (the previous problems came more from metamathematics).

6.1.1 Modules in Standard ML

From a software engineering point of view, modularity is an essential attribute. It differentiates real languages such as Modula-2 and Common Lisp from school lan-
guages such as Pascal and Scheme. This is one reason for the ability of FORTRAN to survive, due to its quite rudimentary modularity (separate compilation), and the suspicion that was first held towards functional and logic languages, which had at first no concept of modularity. A modular structure meets several objectives: decomposition into simple elements, abstraction, separation into specification and implementation (ensuring independence from the choices of implementation and hardware), protection, separate compilation, reusability (use of libraries) and verification of compatibility, among others.

These objectives led to the design of two good modular languages: Modula-2, a descendant of Pascal, also designed by N. Wirth, and Ada, developed by a team led by J. Ichbiah. Another approach, motivated by modeling problems, can be found in object-oriented programming, whose concepts of class, instance and inheritance first appeared in K. Nygaard and O.J. Dahl’s Simula, before being popularized by A. Kay’s Smalltalk.

If modularity is an essential aspect of languages, the correctness of the resulting software—the most important objective from the user’s point of view—can only be ensured through a rigorous approach, using verification or proof techniques, either integrated into the language, such as a compiler’s type checker, or in production tools. There are two logical approaches to modularity: universal algebra and the \( \lambda \)-calculus’s type system, and they do not cohabit easily.

Universal algebra favors specification by axioms, models and general functorial constructions. This approach is normally used in specification languages, some of which, such as the OBJ family, also allow computation.

Typed \( \lambda \)-calculi offer powerful type systems with properties such as genericity, type inclusion and dependent types, particularly adapted to functional and, sometimes, object-oriented programming.

As for logic programming languages, in which ad hoc modular structures were added to ensure industrial credibility (Prolog-II, M-Prolog), it is only recently that a logical status of modularity has been sought (see [107], where modules are presented as an extension of the Horn clause language).

It seems for the moment that Standard ML has, at the programming language level, the most evolved concepts of modularity. This success probably comes from the fact that its constructions can be viewed either as algebras or as dependent types in a two-level type system. Its three modular constructions are signatures, structures and functors.

**Signatures and structures**

What is written in logic as ‘let \( \Sigma \) be a signature with two sorts \( s_1 \) and \( s_2 \), along with symbols \( a : s_1 \) and \( f : s_1 \times s_2 \rightarrow s_1 \)’, is written in ML as:

```ml
signature SIGMA =
  sig
    type s1 and s2
    val a : s1
```
and \( f : s_1 \times s_2 \rightarrow s_1 \)

end;

An example of a \( \Sigma \)-algebra \( A \), with two sets \( A_1 \) and \( A_2 \), of an element \( A_1 \) and a mapping from \( A_1 \times A_2 \) to \( A_1 \), would be written in ML as:

```ml
structure A =
  struct
    type s1 = int and s2 = bool
    val a = 3
    fun f(x,true) = x+1
        | f(x,false) = 0
  end;
```

and ML would answer as follows:

```ml
∀ structure A :

  sig
    eqtype s1
    eqtype s2
    val a : int
    val f : int * bool \rightarrow int
  end
```

In the terminology of language semantics, a structure is an environment, i.e., a set of bindings of names to values, which can be types, constants or functions, as in the preceding example, but also structures. ML’s structures resemble Modula-2’s IMPLEMENTATION MODULEs and the bodys of Ada’s packages. From the logical point of view, these are \( \Sigma \)-algebras. They are also the elements of a sum type, concept introduced by P. Martin-Löf in his Intuitionistic Type Theory, and one would write:

\[
\sum_{s_1,s_2: \text{Type}} [a : s_1, f : s_1 \times s_2 \rightarrow s_1].
\]

The components \( A.s_1, A.s_2, A.a, A.f \) of \( A \) can be reached directly by opening \( A \):

```ml
open A;
```

A signature declares a set of names of types, typed values and structures, and, like Modula-2’s DEFINITION MODULEs and the specifications of Ada’s packages, acts as the interface for a structure. ML signatures are similar to those in logic: they can be seen as the ‘type’ of a structure.

A data structure, formed of one or more object sorts and of operations manipulating them, must be put together in a module. Here is an example of a stack:

```ml
structure Stack =
  struct
    datatype 'a stack = empty | push of 'a * 'a stack
    fun pop(push(_,p)) = p
        and top(push(x,_)) = x
    end;
```
\( \text{std_in:19.5-20.26 Warning: match not exhaustive} \\
\text{push } (x,\_)) => \ldots \\
\text{std_in:19.5-20.26 Warning: match not exhaustive} \\
\text{push } (\_,p) => \ldots \\
\text{structure Stack} : \\
\text{sig} \\
\text{datatype } 'a \text{ stack} \\
\text{con empty : } 'a \text{ stack} \\
\text{con push : } 'a \times 'a \text{ stack } \rightarrow 'a \text{ stack} \\
\text{val pop : } 'a \text{ stack } \rightarrow 'a \text{ stack} \\
\text{val top : } 'a \text{ stack } \rightarrow 'a \\
\text{end} \\
\) 

An adequate signature for this structure is:

\( \text{signature STACK} = \\
\text{sig} \\
\text{datatype } 'a \text{ stack} = \text{empty} | \text{push of } 'a \times 'a \text{ stack} \\
\text{val pop : } 'a \text{ stack } \rightarrow 'a \text{ stack} \\
\text{and top : } 'a \text{ stack } \rightarrow 'a \\
\text{end} ; \\
\) 

Note that the signature contains the definition of the stack type and not only the declaration of its name, in order to export the constructors empty and push. Other signatures are equally adequate for this structure, which allows different interfaces to be designed by hiding certain components of the structure, which is why one refers to an adequate signature and not to the signature of a structure. Another adequate signature is:

\( \text{signature STACK1} = \\
\text{sig} \\
\text{datatype } 'a \text{ stack} = \text{empty} | \text{push of } 'a \times 'a \text{ stack} \\
\text{val pop : } 'a \text{ stack } \rightarrow 'a \text{ stack} \\
\text{end} ; \\
\) 

\( \text{signature STACK1} = \\
\text{sig} \\
\text{datatype } 'a \text{ stack} \\
\text{con empty : } 'a \text{ stack} \\
\text{con push : } 'a \times 'a \text{ stack } \rightarrow 'a \text{ stack} \\
\text{val pop : } 'a \text{ stack } \rightarrow 'a \text{ stack} \\
\text{end} \\
\) 

which restricts the exported operations (top disappeared):

\( \text{structure Stack1 : STACK1 = Stack ;} \)

In logic, this is the well-known operation of restriction: if \( \Sigma \subseteq \Sigma' \), every \( \Sigma' \)-algebra \( A \) is automatically a \( \Sigma \)-algebra written \( A \mid \Sigma \), obtained by only interpreting the elements of \( \Sigma \).
The complementary operation for restriction is extension. A test function is
added to the Stack structure. A new structure is created by importing the pre-
ceding one and by defining the new operation:

```
structure Stack2 =
    struct
        structure S = Stack
        fun isempty(S.empty) = true
        | isempty(S.push(_,_)) = false
    end;

structure Stack2 :
    sig
        structure S : ...
        val isempty : 'a Stack.stack -> bool
    end
```

An adequate signature for Stack2 is:

```
signature STACK2 =
    sig
        structure S : STACK
        val isempty : 'a S.stack -> bool
    end;

signature STACK2 =
    sig
        structure S : ...
        val isempty : 'a S.stack -> bool
    end
```

**Functors**

A functor is a ‘function’ associating a structure to a structure: it is used to define
generic structures.

ML’s type system allows the definition of polymorphic functions taking as
argument other functions, simulating Ada’s *generic packages* quite well. Functors
can be used to handle (data) structures as such collectively.

Consider the following mathematical formulation: ‘consider the lexical order $\leq^*$
over the free monoid $A^*$ constructed from an order $\leq$ over the alphabet $A$. A
structure $(A, \leq)$ is given, and a structure $(A, A^*, \leq, \leq^*)$ must be generated. The
‘type’ of what is given is the signature `RELATION` of ordered sets:

```
signature RELATION =
    sig
        eqtype t
        val order : t * t -> bool
    end;
```
and the ‘type’ of the resulting structure is:

```ml
signature BIRELATION =
  sig
    structure A : RELATION
eqtype m
    val lex : m * m -> bool
  end;
```

The functor which does the transformation is:

```ml
functor MyLex(X : RELATION) : BIRELATION =
  struct
    structure A = X
    eqtype m = (A.t) list
    fun lex([] , _  ) = true
      | lex(x::u , y::v) =
          if A.order(x,y)
            then if A.order(y,x) then lex(u,v) else true
            else false
  end;
```

When this functor is applied to the structure of ordered integers:

```ml
structure N =
  struct
    eqtype t = int
    fun order (x,y:t) = x>=y
  end;
```

the result is the structure of lexically ordered integer words:

```ml
structure Nstar = MyLex(N);
```

This ML formulation is as natural, and as precise, as the initial mathematical formulation. A good programming language must have this ‘faithful’ capacity, be it to represent mathematical constructions, as here, or physical systems, or, more generally, ‘knowledge’.

In algebra, a functor $F$ between two categories $\mathcal{X}$ and $\mathcal{Y}$ associates with each object $A$ of $\mathcal{X}$ an object $F(A)$ of $\mathcal{Y}$, and with each morphism $p : A \to B$ of the category $\mathcal{X}$ a morphism $F(p) : F(A) \to F(B)$ of $\mathcal{Y}$, satisfying the conditions $F(1_A) = 1_B$ and $F(p \circ q) = F(p) \circ F(q)$. Since there is no notion of morphism in ML, which is perhaps a defect, its functors act only on objects, i.e., structures, and they can be seen as functors from a category $\text{Alg}_\Sigma$ of $\Sigma$-algebras ($\Sigma$ being the signature of its argument structure) to another, $\text{Alg}_\mathcal{Y}$.

The example of the RELATION signature of ordered set shows, however, the limits of the module system: the signature was given for sets with a binary relation, not sets with an order relation. Furthermore, the functor MyLex produces a binary relation over words, but the compiler is incapable of guaranteeing that it is an order, even if it is ‘known’ that the component order of its argument is an order. A type
system would have to be devised in which the properties of objects can be given. A few experiments in this direction have been undertaken, but languages must be extended—possibly in the compiler—with proof techniques which are much more powerful than the ones used by existing type checkers. And of course, in the most general case, the static verification of these type–property pairs is undecidable. This problem calls into question the idea that a type is simply a set of values. So long as the examples of types are limited to the set of all integers, to Boolean strings or to arrays of pointers to Boolean records, this is not problematic, but it is when one wishes to refer to the types of even integers, or of character strings which are syntactically correct C programs or which form the definition of a non-ambiguous algebraic grammar. However, structures such as stacks, heaps and binary search trees, which are considered to be normal data structures, and which form the basis for the modular organization of software, are defined by properties of the same kind as even integers. Methods to handle them must be found.

6.1.2 \( \Sigma \)-algebras

After recalling the definitions of signatures, algebras and morphisms, as well as the principal algebraic constructions, a correspondence will be given between equational theories and classes of algebras.

\textit{Algebras, morphisms, products and subalgebras}

A \textit{signature} is a graded set \( \Sigma = \bigcup_n \Sigma_n \), where \( \Sigma_n \) is the set of (functional) symbols of arity \( n \). In the examples, the arithmetic signature \( \Sigma = \{0, s, +, *\} \) will often be used, as will that of group theory (neutral element \( e \), product written using juxtaposition, and inverse written \( -1 \)). A \( \Sigma \)-\textit{algebra} \( A \) consists of:

- a set \( A \), the \textit{domain} of \( A \);
- for each \( n \geq 0 \) and each \( f \in \Sigma_n \), of a mapping \( f^A : A^n \to A \), the \textit{interpretation} of \( f \).

A \( \Sigma \)-\textit{morphism} \( m : A \to B \) is a mapping of \( A \) to \( B \) such that for every \( n \) and every \( f \in \Sigma_n \), \( m(f^A(a_1, \ldots, a_n)) = f^B(m(a_1), \ldots, m(a_n)) \). The category of \( \Sigma \)-algebras and of \( \Sigma \)-morphisms is written \( \text{Alg}_\Sigma \). Algebras and morphisms will be mentioned without referring to the signature, should no ambiguity arise.

Note that, unlike the \( \Sigma \)-algebras of the predicate calculus, the domain of an algebra can be empty. This possibility can only arise in signatures with no constant symbols (\( \Sigma_0 = \emptyset \)), in which case the set \( T_\Sigma \) of closed terms is empty and this is the unique empty algebra. The situation will be different when many-sorted signatures will be treated, in which some of the domains might be empty.

How are algebras constructed? Given a signature \( \Sigma \) and a set \( X \), the algebra \( T_\Sigma[X] \) of \( \Sigma \)-terms with variables in \( X \) has already been presented.

Another simple construction is the product. Let \( A \) and \( B \) be two \( \Sigma \)-algebras. The \textit{product} \( \Sigma \)-algebra \( A \times B \) has as domain the product \( A \times B \) of the domains of
interpretation

\[ f^{A \times B}((a_1, b_1), \ldots , (a_n, b_n)) = (f^A(a_1, \ldots , a_n), f^B(b_1, \ldots , b_n)). \]

The projections of \( A \times B \) to \( A \) and \( B \) are morphisms. Similarly for the product \( \Pi_{i \in I} A_i \) of an arbitrary family \( (A_i)_{i \in I} \) of \( \Sigma \)-algebras. The particular case \( I = \emptyset \) yields a trivial algebra whose domain has a single element; these algebras being isomorphic, an arbitrary one may be designated \( 1_{\Sigma} \). It is terminal in the sense that for every \( \Sigma \)-algebra \( A \), there exists a unique \( \Sigma \)-morphism of \( A \) to \( 1_{\Sigma} \).

The dual property for ‘terminal’ is ‘initial’. The algebra \( T_{\Sigma} \) of closed terms is initial: for every \( \Sigma \)-algebra \( A \), there exists a unique \( \Sigma \)-morphism \( \iota_A \) from \( T_{\Sigma} \) to \( A \), the initial morphism.

\( B \) is a subalgebra of \( A \) if \( B \subseteq A \) and \( f^B(b_1, \ldots , b_n) = f^A(b_1, \ldots , b_n) \) for every \( f \in \Sigma_n \) and \( b \in B^n \). In other words, the injection of \( B \) to \( A \) is a morphism. For example, the image \( m(A) \) of a morphism \( m : A \to B \) is a subalgebra of \( B \).

The intersection of an arbitrary family of subalgebras of \( A \) is a subalgebra: this property makes the set of subalgebras into a complete lattice with inclusion. In particular, a family of subalgebras has a least upper bound. If \( E \) is an arbitrary subset of \( A \), the \( \Sigma \)-algebra generated by \( E \) is the intersection of the family of algebras containing \( E \). This family is non-empty since it contains \( A \). The lattice of subalgebras of \( A \) has a greatest element \( (A) \), and a least element (the minimal subalgebra of \( A \), the image of \( T_{\Sigma} \) by the initial morphism \( \iota_A \)).

An algebra is finitely generated, or of finite type, if there exists a finite subset \( E \) of \( A \) such that \( A \) is the algebra generated by \( E \); \( T_{\Sigma} \) is of finite type since it is generated by the empty set, and \( T_{\Sigma}[X] \) is of finite type if and only if \( X \) is finite.

**Congruences**

A congruence over a \( \Sigma \)-algebra \( A \) is an equivalence relation \( \approx \) over \( A \) compatible with the operations: for every \( f \in \Sigma_n \), if \( a_i \approx b_i \) for \( i = 1, \ldots , n \), then \( f^A(a_1, \ldots , a_n) \approx f^A(b_1, \ldots , b_n) \). The set \( A/\approx \) of equivalence classes is the domain of an algebra \( A/\approx \), the operations still working under the quotient because of the compatibility of \( \approx \). The natural surjection \( \pi_{\approx} \) of \( A \) to \( A/\approx \) is a morphism which will frequently be used in discussions.

**Examples**

- \( A^2 \) and \( \Delta = \{(a, a); a \in A\} \) are congruences, with \( A/A^2 = 1_{\Sigma} \) and \( A/\Delta = A \).
- A congruence \( \approx \) over a group \( G \) is determined by the class of the neutral element, which is a normal subgroup \( H_{\approx} \) of \( G \). \( a \approx b \) if and only if \( ab^{-1} \in H_{\approx} \).
- A congruence \( \approx \) over a ring \( A \) is determined by the class of 0, which is an ideal \( \pi_{\approx} \) of \( A \). \( a \approx b \) if and only if \( a - b \in \pi_{\approx} \).

In the two last examples, \( G/H_{\approx} \) is a group, and \( A/\pi_{\approx} \) is a ring, not only an algebra: this is not coincidental, as will be seen later.
The kernel of a morphism \( m : A \to B \) is
\[
\text{Ker } m = \{(x, y) \in A^2; m(x) = m(y)\}.
\]

As a binary relation over \( A \), Ker \( m \) is an equivalence. It is also a congruence, since if \((a_i, b_i) \in \text{Ker } m\), it is because \( m(a_i) = m(b_i) \). Hence
\[
m(f^A(a_1, \ldots, a_n)) = f^A(m(a_1), \ldots, m(a_n))
= f^A(m(b_1), \ldots, m(b_n))
= m(f^A(b_1, \ldots, b_n)),
\]
i.e., \((f^A(a_1, \ldots, a_n), f^A(b_1, \ldots, b_n)) \in \text{Ker } m\).

If \( \approx \) is a congruence, the kernel of the projection of \( A \) to \( A/\approx \) is precisely \( \approx \). As in linear algebra, every morphism \( m : A \to B \) is factorizable through the quotient by its kernel \( A/\text{Ker } m \), and its image \( m(A) \) as the composition of a natural surjection, an isomorphism, and inclusion:
\[
A \to A/\text{Ker } m \overset{\sim}{\to} m(A) \to B.
\]

The set of congruences over an algebra is also a complete lattice under inclusion, since the intersection of an arbitrary family of congruences is a congruence. It follows that there exists a congruence generated by a relation \( R \) (subset of \( A^2 \)), namely the intersection of the congruences containing \( R \). This lattice has the diagonal \( \Delta \) as least element, and \( A^2 \) as largest element.

Unlike for term algebras and product algebras, for which every element is easily represented in a machine, quotient algebras are not immediately representable. It is easy to represent an equivalence class by an element of that class, but there remains the problem of determining if two elements belong to the same class. Since this problem is not always decidable, this construction must be seen as an abstract definition, which is made concrete when (for example, rewriting) 'algorithms' are available.

**Free algebras, generators and relations**
The free algebra generated by \( X \) was constructed over the set \( T_\Sigma[X] \) of closed terms with variables in \( X \). Its fundamental property is that for every \( \Sigma \)-algebra \( A \) and for every mapping \( \xi : X \to A \), there exists a unique \( \Sigma \)-morphism, \( \hat{\xi}_\Sigma : T_\Sigma[X] \to A \), such that \( \hat{\xi}_\Sigma(x) = \xi(x) \) for every \( x \in X \), as seen in Figure 6.1.

The morphism \( \hat{\xi}_\Sigma \) is defined by:
\[
\hat{\xi}_\Sigma(x) = \xi(x)
\]
\[
\hat{\xi}_\Sigma(ft_1 \ldots t_n) = f^A(\hat{\xi}_\Sigma(t_1), \ldots, \hat{\xi}_\Sigma(t_n)).
\]
The subscript \( \Sigma \) is normally left out, as also is the ^ of \( \hat{\xi}_\Sigma \). The fact that \( T_\Sigma[X] \) is generated by \( X \) is exactly the definition by induction of the set of terms.
A universal problem This property of the free algebra has two characteristics found in other constructions: it is universal and relative.

The free algebra (or more precisely, the inclusion $X \hookrightarrow T_\Sigma[X]$) is a solution of the following universal problem: given a fixed set $X$, a fixed set $A$, and a mapping $f : X \rightarrow \mathcal{F}$ such that for every $\Sigma$-algebra $\mathcal{A}$ and for every mapping $\xi : X \rightarrow A$, there exists a unique morphism ('extension') $\hat{\xi} : \mathcal{F} \rightarrow \mathcal{A}$ satisfying $\xi \circ f = \hat{\xi}$.

A solution to this problem, if it exists, is necessarily unique up to isomorphism. So, let $f : X \rightarrow \mathcal{F}$ and $g : X \rightarrow \mathcal{G}$ be two solutions. Then there exist $\hat{g} : \mathcal{F} \rightarrow \mathcal{G}$ satisfying $\hat{g} \circ f = g$ and $\hat{f} : \mathcal{G} \rightarrow \mathcal{F}$ satisfying $\hat{f} \circ g = f$. The composition $\hat{f} \circ \hat{g}$ is a $\Sigma$-endomorphism of $\mathcal{F}$ satisfying $(\hat{f} \circ \hat{g}) \circ f = \hat{f} \circ g = f$. But the identity of $\mathcal{F}$ already satisfies this property, from which, by the required unicity of the extension of $f$, $\hat{f} \circ \hat{g}$ is the identity of $\mathcal{F}$. This proves that the two solutions $\mathcal{F}$ and $\mathcal{G}$ are isomorphic.

Many an abstract mathematical construction can be expressed as a solution of a universal problem: quotients, products, inductive and projective limits, complete metric spaces, and more. From a category-theoretic point of view, these solutions are always initial objects of appropriate categories, which allow powerful proof methods. However, the resulting objects are abstract, i.e., defined up to isomorphism, generally without an effective construction.

The other characteristic of the property is that it is 'relative', relating two kinds of structures, sets and algebras. A mapping $\xi$ is given and a $\Sigma$-morphism $\hat{\xi}_\Sigma$ is obtained. Relative concepts have also had much success in algebra and geometry. Two variants of the construction of $T_\Sigma[X]$ will be presented, where the relation algebra/set will be replaced by the relation $\Sigma$-algebra/$\Sigma^0$-algebra and by the relation $\mathcal{E}$-algebra/$\mathcal{E}^0$-algebra.

What happens if the construction already starts from a morphism between algebras? Let $\mathcal{A}$ be an algebra and form $T_\Sigma[A]$, the free $\Sigma$-algebra generated by the domain of $\mathcal{A}$. For example, the elements of $T_{\{0,s\}}[N]$ are terms with 'parameters' in $N$ such as $2$ and $ss4$, in addition to the closed terms $0$ and $sss0$. The relation stating the evaluation of these terms (such as $s3 = 4$) is the 'diagram' of $N$.

In general, there is an evaluation morphism $ev_\mathcal{A} : T_\Sigma[A] \rightarrow \mathcal{A}$ which is the extension of the identity of $\mathcal{A}$.
In other words,
\[
\begin{align*}
ev_A(a) &= a, \\
ev_A(f M_1 \ldots M_n) &= f^A(\ev_A(M_1), \ldots, \ev_A(M_n)).
\end{align*}
\]

The kernel of \( \ev_A \) is a congruence, called the (equational) diagram of \( A \).

**Example** The diagram of \( T_\Sigma \) is the diagonal \( \Delta \) of \( T_\Sigma \), the evaluation morphism being the identity. \( \square \)

Now, if \( \xi : A \to B \) is also a morphism of \( A \) in \( B \), its extension \( \hat{\xi} : T_\Sigma[A] \to B \) is nothing more than \( \xi \circ \ev_A \), since \( (\xi \circ \ev_A)(a) = \xi(a) \), and \( \hat{\xi} \) is the unique morphism satisfying this property.

**Presentations** The preceding construction shows that the set \( X \) appearing in \( T_\Sigma[X] \), whose elements are the variables in the syntax of terms, plays the role of generating an algebra. What is important is not a particular algebra \( T_\Sigma[X] \), such as \( T_\Sigma \), but the functor \( X \mapsto T_\Sigma[X] \), in the same way that the cosine function is much more interesting than any particular value of \( \cos x \).

This role is illustrated by the definition of an algebra by generators and relations. Start with an algebra \( \mathcal{A} \) and a set \( G \subseteq A \) generating it. Such a set always exists: one can always take \( G = A \), but the smaller, the more interesting. The inclusion \( \theta : G \hookrightarrow A \) extends to a morphism \( \hat{\theta}_\Sigma : T_\Sigma[G] \to A \). The image of \( \hat{\theta}_\Sigma \) is a subalgebra of \( \mathcal{A} \) containing \( G \); hence, by assumption on \( G \), the image is \( \mathcal{A} \). There is therefore a surjection \( T_\Sigma[G] \twoheadrightarrow \mathcal{A} \), and \( \mathcal{A} \) is isomorphic to the quotient of \( T_\Sigma[G] \) by the kernel of \( \hat{\theta}_\Sigma \): every algebra is therefore a quotient of a free algebra. In this construction, the elements of \( G \) appear more as constants than as variables.

The particular case where \( \mathcal{A} \) is generated by \( G = \emptyset \) is important: the initial morphism \( \iota_\mathcal{A} : T_\Sigma \to \mathcal{A} \) is then surjective and \( \mathcal{A} \) is isomorphic to \( T_\Sigma / \text{Ker} \,(\iota_\mathcal{A}) \). This surjectivity ensures that the elements of \( \mathcal{A} \) are all reachable, i.e., they can be designated by closed terms.

There is therefore an interesting method to define an algebra. A **presentation by generators and relations** of an algebra \( \mathcal{A} \) is a set \( G \) and a subset \( R \) of \( T_\Sigma[G] \times T_\Sigma[G] \) such that \( \mathcal{A} \) is isomorphic to \( T_\Sigma[G]/\sim \), where \( \sim \) is the congruence generated by \( R \). From above, every algebra has such a presentation (it suffices to take \( G = \mathcal{A} \) and for \( R \) the kernel of an evaluation morphism \( \ev_\mathcal{A} \)). This concept is particularly interesting in the case where \( G \) and \( R \) are finite sets: a method to define an algebra with finite means becomes possible, as for the definition of formal language, typically infinite, using a finite grammar. An algebra with a finite presentation is of finite type.

However, it is not always convenient to write finite presentations. Consider the case of \( B^* \), the set of binary words, i.e., over the alphabet \( B = \{0, 1\} \). It is a \( \Sigma \)-algebra for \( \Sigma \) formed of a constant symbol 0 (the empty word) and a binary symbol (concatenation). There are obviously two generators 0 and 1. The relations state that 0 is neutral (\( 0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 1 \cdot 1 = 1 \)) and that concatenation is associative.
$((00)0 = 0(00), (00)1 = 0(01), (01)0 = 0(10), \ldots)$. It would be preferable to write
one general law for associativity rather than eight particular laws. This will be
done by introducing equational propositions.

**Varieties and equalities**

To express general laws, a countably infinite set, distinct from the signature $\Sigma$
of variables is introduced, along with a binary relational symbol $=\,$ for equality.
Choosing $X$ to be infinite allows laws using an arbitrary number of variables. The
formulas to be manipulated are equational propositions, or equations, of the form
$\forall x_1 \ldots x_k (M = N)$, where $\{x_1, \ldots, x_k\} = \text{var}(M) \cup \text{var}(N)$. Since all formulas are
of this form, the quantifiers will always be omitted (only $M = N$ will be written).
They were made explicit here only for the purposes of justifying the semantics of
equations in the general framework of the predicate calculus. Hence associativity, a
property which was awkwardly expressed using several relations between generators
$((00)1 = 0(01), \ldots)$ to specify the free monoid $B^*$, can now be expressed using a
single equation between terms with variables: $(xy)z = x(yz)$. Here the variables
find their intuitive role of ‘metaterm’ standing for an arbitrary term. But semantics
gives them a stronger meaning, by making them stand for an arbitrary value.

An algebra $A$ is a model of, or satisfies, equation $M = N$, written $A \models M = N$,
if for every $\xi \in A^X$, $\xi(M) = \xi(N)$. The elements of $A^X$ are called valuations.

The examples will show that equations have two roles:

- to state properties, typically in an axiomatization of a class of structures,
such as that of groups:

$$
\begin{align*}
\text{(groups)} & \quad \begin{cases}
  ex &= x \\
  x^{-1}x &= e \\
  (xy)z &= x(yz);
\end{cases}
\end{align*}
$$

- to define operations in a manner close to that of a recursive program; here is
a definition of addition by induction, using the signature $\{0, s, +\}$:

$$
\begin{align*}
(+) & \quad \begin{cases}
  x + 0 &= x \\
  x + s(y) &= s(x + y).
\end{cases}
\end{align*}
$$

The set $\mathbb{N}$, with the usual interpretations of $0$, $s$, and $+$ is a model of $(+)$,
but there are others \ldots

**The correspondence $\text{Mod} \leadsto \text{Eq}$**

If $\mathcal{E}$ is a set of equations, $\text{Mod}(\mathcal{E})$ is the class
of algebras which are models of each of the equations of $\mathcal{E}$. The models of $\mathcal{E}$ are
also called $\mathcal{E}$-algebras, or $(\Sigma, \mathcal{E})$-algebras. Note that every set of equations has at
least one model: the trivial algebra $\mathbb{I}_\Sigma$. This was not the case for the predicate
calculus.

Conversely, with every class $\mathcal{C}$ of algebras and every set $X$ is associated the
equational theory of $\mathcal{C}$ over $X$, which is the set $\text{Eq}_X(\mathcal{C})$ of equations over $X$ which
are satisfied by all the algebras of $C$:

$$\text{Eq}_X(C) = \{(M, N) \in T_\Sigma[X] \times T_\Sigma[X]; A \models M = N \text{ for every } A \in C\}.$$  

$X$ will often be the countably infinite set of variables, but not necessarily. In particular $\text{Eq}_\omega(C)$, formed of equations between closed terms satisfied in $C$, is of interest. When it is not necessary to state $X$, simply write $\text{Eq}(C)$. In the case of a class formed of a single algebra $A$, write $\text{Eq}(A)$ instead of $\text{Eq}(\{A\})$.

These two transformations $\text{Mod}$ and $\text{Eq}$ satisfy the following obvious properties:

$$\mathcal{E}_1 \subseteq \mathcal{E}_2 \Rightarrow \text{Mod}(\mathcal{E}_2) \subseteq \text{Mod}(\mathcal{E}_1),$$  \hspace{1cm} (6.1)

$$\mathcal{C}_1 \subseteq \mathcal{C}_2 \Rightarrow \text{Eq}(\mathcal{C}_2) \subseteq \text{Eq}(\mathcal{C}_1),$$  \hspace{1cm} (6.2)

$$\mathcal{E} \subseteq \text{Eq}(\text{Mod}(\mathcal{E})),$$  \hspace{1cm} (6.3)

$$\mathcal{C} \subseteq \text{Mod}(\text{Eq}(\mathcal{C})).$$  \hspace{1cm} (6.4)

These four properties characterize the pair $\text{Mod} \supseteq \text{Eq}$ as a Galois correspondence between the sets ordered by inclusion of the sets of equations and the classes of algebras. They will often be used below, as will the following equations, which follow directly (exercise 5):

$$\text{Mod}(\text{Eq}(\text{Mod}(\mathcal{E}))) = \text{Mod}(\mathcal{E}),$$  \hspace{1cm} (6.5)

$$\text{Eq}(\text{Mod}(\text{Eq}(\mathcal{C}))) = \text{Eq}(\mathcal{C}),$$  \hspace{1cm} (6.6)

$$\text{Mod}(\bigcup_{i} \mathcal{E}_i) = \bigcap_{i} \text{Mod}(\mathcal{E}_i),$$  \hspace{1cm} (6.7)

$$\text{Mod}(\bigcap_{i} \text{Eq}(\text{Mod}(\mathcal{E}_i))) = \text{Mod}(\text{Eq}(\bigcup_{i} \text{Mod}(\mathcal{E}_i))).$$  \hspace{1cm} (6.8)

The name ‘Galois correspondence’ evidently came from the famous Galois theory, which establishes a correspondence between subgroups of the Galois group of a field extension and the intermediate fields of this extension.

**Definition 6.1** A set of equations $\mathcal{E}$ is an **equality** if there exists a class of algebras $\mathcal{C}$ such that $\mathcal{E} = \text{Eq}(\mathcal{C})$. A class of algebras is a **variety** if there exists a set of equations $\mathcal{E}$ such that $\mathcal{C} = \text{Mod}(\mathcal{E})$.

**Examples**

- The class of algebras is a variety, associated with $\mathcal{E} = \emptyset$.
- The class formed of trivial algebras (i.e., of which the domain has only one element), and possibly of the empty algebra, is the variety associated with an equation $x = y$.
- Algebraic structures give numerous familiar and non-trivial examples of varieties: monoids, groups, abelian groups, rings, Boolean algebras, etc.
However, the class of fields is not a variety. This does not come as a surprise, since the usual axioms of field theory include a formula $\forall x(x \neq 0 \Rightarrow \exists y(xy = 1))$ which is not equational, but this does not imply the non-existence of equational axioms for fields. For example, the (non-equational) axiom $\forall x\exists y(xy = 1)$ is sometimes given for groups, but it does not prevent the class of groups from being a variety. Other examples will be given for the specification of data structures in computer science.

The properties $\text{Mod}(\text{Eq}(\text{Mod}(\mathcal{E}))) = \text{Mod}(\mathcal{E})$ and $\text{Eq}(\text{Mod}(\text{Eq}(\mathcal{C}))) = \text{Eq}(\mathcal{C})$ show that for every equality $\mathcal{E}$, and for every variety $\mathcal{C}$,

$$\text{Mod}(\text{Eq}(\mathcal{C})) = \mathcal{C},$$

$$\text{Eq}(\text{Mod}(\mathcal{E})) = \mathcal{E}.$$

So, the transformations $\text{Mod}$ and $\text{Eq}$ establish a bijection between equalities and varieties. Similar definitions can be given in the general framework of the predicate calculus, where one would refer to complete theories and axiomatized classes, and they also lead to a bijection. In the equational case, there are also simple characterizations of equalities and varieties, using stability properties.

**Equalities and stable congruences** Let $\mathcal{E} = \text{Eq}(\mathcal{C})$ be an equality. First, it is a congruence, since its definition can be reformulated as

$$\text{Eq}(\mathcal{C}) = \bigcap_{\mathcal{A} \in C} \bigcap_{\xi \in A^X} \text{Ker} \hat{\xi},$$

and the kernel of a morphism is a congruence, as is the intersection of congruences. One can then write $\text{Eq}(\mathcal{C}) = \text{Ker} \theta_\mathcal{C}$ where $\theta_\mathcal{C}$ is the morphism

$$\theta_\mathcal{C} : T_\Sigma[X] \rightarrow \prod_{\mathcal{A} \in C} \prod_{\xi \in A^X} \mathcal{A},$$

$$M \mapsto ((\hat{\xi}(M))_{\xi \in A^X})_{\mathcal{A} \in C}.$$

Equalities are not only congruences, they also satisfy the following stability property:

**Definition 6.2** A binary relation $R$ over $T_\Sigma[X]$ is stable under substitutions if $(M, N) \in R$ implies $(\hat{\sigma}M, \hat{\sigma}N) \in R$, for every substitution $\sigma : X \rightarrow T_\Sigma[X]$.

That $\text{Eq}(\mathcal{C})$ be stable under substitutions follows intuitively from the universal quantification of equational propositions: if $(M, N) \in \text{Eq}(\mathcal{C})$ and $\sigma$ is a substitution, $(\hat{\sigma}M, \hat{\sigma}N) \in \text{Eq}(\mathcal{C})$ must be proven, i.e., it must be proven that $\hat{\xi}(\hat{\sigma}M) = \hat{\xi}(\hat{\sigma}N)$ for every $\xi \in A^X$, which follows from $\hat{\xi} \circ \hat{\sigma} = \hat{\xi} \circ \sigma$.

**Proposition 6.1** Let $\mathcal{E}$ be a set of equations and $\mathcal{E}'$ be the congruence stable under substitutions generated by $\mathcal{E}$. Then $\mathcal{E}$ and $\mathcal{E}'$ have the same models and $\mathcal{E}'$ is the equational theory of the models.
Proof If \( \mathcal{E} \) is an arbitrary set of equations, then the equality \( \mathcal{E}' = \text{Eq}(\text{Mod}(\mathcal{E})) \) is a congruence stable under substitutions. According to the general properties of the correspondence \( \text{Mod} \cong \text{Eq} \),

\[
\text{Mod}(\mathcal{E}') = \text{Mod}(\text{Eq}(\text{Mod}(\mathcal{E}))) = \text{Mod}(\mathcal{E}) ;
\]

\( \mathcal{E} \) and \( \mathcal{E}' \) therefore have the same models.

Conversely, it will be shown that a congruence \( \mathcal{E} \) over \( T_\Sigma[X] \), stable under substitutions, is an equality, i.e., an equational theory. A class \( \mathcal{C} \) of algebras must be found such that \( \mathcal{E} = \text{Eq}(\mathcal{C}) \). Obviously, \( \mathcal{C} = \text{Mod}(\mathcal{E}) \). Form the quotient \( T_\Sigma[X]/\mathcal{E} \), written \( T_{\Sigma,\mathcal{E}}[X] \), or \( T_\mathcal{E}[X] \), and let \( \varpi : T_\Sigma[X] \to T_\mathcal{E}[X] \) be the natural surjection.

First, it must be shown that \( T_\mathcal{E}[X] \) is a model of \( \mathcal{E} \). Let \( (M, N) \in \mathcal{E} \) and \( \xi : X \to T_\mathcal{E}[X] \). The morphism \( \hat{\xi} \) factors through \( T_{\Sigma,\mathcal{E}}[X] \), by using an arbitrary right inverse (or section) \( s \) of the surjection \( \varpi \). (There is no reason to believe that the inverse is a morphism. It is simply supposed that the composed mapping \( \varpi \circ s \) is the identity of \( T_\mathcal{E}[X] \).) Define, then, \( \eta = s \circ \xi : X \to T_\Sigma[X] \). Its extension \( \hat{\eta} \) satisfies \( \varpi \circ \hat{\eta} = \hat{\xi} \), by the unicity of \( \hat{\xi} \) since \( \varpi \circ \eta = \varpi \circ s \circ \xi = \xi \).

Hence, \( \hat{\xi}(M) = \varpi(\hat{\eta}(M)) = \varpi(\hat{\eta}(N)) = \hat{\xi}(N) \), since \( \mathcal{E} \) is stable under substitutions and \( (\hat{\eta}(M), \hat{\eta}(N)) \in \mathcal{E} \) if \( (M, N) \in \mathcal{E} \). This proves \( T_\mathcal{E}[X] \models \mathcal{E} \).

To prove that \( \mathcal{E} \) is an equality, it suffices to show the inclusion \( \text{Eq}(\text{Mod}(\mathcal{E})) \subseteq \mathcal{E} \), since the converse inclusion always holds. Let \( (M, N) \in \text{Eq}(\text{Mod}(\mathcal{E})) \). Since it has been shown that \( T_\mathcal{E}[X] \) is a model of \( \mathcal{E} \) if \( \mathcal{E} \) is stable under substitutions, then in particular \( T_\mathcal{E}[X] \models M = N \). Note that if \( \xi : X \to T_\mathcal{E}[X] \) associates its class modulo \( \mathcal{E} \) with each variable, then \( \hat{\xi} \) is the natural surjection \( \varpi \). So, \( T_\mathcal{E}[X] \models M = N \) implies \( \varpi(M) = \varpi(N) \), i.e., \( (M, N) \in \mathcal{E} \), which proves \( \text{Eq}(\text{Mod}(\mathcal{E})) \subseteq \mathcal{E} \).

Let \( \mathcal{E} \) be an arbitrary set of equations, and \( \mathcal{E}_1 \) be an equality containing \( \mathcal{E} \). From \( \mathcal{E} \subseteq \mathcal{E}_1 \), it follows that \( \text{Eq}(\text{Mod}(\mathcal{E})) \subseteq \text{Eq}(\text{Mod}(\mathcal{E}_1)) \). But, \( \mathcal{E}_1 \) being an equality, \( \text{Eq}(\text{Mod}(\mathcal{E}_1)) = \mathcal{E}_1 \). On the other hand, it was seen that \( \mathcal{E}' = \text{Eq}(\text{Mod}(\mathcal{E})) \) is an equality. It follows that \( \text{Eq}(\text{Mod}(\mathcal{E})) \) is the least equality containing \( \mathcal{E} \): it is the equality generated by \( \mathcal{E} \). It is also, according to the given characterization of equalities, the least congruence stable under substitutions containing \( \mathcal{E} \). \( \square \)
Varieties and stable classes In the same manner that equalities were characterized as congruences stable under substitutions, varieties are the classes of algebras satisfying certain stability properties.

Definition 6.3 A class $C$ of algebras is called

- **stable under isomorphisms** if $C$ contains every algebra which is isomorphic to an algebra in $C$;
- **stable under subalgebras** if $C$ contains all the subalgebras of its algebras;
- **stable under quotients** if $C$ contains every algebra which is a quotient (by a congruence) of an algebra in $C$;
- **stable under products** if for every family $A_i$ of algebras in $C$, the product $\Pi_{i \in I} A_i$ is in $C$.

It is easy to prove that every variety is stable under isomorphisms, subalgebras, quotients and products (exercise 6).

In the case of the variety of groups, these properties are well-known: a subalgebra of a group, a quotient of a group (by a particular subgroup) and a product of groups are all groups, not just algebras. Stability under subalgebras is a property of the models of universal theories. Stability under products is a property of models of Horn theories.

G. Birkhoff showed, conversely, that a class of algebras $C$, stable under isomorphisms, subalgebras, quotients and products, is a variety. If $C$ satisfies these stability properties, a set of equations $\mathcal{E}$ must be constructed such that $C = \text{Mod}(\mathcal{E})$.

An infinite set $X$ having been chosen, the equational theory $\mathcal{E} = \text{Eq}_X(C)$ is formed, followed by the quotient $T_\mathcal{E}[X]$. By definition, $\text{Eq}_X(C) = \text{Ker} \ \theta_C$ where $\theta_C$ is the morphism from $T_\Sigma[X]$ to the product algebra $\Pi C = \Pi_{A \in C} \Pi_{x \in A} A$, associating the family $((\hat{\xi}(M))_{\xi \in \text{Ax}})_{A \in C}$ with a term $M$. By passing through the quotient by $\mathcal{E}$, an isomorphism of $T_\mathcal{E}[X]$ to a subalgebra $\mathcal{U}$ of $\Pi C$ is obtained. By assumption, $C$, being stable under products, contains $\Pi C$. Being stable under subalgebras, it contains $\mathcal{U}$, a subalgebra of $\Pi C$. Being stable under isomorphisms, it contains $T_\mathcal{E}[X]$, which is isomorphic to $\mathcal{U}$.

So $C$ has been shown to contain $T_\mathcal{E}[X]$, where $\mathcal{E}$ is the equational theory of $C$. It remains to be shown that $\text{Mod}(\mathcal{E}) \subseteq C$. Let $A$ be a model of $\mathcal{E}$. Choose a set $Y$, containing $X$, large enough that there exists a surjection $\eta$ from $Y$ to $A$, hence a surjective morphism from $T_\Sigma[Y]$ to $A$. But it is easy to see that $\text{Mod}(\text{Eq}_X(C)) \subseteq \text{Mod}(\text{Eq}_Y(C))$ if $X \subseteq Y$.

Hence, $A$, which is a model of $\mathcal{E} = \text{Eq}_X(C)$, is also a model of $\text{Eq}_Y(C)$. But, $T_\Sigma[Y]/\text{Eq}_Y(C)$ is the free model generated by the $Y$ of the theory $\text{Eq}_Y(C)$. There is therefore a (unique) extension $\hat{\eta} : T_\Sigma[Y]/\text{Eq}_Y(C) \to A$ of $\eta$, which is surjective, like $\eta$. What was proved for $T_\mathcal{E}[X]$ also holds for the algebra $T_\Sigma[Y]/\text{Eq}_Y(C)$: it is contained in $C$. So, there is a surjective morphism $\eta$ from an algebra of $C$ to $A$, hence an isomorphism between $A$ and a quotient (by the kernel of $\eta$) of an algebra of $C$. By using the assumptions of stability under isomorphisms and quotients once
again, it follows that \( \mathcal{A} \) is contained in \( \mathcal{C} \). This proves that \( \text{Mod}(\mathcal{E}) \subseteq \mathcal{C} \), and, with the inverse inclusion, that \( \mathcal{C} \) is the variety \( \text{Mod}(\mathcal{E}) \).

To conclude, here is Birkhoff's theorem:

**Theorem 6.2 (Birkhoff stability)**  
1. A binary relation over \( T_\Sigma[X] \) is an equality, i.e., the equational theory of a class of algebras, if and only if it is a congruence that is stable under substitutions.
2. A class of algebras is a variety, i.e., the class of models of a set of equations, if and only if it is stable under isomorphisms, subalgebras, quotients and products.

This theorem, difficult to prove, is easily applied, especially to prove that a given class of algebras is not a variety. For example, the class of fields cannot be a variety, since a subalgebra (in particular, a subring) of a field is not necessarily a field. It is therefore useless to look for an equational definition of fields.

**Many-sorted signatures**
Recall a definition given in Chapter 1. A *many-sorted signature* is a pair \((S, \Sigma)\), where

- \( S \) is a set of *sort* symbols;
- \( \Sigma \) is a set of typed constant and function symbols:

\[
\begin{align*}
  c &: s \\
  f &: s_1 \ldots s_n \rightarrow s,
\end{align*}
\]

where \( s_1, \ldots, s_n, s \in S \) and \( n \geq 1 \).

An \( S \)-set is a family \( A = (A_s)_{s \in S} \) of sets indexed by \( S \), and an \( S \)-mapping \( m : A \rightarrow B \) between two \( S \)-sets \( A \) and \( B \) is a family of mappings \( m_s : A_s \rightarrow B_s \), \( s \in S \). Introduce an \( S \)-set \( X = (X_s)_{s \in S} \) of variables. The rules for term formation are:

1. a variable of sort \( s \) is a term of sort \( s \);
2. a constant of sort \( s \) is a term of sort \( s \);
3. if \( f \) is a symbol of type \( s_1 \ldots s_n \rightarrow s \) and if \( M_i \) is a term of sort \( s_i \), \( i = 1, \ldots, n \), then \( fM_1 \ldots M_n \) is a term of sort \( s \).

The set of terms of sort \( s \) is \( T_\Sigma[X]_s \) and the set of all terms is \( T_\Sigma[X] = \bigcup_{s \in S} T_\Sigma[X]_s \).

From a semantics point of view, each sort \( s \in S \) is interpreted by a set (its domain) \( A_s \). An \((S, \Sigma)\)-algebra (or \( \Sigma \)-algebra) \( \mathcal{A} \) is an \( S \)-set \( A \) with, for each symbol \( f : s_1 \ldots s_n \rightarrow s \) of \( \Sigma \), a mapping \( f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s \). A \( \Sigma \)-morphism \( m : A \rightarrow B \) is an \( S \)-mapping commuting with the interpretations of the elements of \( \Sigma \). All the definitions and properties given in this section are also valid for many-sorted algebras. The same notations are used for \( T_\Sigma[X] \), the \( \Sigma \)-algebra.
freely generated by an S-set \( X \), and \( T_\Sigma \), the \( \Sigma \)-algebra of closed terms, which is an initial \( \Sigma \)-algebra.

Many-sorted signatures give a natural framework to describe algebraic structures such as vector spaces, which have two sorts of objects: scalars and vectors. The specification of data structures is another application domain, which has been studied in detail since the middle of the 1970s.

\textit{Example}  The specification of a set structure will be given by a signature with two sorts \textit{element} and \textit{set}, containing a constant and an insertion operation of an element in a set:

\[
\begin{align*}
\varnothing & : \text{set} \\
\# & : \text{element} \times \text{set} \rightarrow \text{set}
\end{align*}
\]

and the axioms

\[
(\#) \quad \begin{cases} 
  a\#(a\#x) = a\#x \\
  a\#(b\#x) = b\#(a\#x).
\end{cases}
\]

\[\square\]

\subsection{The inference system of equational logic}

\textit{From equality axioms to rules}

From a deductive point of view, consider a set \( \mathcal{E} \) of equational axioms and the equational propositions derived from \( \mathcal{E} \).

Consider a general inference system, such as the NK system of natural deduction. Axioms, already given, must be added to state that \( = \) is the equality:

\[
\begin{align*}
(A_{x=1}) & \quad \forall x \ x = x \\
(A_{x=2}) & \quad \forall xy \ x = y \Rightarrow y = x \\
(A_{x=3}) & \quad \forall xyz \ x = y \land y = z \Rightarrow x = z \\
(A_{x=f}) & \quad \forall x_1 \ldots x_n y_1 \ldots y_n \ x_1 = y_1 \land \ldots \land x_n = y_n \Rightarrow f_{x_1} \ldots_{x_n} = f_{y_1} \ldots y_n
\end{align*}
\]

Apart from the first, these axioms are not all equational. However, they all have a restricted form: they are equational Horn clauses, also called conditional clauses, which share with equations a certain number of properties which are found in logic programming.

To manipulate them, the rules of NK dealing with the connectives \( \Rightarrow \) and \( \land \) and the quantifier \( \forall \) are used. It follows that formulas which are not equations must be used to prove an equation from other equations. This detour is anything but natural. To avoid it, a solution consists of combining the use of equality axioms and the rules of NK in the derivation of new specialized rules. Here, for example, is how the symmetry rule

\[
(S) : \quad \forall \overline{x}(M = N) \ \Rightarrow \ \forall \overline{x}(N = M)
\]
can be derived in NK:

\[
\begin{align*}
(E_v) : & \quad \forall \bar{\alpha} (M = N) \\
& \quad \frac{M = N}{\forall \bar{\alpha} (M = N)} \quad (Ax_{=2}) : \quad \forall x y (x = y \Rightarrow y = x) \\
(E_{=}) : & \quad \frac{\forall \bar{\alpha} (M = N)}{M = N} \\
(I_{=}) : & \quad \frac{\forall \bar{\alpha} (N = M)}{N = M}
\end{align*}
\]

From semantics to rules

The second approach is semantic and exploits the results obtained. Instead of deriving the rules in a general inference system such as NK, without any intuition, an inference system will be constructed. The relation \( \vdash \) of deductive consequence will be equivalent to the relation \( \models \) of semantic consequence: \( \mathcal{E} \vdash M = N \) if and only if \( \mathcal{E} \models M = N \). This equivalence will be stated as the completeness and the soundness of the inference system.

Fix an infinite set \( X \) of variables for what follows, and let \( \mathcal{E} \) be a set of equations over these variables. By using the correspondence \( \text{Mod} \equiv \text{Eq} \), \( \mathcal{E} \vdash M = N \) means exactly \( M = N \in \text{Eq}(\text{Mod}(\mathcal{E})) \). Then let \( \mathcal{E}' = \text{Eq}(\text{Mod}(\mathcal{E})) \): it was seen that \( \mathcal{E} \) and \( \mathcal{E}' \) have the same models, that \( \mathcal{E}' \) is the equality generated by \( \mathcal{E} \), i.e., from the characterization of equalities, \( \mathcal{E}' \) is the least congruence stable under substitutions containing \( \mathcal{E} \).

The problem comes back to finding the equality generated by a set of equations, which is of purely syntactic nature. The rules of Table 6.1 define the generated stable congruence exactly. Write \( \vdash_{eq} \) for the deduction relation defined by this inference system.

\[
\begin{align*}
(R) : & \quad \frac{M = M}{M = M} \quad (S) : \quad \frac{M = N}{N = N} \\
(T) : & \quad \frac{M = M' \quad M' = M''}{M = M''} \\
(\text{cong}) : & \quad \frac{M_1 = N_1 \quad \cdots \quad M_r = N_r}{f M_1 \cdots M_r = f N_1 \cdots N_r} \\
(\text{subst}) : & \quad \frac{M = N}{\sigma(M) = \sigma(N)}
\end{align*}
\]

Table 6.1 The inference system of equational logic

The following lemma is easily checked, and it leads directly, along with the preceding discussion, to Birkhoff's completeness theorem.

Lemma 6.3 If \( \mathcal{E}' \) is the equality generated by \( \mathcal{E} \), then \((M, N) \in \mathcal{E}'\) if and only if \( \mathcal{E} \vdash_{eq} M = N \), i.e., the judgment \( M = N \) is derivable from the axioms \( M = N \), for \((M, N) \in \mathcal{E} \).
Theorem 6.4 (Birkhoff completeness) \( \mathcal{E} \vdash_{eq} M = N \) if and only if \( \mathcal{E} \models M = N \).

It will be convenient to use a more general rule for context passing which can be derived from the preceding ones. It operates over an arbitrary context \( C[\ ] \), i.e., a term with a 'hole', in contrast to the rules \((f, i)_{1 \leq i \leq n}\), which deal only with 'superficial' contexts:

\[
\text{(cont) : } \frac{M = N}{C[M] = C[N]}.
\]

The many-sorted case

The many-sorted case is not identical to the single-sorted one from the deductive point of view. If the preceding inference system is applied directly, and if the signature allows certain interpretation domains to be empty, invalid equations might be derivable, making the inference system unsound, and this would be unacceptable.

Example Consider a signature with two sorts \( s \) and \( s' \) with a symbol \( f : s \rightarrow s' \) and the equation \( f(x) = y \). From \( f(x) = y \), deduce \( f(x) = z \) using the substitution \([y := z]\); then by symmetry and transitivity, \( y = z \). Consider the algebra \( A \) with \( A_s = \emptyset \) and \( A_{s'} = \{a, b\} \), the interpretation of \( f \) being the empty function from \( A_s \) to \( A_{s'} \) (there is no other choice for \( f^A \)). So \( A \models f(x) = y \); a universal proposition is always true in an empty domain! But \( A \not\models y = z \) since \( a \neq b \). The inference \( f(x) = y \vdash y = z \) is therefore unsound. \( \Box \)

This situation cannot occur in the single-sorted case, since all the equations are valid in the empty algebra. In the many-sorted case, either a restriction must be made to signatures so that empty domains are not allowed, or the inference system must be modified so that it no longer deduces invalid equations.

It is very convenient to use algebras, some of whose domains can be empty. For example, the initial algebra over the signature of lists has the empty set for domain of sort element. This is perfectly normal, since a list needs generators to exist and the initial algebra has no generator.

The inference system will therefore be modified; the first part to be modified is the form of judgments. For each sort \( s \) of the signature introduce a predicate \( H_s \) stating that the sort \( s \) is inhabited, i.e., \( H_s \) means that \( \exists x : s \). This existential in an equational framework is not problematic, since the \( \exists \) will have a negative sign in the judgment, hence a universal force.

Judgments are guarded equations

\[ H_{s_1}, \ldots, H_{s_p} \Rightarrow M = N. \]

Such a judgment is written \( H \Rightarrow M = N \), where \( H \) is its guard, \( M = N \) the equation and \( s_1, \ldots, s_p \) are the guarded sorts. A variable, whether or not it appears in the equation, is guarded if its sort is guarded.

So a new form of satisfaction of judgments is defined:

\[
A \models H_{s_1}, \ldots, H_{s_p} \quad \text{if} \quad A_{s_i} \neq \emptyset \text{ for } i = 1, \ldots, p.
\]
Then
\[ \mathcal{A} \vdash H \Rightarrow M = N \quad \text{if} \quad \mathcal{A} \vdash H \text{ implies } \mathcal{A} \vdash M = N. \]

The axioms of an equational theory must be written under a guarded form, by guarding the sorts of all the variables. Since this guard is systematic, it can be considered as implicit—for the axioms only.

Here are the rules:

(\text{R}) : \quad \frac{H \Rightarrow M = M'}{H \Rightarrow M = M'}

where \( H \) is formed of guards over the sorts of variables of \( M' \);

(\text{S}) : \quad \frac{H \Rightarrow M = N}{H \Rightarrow N = M}, \quad (\text{T}) : \quad \frac{H \Rightarrow M = N \quad H \Rightarrow N = P}{H \Rightarrow M = P},

(\text{subst}) : \quad \frac{H \Rightarrow M = N}{H' \Rightarrow \sigma(M) = \sigma(N)},

where \( H' \) is formed of guards over the sorts of the variables of the image by \( \sigma \) of the guarded variables of the premise;

(\text{cont}) : \quad \frac{H \Rightarrow M = N}{H' \Rightarrow C[M] = C[N]},

where \( H' \) is \( H \) extended by the guards over the sorts of the variables figuring in the context \( C[\ ] \).

Example (\text{cont.}) Start from the axiom \( H_s, H_s' \Rightarrow f(x) = y \). Deduce by substitution \( \frac{f(x)}{z} \): \( H_s, H_s' \Rightarrow f(x) = z \) (the guard \( H_s \) has no reason to disappear, even if no other variable is of sort \( s \) in the inferred equation, since the variables of sort \( s \) are guarded in the premise and invariant by this substitution), then by symmetry and transitivity, \( H_s, H_s' \Rightarrow y = z \), always with the same guard. From the semantic point of view, \( \mathcal{A} \models H_s, H_s' \Rightarrow f(x) = y \) for the simple reason that \( \mathcal{A} \not\models H_s \), i.e., \( A_s = \emptyset \), and similarly for the formulas that are derived from it: \( \mathcal{A} \models H_s, H_s' \Rightarrow y = z \). On the other hand, \( \mathcal{A} \not\models H_s' \Rightarrow y = z \), but this judgment was not inferred.

The guard is conserved or extended by (\text{R}), (\text{S}), (\text{T}), and (\text{cont}). One would want to infer judgments with the smallest possible guard. Only the rule (\text{subst}) can reduce the guard (this rule can also extend it).

A weakening rule can be derived:

\text{(weak)} : \quad \frac{H \Rightarrow M = N}{H, H_s \Rightarrow M = N}, \quad (6.9)

which is used essentially so that the two premises of the rule (\text{T}) have the same guard. It is more interesting to have a reinforcement rule:

\text{(reinf)} : \quad \frac{H_s, H \Rightarrow M = N}{H \Rightarrow M = N},
which eliminates an unnecessary guard, i.e., a guard over a sort \( s \) for which no variable of \( M = N \) is of that sort. Suppose that there exists a closed term \( P \) of sort \( s \), and apply the rule (\text{subst}) with the substitution \( \sigma \) associating \( P \) with all the variables of sort \( s \). Then \( \sigma \) leaves \( M \) and \( N \) invariant, so that the conclusion is \( H \Rightarrow M = N \). The same manipulation can be done by using for \( P \) a term of sort \( s \) whose variables are of sorts already guarded by \( H \).

\textbf{Example} Consider a signature with two sorts \( s \) and \( s' \), with two unary symbols \( f : s \to s' \) and \( g : s' \to s \) (and no constant, which means that the initial algebra has its two empty domains). From \( H_s, H_{s'} \Rightarrow f(x) = y \), it follows as for the previous example that \( H_s, H_{s'} \Rightarrow y = z \). Now, use the substitution \( \sigma \) associating with every variable of sort \( s \) the term \( g(u) \), where \( u \) is a variable of sort \( s' \). By applying (\text{subst}), infer \( H_{s'} \Rightarrow y = z \), since the image by \( \sigma \) of the guarded variables of \( H_s, H_{s'} \Rightarrow y = z \) (i.e. the variables of sort \( s \) and \( s' \), including \( y \) and \( z \)) is reduced to \( y, z \) and \( u \), all of sort \( s' \). The unnecessary guard \( H_s \) has been eliminated. \( \square \)

Goguen proposed a condition over a signature allowing the elimination of unnecessary guards, and thereby allowing work with an unguarded inference system, which is more pleasant: for every pair of sorts \( s, s' \), \((T_E[x_s]_s\), is non-empty, i.e., there exists a term of sort \( s \) with a variable \( x_{s'} \) of sort \( s' \). This condition is natural and so commonly satisfied in practice that one might say that signatures which do not satisfy it are bad.

\section{6.1.4 \ Free models}

The algebra \( T_E[X] \) is an \( E \)-\textit{free algebra generated by} \( X \). It is an \( E \)-algebra freely generated by \( X \) in the category \( \text{Alg}_E \) of \((\Sigma, E)\)-algebras and of \( \Sigma \)-morphisms, just as \( T_\Sigma[X] \) is an object freely generated by \( X \) in the category \( \text{Alg}_\Sigma \). It is the solution, unique up to isomorphisms, of the following universal problem (\( \varpi \) is the projection of \( T_\Sigma[X] \) in \( T_E[X] \)).

\textbf{Lemma 6.5} For every \( A \in \text{Mod}(E) \) and every mapping \( \xi : X \to A \), there exists a unique morphism \( \xi_E : T_E[X] \to A \) such that \( \xi_E(\varpi(x)) = \xi(x) \) for every \( x \in X \).

\textbf{Proof} It is important that \( T_\Sigma[X] \) be free in \( \text{Alg}_\Sigma \). The morphism \( \xi_E(M) \) must be defined if \( M \in T_E[X] \). But, if \( \varpi(M) = \varpi(N) \), i.e., \((M, N) \in E \), then \( \xi(M) = \xi(N) \), since \( A \) is a model of \( E \). One can then define \( \xi_E(M) = \xi(s(M)) \), where \( M \) is the equivalence class of \( M \), and \( s \) is a right-inverse of \( \varpi \). By the unicity of \( \xi_E \), \( \xi_E \circ \varpi = \xi \). The unicity of \( \xi_E \) follows from this relation and from the surjectivity of \( \varpi \). See Figure 6.2. \( \square \)

In particular, there is, for \( X = \emptyset \), an \textit{initial model}. By writing \( T_E \) instead of \( T_E[\emptyset] \), the initiality property in \( \text{Mod}(E) \) can be stated as follows: for every model \( A \) of \( E \), there exists a unique morphism \( \iota_A : T_E \to A \). (The initiality property in
\( \text{Alg}_\Sigma \) was: for every \( \Sigma \)-algebra, there exists a unique morphism \( \iota_A : T_\Sigma \rightarrow A \). The initial model will play an important role in the case of data structures.

The existence of free models is a very strong property of varieties. By reconsidering the examples of algebraic structures, the above discussion ensures the existence, for every set \( X \), of the free monoid \( X^* \), of the free commutative monoid \( X^{(N)} \), of the free group \( L(X) \), of the free abelian group \( Z^{(X)} \), of the free ring, etc., generated by \( X \). In the case of data structures, the free models allow the construction of parametrized structures.

**Definitions by generators and relations**

The existence of free models allows the definition of a \( \Sigma \)-algebra by generators and relations in \( \text{Alg}_\Sigma \), and not only in \( \text{Alg}_E \). The axioms of the variety state the general laws such as associativity, while the relations state particular properties. Finite presentations are particularly interesting. The varieties most studied using presentations are monoids and groups.

Here is the case of groups, which has received much attention due to its practical applications to other branches of mathematics, such as algebraic topology. Let \( G \) be a finite set of 'generators': the group freely generated by \( G \) is \( L(G) = T_E[G] \), where \( E \) is the congruence stable under substitutions generated by the three usual axioms of left-neutral element, left-inverse and associativity. One can represent \( L(G) \) by the set of words over the alphabet of \( g, \tilde{g} \), with \( g \in G \), and \( \tilde{g} \) designating the inverse of \( g \), which do not contain any subwords of the form \( gg \) or \( \tilde{g}\tilde{g} \), since these subwords would simplify. Introduce \( R \), a finite set of elements of \( T_\Sigma[G] \), the 'relators' (in the case of groups, a relation under the form of a pair of terms \( M = N \), is equivalent to \( MN^{-1} = e \), i.e. to the relator \( MN^{-1} \)). Let \( N_R \) be the normal subgroup of \( L(G) \) generated by the image \( \varpi(R) \) of the relators by the natural surjection. The group defined by the presentation \((G; R)\) and by generators and relators, is then the quotient group \( L(G; R) = L(G)/N_R \).

**Example** The free group with one generator \( g \) is isomorphic to \( Z \), by sending \( g \) to \( 1 \). By introducing the relator \( g^6 \), i.e., the relation \( g^6 = e \), one obtains the presentation group \( (g; g^6 = e) \), which is the additive group \( Z/6Z \). This group allows other presentations. Here is one with two generators and three relations.
(h, k; h^3 = k^2 = h^{-1}k^{-1}hk = e). One can go from one to the other by defining
h = g^4, k = g^3 and g = hk.

In the general case, let \( E \) be an equational theory, \( G \) a set of generators and
\( R \subseteq T_\Sigma[G] \times T_\Sigma[G] \) a set of relations. The \( E \)-algebra defined by \((G; R)\) is the quotient \( T_\Sigma[G; R] \) of \( T_\Sigma[G] \) by the congruence generated by the image of \( R \).

In the case of monoids, a Thue congruence is a congruence generated by a Thue system, i.e., by a set \( R \) of pairs of words over an alphabet \( A \). The monoid defined by \((A; R)\) is the quotient of the free monoid \( A^* \) by the congruence generated by \( R \).

**Example** Correct parenthesization is formalized using the alphabet
\[
P_n = \{p_1, \overline{p}_1, \ldots, p_n, \overline{p}_n\}
\]
formed of \( n \) pairs of parentheses (for \( n = 3 \), one could use \( P_3 = \{(, ), \{, \}, [ , ]\} \)), and the relations \( R = \{ p_ip_i = e; i = 1, \ldots, n \} \), which state simplification rules. \([(())[[[]]]][][[()]]\}[[[]]]\} \) is a correct parenthesization, while \([[]\}) \) is not. The class of the empty word is the set of correct parenthesizations. This set is called the Dyck language with \( n \) pairs of parentheses.

At first, the word problem was defined in the case of algebras defined by generators and relations in a variety, first in the case of groups by Dehn [37]. Let \( E \) be an equational theory, and \((G; R)\) be a finite presentation. The word problem for \((G; R)\) is, given a pair \((M, N) \in T_\Sigma[G] \times T_\Sigma[G]\), determine if \( T_\Sigma[G; R] \models M = N \).

For example, for the presentation of group \((h, k; h^3 = k^2 = h^{-1}k^{-1}hk = e)\) (of \( \mathbb{Z}/6\mathbb{Z} \), but this is not needed), the problem \( h^5k^5 = k^3h^{-1} \) has a positive solution: since the generators \( h \) and \( k \) commute, this equation is equivalent to \( h^6k^6 = e \), which is satisfied, since \( h^6 = (h^2)^3 = e \) and \( k^6 = (k^2)^3 = e \). This equational reasoning is not always simple: in 1955 Novikov and Boone [114] found a finite presentation of a group for which the word problem is undecidable. The word problem for a free group (this is the case where \( R = \emptyset \)) is decidable, as will be seen, but not in the whole variety of groups.

**Genericity of free models**
From a semantic point of view, the free objects of a variety play a generic role, in the sense that the equations between generators are satisfied by all the algebras of the variety.

**Proposition 6.6** \( \text{Eq}_X(T_\Sigma[X]) = E \) if \( E \) is an equality over \( T_\Sigma[X] \).

**Proof** Let \( X \) be a set of variables and \( M = N \) an equation with variables in \( X \). Suppose that \( T_\Sigma[X] \models M = N \). It must be shown that, for every model \( A \) of \( E \), \( A \models M = N \) must also hold. To prove this, \( \xi(M) = \xi(N) \) must be proven for every \( \xi \in \Sigma^X \). Since \( T_\Sigma[X] \) is the \((\Sigma, E)\)-algebra freely generated by \( X \), there is an extension \( \xi : T_\Sigma[X] \to A \) of \( \xi \) satisfying \( \xi \circ \omega = \hat{\xi} \). But, \( T_\Sigma[X] \models M = N \) implies
\(\varpi(M) = \varpi(N)\) since \(\varpi\) is the extension of the inclusion of \(X\) in \(T\). It follows that \(\xi(M) = \xi(\varpi(M)) = \xi(\varpi(N)) = \xi(N)\), which is the desired result. \(\square\)

Hence, to establish validity in \(\text{Mod}(\mathcal{E})\) of an arbitrary equation (which here means with an arbitrary number of variables), it suffices to work in a unique algebra, which is \(T\), where \(X\) is an infinite set fixed once and for all. Therefore, the study of a variety of algebras can be done through a single algebra, of a syntactic nature, which will allow equational reasonings.

In particular, this result is applicable if \(X = \emptyset\): if \(M = N\) is an equation between closed terms satisfied by the initial model \(T\), then it is also satisfied by all of the models of \(\mathcal{E}\). However, the assumption ‘equations between generators’ is crucial. For example, if \(T \models M = N\), where \(M = N\) is an equation between terms containing two variables \(x\) and \(y\), one cannot deduce that \(M = N\) is satisfied by all of the models of \(\mathcal{E}\). To obtain this conclusion, \(T \models x = y\) would have had to be proven, not just \(T \models M = N\).

Consider the case of groups. The initial model of the variety of groups, which is the free group with 0 generators \(L(\emptyset)\), is the trivial group \(\{e\}\): it satisfies any equation! The situation is hardly improved by adding a generator: the free group with one generator \(L(\{g\})\) is isomorphic to \(\mathbb{Z}\). It satisfies at least the commutativity relation (and many others), which is not satisfied by all groups. It turns out that the equational theory of the free group with two generators coincides with the equational theory of groups, but that is a specific property of the variety of groups.

The situation is generally less caricatural in the case of the initial model of a data structure whose equational theory is of real interest.

Addition is specified by the following equations:

\[
(+) \quad \begin{cases} 
  x + 0 &= 0 \\
  x + s(y) &= s(x + y).
\end{cases}
\]

Their ‘standard’ model is \(\mathbb{N}\), which is the initial model. A non-standard model will be constructed, the algebra \(\mathbb{N}'\) of colored integers, to show that there are equations satisfied by \(\mathbb{N}\) which are not consequences of the addition axioms. The domain of \(\mathbb{N}'\) is formed of two copies of \(\mathbb{N}\), the black integers \((\blacklozenge, n)\) and the red integers \((\blacklozenge', n)\). The constant 0 is interpreted by the black integer \((\blacklozenge, 0)\). The operations are normally interpreted over the integers, with the added stipulation that the successor conserves the color of its argument and addition conserves the color of its first argument:

\[
s(\gamma, n) = (\gamma, n + 1) \quad (\gamma, n) + (\gamma', n') = (\gamma, n + n'),
\]

where \(\gamma\) is either the color \(\blacklozenge\) or the color \(\blacklozenge'.\) This algebra is a model of additive arithmetic.
For \((\gamma, n), (\gamma', n') \in \mathbb{N}'\),
\[
\begin{align*}
(\gamma, n) + (\bigodot, 0) &= (\gamma, n) \\
(\gamma, n) + s(\gamma', n') &= (\gamma, n) + (\gamma', s(n')) \\
&= (\gamma, n + s(n')) \\
&= (\gamma, s(n + n')) \\
&= s(\gamma, n + n').
\end{align*}
\]

However, the equations \(0 + x = x\) and \(x + y = y + x\) are not satisfied in \(\mathbb{N}'\):
\[
\begin{align*}
(\bigodot, 0) + (\bigtriangledown, 3) &= (\bigodot, 3) \neq (\bigtriangledown, 3) \\
(\bigodot, 1) + (\bigtriangledown, 2) &= (\bigodot, 3) \\
&\neq (\bigtriangledown, 3) \\
&= (\bigtriangledown, 2) + (\bigodot, 1).
\end{align*}
\]

As is standard, this example suffices to show that commutativity is not a consequence of the equations defining addition.

**Free extensions without axioms**

Given the signature of lists and an arbitrary set \(A\) of ‘atoms’, the algebra freely generated by \(A\) is the set of lists of elements of \(A\). If \(A\) is not only a set, but an algebra with its own operations, one should be able to pass these operations as parameters. This is what the functor construction of Standard ML implements.

Starting with a signature \(\Sigma^0\) and an extension \(\Sigma = \Sigma^0 \cup \Sigma^1\) of \(\Sigma^0\), a ‘functor’ is constructed: it associates with every \(\Sigma^0\)-algebra \(\mathcal{X}\) a \(\Sigma\)-algebra \(F(\mathcal{X})\). The idea is that the operations of \(\Sigma^0\) which are already interpreted in \(\mathcal{X}\) keep this interpretation, while the operations of \(\Sigma^1\) are ‘free’. So the evaluation takes place in \(T_\Sigma[X]\), and everything which can be evaluated in \(\mathcal{X}\) is associated with its value. The elements of \(T_\Sigma[X]\) which can be first evaluated are those of \(T_{\Sigma^0}[X]\), using the evaluation morphism \(ev_\mathcal{X} : T_{\Sigma^0}[X] \to \mathcal{X}\), whose kernel is the equational diagram of \(\mathcal{X}\). To evaluate a term of \(T_\Sigma[X]\), evaluate its subterms, which are in \(T_{\Sigma^0}[X]\), which is expressed by forming the \(\Sigma\)-congruence of \(T_\Sigma[X]\) generated by the diagram of \(\mathcal{X}\). Let \(T_{\Sigma^0}[\mathcal{X}]\) be the quotient of \(T_\Sigma[X]\) by this congruence. This \(\Sigma\)-algebra is the one sought, and can be written in ML:

\[
\text{functor } T (X : \text{SIGMA0}) = T_{\Sigma^0}[\mathcal{X}] : \text{SIGMA}
\]

By taking the most evaluated term in each class, the domain of \(T_{\Sigma^0}[\mathcal{X}]\) can be equated with the subset of \(T_\Sigma[X]\) composed of terms for which every subterm belonging to \(T_{\Sigma^0}[X]\) is in fact an element of \(X\), which gives an injection from \(T_{\Sigma^0}[\mathcal{X}]\) to \(T_\Sigma[X]\), which is a right-inverse of the natural surjection. This injection is not a \(\Sigma\)-morphism, since the operations of \(\Sigma^0\) are interpreted in \(T_{\Sigma^0}[\mathcal{X}]\), and not in \(T_\Sigma[X]\), but it is a \(\Sigma^1\)-morphism. Nevertheless, this allows the elements of \(T_{\Sigma^0}[\mathcal{X}]\) to be interpreted as terms—assuming that those of \(\mathcal{X}\) can be implemented.
The property which justifies this construction is similar to that of the free algebra. It has simply been relativized to $\Sigma^0$: for every $\Sigma^0$-morphism $\xi : \mathcal{X} \to A$, there exists a unique $\Sigma$-morphism $\hat{\xi}_{\Sigma^0} : T_{\Sigma^0}[\mathcal{X}] \to A$ such that for every $x \in X$, $\hat{\xi}_{\Sigma^0}(x) = \xi(x)$.

As for the free algebra, it is not the algebra $T_{\Sigma^0}[\mathcal{X}]$ which is important, but the functor $T_{\Sigma^0} : \mathcal{X} \mapsto T_{\Sigma^0}[\mathcal{X}]$, for which there is an implementation in Standard ML's system of modules.

Free extensions and axioms

The functor $T_{\Sigma^0}$ gives a satisfying answer to the parametrization problem in the absence of axioms. If what is desired is not a list of $foo$ but, rather, a set of $foo$, there are axioms which state that insertion into a set is ‘idempotent’, ‘associative’ and ‘commutative’, and hence axioms over the produced algebra. If one also wishes to transform a binary relation over the elements into an operation over sets of elements, similarly to the iterators of functional programming, this operation itself must be commutative and associative, otherwise, the transformation would have no meaning. Therefore axioms must be imposed on a parametrized algebra.

Frequently, the base type is the initial algebra of a class $\text{Alg}_\Sigma$, for which the signature is the set of constructors or generators: this is the case for the natural integers (constructors 0 and s) and the Booleans (constructors true and false). Rational integers are generated by 0, s and p, with $psx = x$ and $spx = x$.

Starting from a signature $\Sigma^0$, a set of axioms $E^0$ over $\Sigma^0$, an extension $\Sigma = \Sigma^0 \cup \Sigma^1$ and $E = E^0 \cup E^1$, where $E^1$ is a set of equations over $\Sigma$ (not only over $\Sigma^1$), one constructs a functor $T_{E^0} : \mathcal{X} \mapsto T_{E^0}[\mathcal{X}]$ from the category of $E^0$-algebras to that of $E$-algebras.

Let $\mathcal{X}$ be a $E^0$-algebra. The set $T_{E^0}[\mathcal{X}]$ is the quotient of $T_\mathcal{E}[X]$ by the congruence generated by the kernel of the $\Sigma^0$-morphism of $T_\mathcal{E}[X]$ in $\mathcal{X}$ extending the identity of $X$. This algebra verifies the universal property: for every $E$-algebra $A$ and every $\Sigma^0$-morphism $\xi : \mathcal{X} \to A$, there exists a unique $\Sigma$-morphism $\hat{\xi}_{E^0} : T_{E^0}[\mathcal{X}] \to A$ such that for every $x \in X$, $\hat{\xi}_{E^0}(x) = \xi(x)$.

Examples

Here are a few extensions starting from $\Sigma^0 = \{0, s, p\}$, $E^0 = \{psx = x, spx = x\}$ and $\mathcal{X} = \mathbb{Z}$.

- $\Sigma^1 = \{+\}$ and $E^1 = \{x + 0 = x, x + s(y) = s(x + y), x + p(y) = p(x + y)\}$: $T_{E^0}[\mathcal{X}] \approx \mathbb{Z}$: this is a good situation, and one can say that $E$ is a good specification of addition over the integers;
- $\Sigma^1 = \emptyset$ and $E^1 = \{sxs = x\}$: $T_{E^0}[\mathcal{X}] \approx \{0, 1\}$: the axiom implies the identification of elements of the domain to integers modulo 2; the extension is not ‘hierarchically consistent’;
- $\Sigma^1 = \{+\}$ and $E^1 = \{x + 0 = x, x + s(y) = s(x + y)\}$: $T_{E^0}[\mathcal{X}] \not\approx \mathbb{Z}$: there are new elements such as $0 + p(0)$ which cannot be evaluated to an integer; the specification of $+$ is not ‘sufficiently complete’.
The deduction problem
In general, the problem is to know, given a class of algebras $C$ and an equation $M = N$, whether $C \vdash M = N$. Since the simplest way of producing a class of algebras is to fix a set $E$ of axioms, and to take for $C$ the variety defined by $E$, the particular case $\text{Mod}(E) \vdash M = N$ should suffice. But this problem can be brought back to the study of a single algebra $T_E[X]$, $X$ infinite, thanks to the genericity property of free models. This problem will be handled using a rewriting system obtained by orienting the equations of $E$, when this is possible.

However, the apparently simpler situation, where an algebra $A$ is taken instead of a class, is in general much more complex, except if $A = T_E[X]$, where $X$ is infinite. However, this situation presents itself naturally with $A = T_E$, which satisfies more equations than the free model, as the colored integer example showed. There is no finite and complete inference system in that case, and one must then use induction proofs.

6.2 Unification

Unification plays a central role, to prove as well as to compute. It also allows the resolution and propagation of constraints, hence the transfer of information.

The history of unification goes back to 1930 with Herbrand's $A$ property [62]. Used by Davis in 1963 [33] in the implementation of a proof procedure extending that of Prawitz (1960, [124]), unification was rediscovered in 1965 by Robinson [128] when he introduced the resolution principle. It became the subject of major research with the contributions of Plotkin (1972, [121]) and Huet (1976, [67]).

The practical importance of unification is due to the fact that it is an elementary operation: the power of 5-th generation machines is measured in kLIPS, for thousands of logical inferences per second, each logical inference corresponding to exactly one unification.

6.2.1 Equation resolution

Let $\Sigma$ be a signature, $M_1, M_2 \in T[X]$, and $A$ be a $\Sigma$-algebra. A solution of the equation $M_1 = M_2$ in $A$ is a valuation $\xi: \text{var}(M_1) \cup \text{var}(M_2) \to A$ such that $\xi(M_1) = \xi(M_2)$. To solve an equation is to search for its solution, which means to find in $A$ values for the variables of $M_1$ and $M_2$ which satisfy the equation.

**Example** The equation $x^3 - 2 = 0$ does not have a solution in $\mathbb{Q}$, has a solution in $\mathbb{R}$ ($\xi(x) = \sqrt[3]{2}$) and has three solutions in $\mathbb{C}$ ($\xi_k(x) = i^k \sqrt[3]{2}$, $k = 0, 1, 2$).

The very definition of the problem uses the algebra $A$ upon which depend the existence and the number of eventual solutions. Many constructions, such as fields of fractions, algebraic extensions, localization, completion of a metric space and
distributions, only exist to guarantee the existence, and in certain conditions the
unicity, of solutions of a certain class of equations.

Solving equations remains an abstract problem, except when equality over $A$ is
decidable, and in particular when the conditions required to answer this question
all hold (see Chapter 7). To unify is to solve equations over a syntactic domain.
Depending on the domain, solving equations is called:

- finite unification, in the algebra of terms $T[X]$;
- rational unification, in the algebra $T^r[X]$ of rational terms;
- unification modulo $\mathcal{E}$, in the free model $T_\mathcal{E}[X]$ of a theory $\mathcal{E}$;
- higher-order unification, in the set of typed $\lambda$-terms.

In the equational case, since the necessary condition of decidability is not al-
ways verified, only some particularly interesting cases are studied: the associa-

tive–commutative (AC), associative (which corresponds to solving equations in the free
monoid) and Boolean ring cases.

Most of this section will deal with finite unification, which is a fundamental
operation for all 'symbolic' (functional, logic and equational) programming. A
solution of $M_1 = M_2$ is in this case a substitution $\sigma$, of domain included in $\text{var}(M_1) \cup \text{var}(M_2)$, called a unifier, such that $\sigma M_1 = \sigma M_2$. It then has universal validity:

if $A$ is a $\Sigma$-algebra, the $\xi \circ \sigma$ are solutions in $A$ for every valuation $\xi$ with values

in $A$. In particular, for $A = T[X]$, all the instances $\mu \circ \sigma$ of $\sigma$ by an arbitrary

substitution $\mu$ are also solutions in $T[X]$, i.e., unifiers. But this infinity of solutions

can be described as the set of instances of one among them, the least one.

Although it is easy to prove, this is a fundamental result, related to the compu-
tation of the least upper bound in the term semi-lattice, which is constantly used
in logic programming and rewriting systems. Proven for the finite and rational

cases, this property is rare for unification modulo $\mathcal{E}$ (it has been known only since
1986 for the theory of Boolean rings [102]).

This property does not even have any meaning in non-syntactic domains: for
example, the equation $e^x = x + 2$ has two solutions in $\mathbb{R}$, neither of which can
reasonably be considered to be an instance of the other. In the case of vector
spaces, a linear equation $Lu = 0$ has in general a set of infinite solutions. But (in
finite dimension, for example), if a finite basis $B = \{e_1, \ldots, e_n\}$ is known, all of the
solutions are obtained by linear combinations of the $e_i$. This situation appears for
some equational theories, such as AC. In other cases, only infinite bases exist, or no
basis whatsoever. In the general context of unification modulo $\mathcal{E}$, finite unification
is only a particular case, that of the empty theory.

### 6.2.2 Systems of equations and substitutions

We begin by examining the relations between substitutions and systems of equa-
tions for which the substitutions are solutions. Write $\mathbb{S}$ for the set of substitutions
and $\bar{M} = \bar{N}$ for a system of equations $M_i = N_i$, $i = 1, \ldots, p$. 

Definition 6.4

- Let $E$ be a system of equations $\bar{M} = \bar{N}$ and $\sigma \in S$. The substitution $\sigma$ is a unifier of $E$ or $\sigma$ unifies $E$ if $\sigma$ is a solution of $E$ in $T[X]$, i.e., for every $i$, $\sigma M_i = \sigma N_i$. Write $U(E)$ for the set of unifiers of $E$.
- $E$ is unifiable if $U(E) \neq \emptyset$.
- Two systems of equations $E$ and $E'$ are equivalent if $U(E) = U(E')$.

The systems which are easiest to solve are obviously those which have already been solved! For example, if a solution to the equation $x = a$ is sought, the obvious solution is $x = a$, or, to conform to the above definition, the substitution $\sigma = [\frac{x}{a}]$. It is therefore natural to associate a substitution with a system of variables having only distinct variables on the left-hand sides.

Definition 6.5

- A system of equations $E$ is in reduced form if it is of the form $x = M$, the $x_i$ variables being pairwise distinct;
- if $E$ is in reduced form, its associated substitution, written $\sigma_E$, is $[\frac{x_1 \ldots x_n}{M_1 \ldots M_n}]$,

$$\begin{cases} \sigma_E x_i = M_i, & 1 \leq i \leq n \\ \sigma_E y = y & \text{if } y \notin \{x_1, \ldots, x_n\} \end{cases}$$

The error would be to believe that, as soon as $E$ is in reduced form, $\sigma_E$ is a solution of $E$. Consider three examples:

$$(E_1) \begin{cases} x = f(z) \\ y = g(b, z) \end{cases} \quad (E_2) \begin{cases} x = y \\ y = a \end{cases} \quad (E_3) \begin{cases} x = f(y) \\ y = g(x) \end{cases}$$

For $E_1$ everything goes well and $\sigma_{E_1} = [\frac{x}{f(z)} \frac{y}{g(b, z)}]$ is a solution of $E_1$.

For $E_2$, things are not as simple: $\sigma_{E_2}$ is not a solution of the first equation. The value of $x$ is not obtainable immediately, since $y$ appears on the left-hand side of the second equation. The substitution $\sigma_{E_2}$ must be iterated a second time to obtain a unifier $\theta = \sigma_{E_2}^2 = [\frac{x}{f(z)} \frac{y}{g(b, z)}]$.

Finally, for $E_3$, $\sigma_{E_3}$ does not satisfy either of the two equations. Furthermore, it is obvious that the iterating method used for $E_2$ leads to an infinite loop. This occurs because of the cyclic dependencies between $x$ and $y$, while in $E_2$ there were no cycles. In fact, $E_3$ is not unifiable in $T[X]$, but, rather, in $T'[X]$ by $\theta = [\frac{x}{f(y)} \frac{y}{g(f(y))}]$, which is the 'limit' of the sequence of iterations of $\sigma_{E_3}$: $\sigma_{E_3}, \sigma_{E_3}^2, \sigma_{E_3}^3, \ldots$.

So a stronger condition must be imposed on systems of equations so that they need not be iterated to obtain their solutions.

Definition 6.6 A system of equations $E$ is in (finite) solved form if it is in reduced form, with $\{x_1, \ldots, x_n\} \cap \bigcup_{i=1}^n \text{var}(M_i) = \emptyset$.

It will be shown that every unifiable system is equivalent to a system in solved form. If $E$ is in solved form, it will be shown that $\sigma_E \in U(E)$ and that it is the most general solution of $E$, under a preorder which will be introduced.
The preorder of substitutions

Definition 6.7 Let \( \sigma \) and \( \theta \) be two substitutions. The substitution \( \sigma \) is called an instance of \( \theta \), or \( \theta \) is more general than \( \sigma \), written \( \sigma \geq \theta \), if there exists a substitution \( \mu \) such that \( \sigma = \mu \theta \).

This relation \( \geq \) is clearly a preorder over \( S \). Another way of defining a preorder over \( S \) consists of extending to substitutions that of \( T[X] \), by extensionality. Write \( \sigma \geq_e \theta \) if \( \sigma M \geq \theta M \) for every \( M \in T[X] \).

Proposition 6.7

- If \( \sigma \geq \theta \), then \( \sigma \geq_e \theta \).
- If \( \sigma \geq_e \theta \), and if \( \Sigma \) is non-linear (i.e., there exists at least one \( f \in \Sigma \) of arity \( > 1 \)), then \( \sigma \geq \theta \).

Proof The first point is trivial. To prove the converse, note that if \( \Sigma \) is non-linear, there exists for every \( W \subseteq X, W \) finite, a term \( M \in T_\Sigma [X] \) such that \( \text{var}(M) \supseteq W \). Suppose that \( \sigma \geq_e \theta \), and take \( M \) such that \( \text{var}(M) \supseteq \text{dom}(\sigma) \cup \text{dom}(\theta) \cup \text{im}(\theta) \).

Since \( \sigma(M) \geq \theta(M) \), there exists \( \mu \) such that \( \sigma M = \mu(\theta M) = (\mu \theta)M \). It will be shown that \( \sigma = \mu \circ \theta \). It has already been shown that \( \sigma|\text{var}(M) = (\mu \theta)|\text{var}(M) \). Furthermore, suppose that \( \text{dom}(\mu) \subseteq \text{var}(\theta M) \). But this last set of variables is itself bounded by: \( \text{var}(\theta M) \subseteq \text{var}(M) \cup \text{im}(\theta) \). Hence, if \( x \notin \text{var}(M) \), \( x \) does not belong to any of \( \text{dom}(\theta) \), \( \text{dom}(\sigma) \) or \( \text{dom}(\mu) \), so \( \sigma x = \mu \theta x = x \). Hence, \( \sigma = \mu \theta \). \( \square \)

The preorder can be used to formalize the following intuitive fact: one can always instantiate a solution of a system of equations.

Proposition 6.8 Every instance of a unifier is a unifier.

This means that \( U(E) \) is a filter of \( (S, \geq) \). The most general unifier property will show that the filter is general, i.e., generated by a single element \( \sigma \), in the sense that \( \emptyset \in U(E) \) if and only if \( \emptyset = \mu \sigma \) for some \( \mu \).

To analyze a substitution \( \sigma \), associate with it the following three sets of variables:

- \( \text{dom}(\sigma) = \{ x \in X; \sigma x \neq x \} \) is the domain of \( \sigma \);
- \( \text{range}(\sigma, V) = \bigcup_{x \in V} \text{var}(\sigma x) \) is the range of \( \sigma \) over \( V \);
- \( \text{im}(\sigma) = \text{range}(\sigma, \text{dom}(\sigma)) \) is the image of \( \sigma \).

Recall that, by definition, \( \text{dom}(\sigma) \) is always a finite set. Note that \( \text{range}(\sigma, V) \) is infinite if and only if \( V \) is infinite and that \( \text{im}(\sigma) \) is therefore finite. A substitution is essentially represented by an application of \( \text{dom}(\sigma) \) in \( T[\text{im}(\sigma)] \), but also by taking an arbitrary \( V \) larger than \( \text{dom}(\sigma) \), by a mapping from \( V \) to \( T[\text{range}(\sigma, V)] \). The following property holds: \( \text{dom}(\sigma) \cup \text{range}(\sigma, V) = V \).
Idempotent substitutions A substitution $\theta$ is idempotent if $\theta^2 = \theta$. The three following statements are equivalent:

1. $\theta$ is idempotent;
2. $\operatorname{dom}(\theta) \cap \operatorname{im}(\theta) = \emptyset$;
3. $\operatorname{dom}(\theta) \cap \operatorname{range}(\theta, X) = \emptyset$.

Another way to state (2) is to say that $\left[ \frac{x_1 \ldots x_n}{M_1 \ldots M_n} \right]$ is idempotent if $\{x_1, \ldots, x_n\} \cap \bigcup_{i=1}^n \operatorname{var}(M_i) = \emptyset$. It follows that, for a system of equations $E$ in reduced form, $\theta_E$ is idempotent if and only if $E$ is in solved form.

In the preorder over $\mathbb{S}$, the idempotent substitutions play a key role.

**Lemma 6.9** $\theta$ is idempotent if and only if for every $\sigma \in \mathbb{S}$, $\sigma \geq \theta$ is equivalent to $\sigma = \sigma \theta$.

**Proof** Suppose $\theta^2 = \theta$. If $\sigma \geq \theta$, there exists $\mu$ such that $\sigma = \mu \circ \theta$, so $\sigma \circ \theta = (\mu \circ \theta) \circ \theta = \mu \circ \theta^2 = \mu \circ \theta = \sigma$. Conversely, if $\sigma = \sigma \theta$, then, by the definition of $\geq$, $\sigma \geq \theta$.

If equivalence holds, then $\theta \geq \theta$ implies $\theta = \theta \theta = \theta^2$. \qed

The link between the preorder over $\mathbb{S}$ and the resolution of equations in reduced form is given by the following lemma, where (2) follows directly from the preceding lemma:

**Lemma 6.10** Let $x$ be a variable, $M$ a term and $\sigma$ a substitution.

1. $\sigma \in U(x = M)$ if and only if $\sigma \left[ \frac{x}{M} \right] = \sigma$;
2. $\left[ \frac{x}{M} \right]$ is idempotent if and only if $U(x = M) = \{\sigma; \sigma \geq \left[ \frac{x}{M} \right]\}$.

The first properties of substitutions lead directly to the desired result.

**Proposition 6.11** If the system $(E) : \bar{x} = \bar{M}$ is in solved form, then $\theta_E$ is the least element of $U(E)$.

**Proof** It has already been seen that $\theta_E \in U(E)$. Let $\sigma \in U(E)$. Then $\sigma x_i = \sigma M_i$ for every $i = 1, \ldots, n$, so $\sigma \circ \left[ \frac{x_i}{M_i} \right] = \sigma$, $i = 1, \ldots, n$ according to the lemma, from which $\sigma \circ \left[ \frac{x_1}{M_1} \right] \circ \ldots \circ \left[ \frac{x_n}{M_n} \right] = \sigma$. But, since the $x_i$ are pairwise distinct and do not appear in the $M_j$, it is clear by induction that

$$\left[ \frac{x_1}{M_1} \right] \circ \ldots \circ \left[ \frac{x_n}{M_n} \right] = \left[ \frac{x_1 \ldots x_n}{M_1 \ldots M_n} \right] = \theta_E$$

and so $\sigma \circ \theta_E = \sigma$, i.e., $\sigma \geq \theta_E$.

$\theta_E$ is called a most general unifier (mgu) of $E$. Of course, since $\geq$ is only a preorder of $\mathbb{S}$, and not an order, this unifier is only defined up to equivalence. However, when it does not create problems, we will refer to the mgu.
6.2.3 Transformations of systems of equations

The search for solutions in a system of equations means searching for an equivalent solved form. The Herbrand unification algorithm to be presented consists of a sequence of steps which either transform the system into an equivalent system closer to the solved form, or detect that the system does not have a solution.

Finite unification
Transformations $E \rightsquigarrow E'$ are defined between systems of equations. Here, a system of equations will refer either to a set of equations, as before, or a symbol $\bot$ representing a system without solution. Write $E \rightsquigarrow E'$ if $E \rightsquigarrow_1 E'$ for one of the transformations $t$ (see Figure 6.3). The transition $\rightsquigarrow$ is the reflexive transitive closure of $\rightsquigarrow_1$. Solving $E_0$ consists of applying a sequence of transformations $E_0 \rightsquigarrow E_n$ so that $E_n$ is in solved form, or else $E_n = \bot$.

<table>
<thead>
<tr>
<th>decomposition: $E \cup {f \bar{M}_1 \ldots \bar{M}_r = f \bar{N}_1 \ldots \bar{N}<em>r} \rightsquigarrow</em>{\text{dec}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \cup {\bar{M}_1 = N_1, \ldots, \bar{M}_r = N_r}$</td>
</tr>
<tr>
<td>erasure: $E \cup {M = M} \rightsquigarrow_{\text{erase}}$</td>
</tr>
<tr>
<td>$E$</td>
</tr>
<tr>
<td>elimination: $E \cup {x = M} \rightsquigarrow_{\text{elim}}$</td>
</tr>
<tr>
<td>$E[x := M] \cup {x = M}$, if $x \notin \text{var}(E), x \notin \text{var}(M)$</td>
</tr>
<tr>
<td>inversion: $E \cup {M = x} \rightsquigarrow_{\text{inv}}$</td>
</tr>
<tr>
<td>$E \cup {x = M}$, if $M \notin X$</td>
</tr>
<tr>
<td>conflict: $E \cup {f \bar{M} = g \bar{N}} \rightsquigarrow_{\text{conflict}}$</td>
</tr>
<tr>
<td>$\bot$, if $f \neq g$</td>
</tr>
<tr>
<td>cycle: $E \cup {x = M} \rightsquigarrow_{\text{cycle}}$</td>
</tr>
<tr>
<td>$\bot$, if $x \in \text{var}(M)$</td>
</tr>
</tbody>
</table>

Figure 6.3 Finite unification

The last two transformations lead immediately to the system $\bot$ to which no more transformations are applicable. They are failures. Elimination is the least elementary transformation, since substitution is not an elementary operation. The broadcasting of a binding

$$E \cup E' \cup \{x = M\} \rightsquigarrow_{\text{diff}} E[x := M] \cup E' \cup \{x = M\}$$

is a general transformation, which is more primitive when $E$ is formed of a single equation. When $E' = \emptyset$, the transformation is elimination.
Example

\[ \{ f(x, g(x), k(y)) = f(h(z), t, k(z)), x = h(y) \} \]

\( \sim_{\text{dec}} \{ x = h(z), g(x) = t, k(y) = k(z), x = h(y) \} \)

\( \sim_{\text{elim}} \{ x = h(z), g(h(z)) = t, k(y) = k(z), h(z) = h(y) \} \)

\( \sim_{\text{inv}} \{ x = h(z), t = g(h(z)), k(y) = k(z), h(z) = h(y) \} \)

\( \sim_{\text{dec}} \{ x = h(z), t = g(h(z)), y = z, h(z) = h(y) \} \)

\( \sim_{\text{dec}} \{ x = h(z), t = g(h(z)), y = z, z = y \} \)

\( \sim_{\text{elim}} \{ x = h(z), t = g(h(z)), y = z, z = z \} \)

\( \sim_{\text{erase}} \{ x = h(z), t = g(h(z)), y = z \} \)

The last system is in solved form, defining the idempotent substitution:

\[ \theta = \begin{bmatrix} x & y & t \\ h(z) & z & g(h(z)) \end{bmatrix}. \]

\[ \square \]

**Lemma 6.12**

- If \( E \sim_{\text{dec}} E' \), then \( E \) and \( E' \) have the same solutions in \( T[X] \) (and in \( T^r[X] \)).
- If \( E \sim_{\text{erase}} E' \), \( E \sim_{\text{elim}} E' \) or \( E \sim_{\text{inv}} E' \), then \( E \) and \( E' \) have the same solutions in every algebra.
- If \( f \neq g \), equation \( fM = g\bar{N} \) has no solution in \( T[X] \) (nor in \( T^r[X] \)).
- If \( x \in \text{var}(M) \), equation \( x = M \) has no solution in \( T[X] \).

**Proof** Consider the case of \( \sim_{\text{elim}} \). Let \( A \) be an arbitrary algebra: it must be shown that \( \{ P = Q, x = M \} \) and \( \{ P[x := M] = Q[x := M], x = M \} \) have the same solutions \( \xi : X \to A \). A solution of one system or the other satisfies \( \xi(x) = \bar{\xi}(M) \). It follows, by induction on \( P \), that \( \bar{\xi}(P) = \bar{\xi}(P[x := M]) \), which establishes the equivalence. \( \square \)

Note the role of the algebra in each case. Decomposition and failure by conflict preserve solutions since the work is being done in \( T[X] \) or \( T^r[X] \). In the general case, every solution of \( E' \) is a solution of \( E \), but not the other way around (in \( \text{N} \), \( x + 2 = 1 + y \) does not imply \( x = 1 \) and \( y = 21 \)). It is at this level that unification in algebras is simpler than unification in an equational theory. The equivalence between equation \( M = N \) and system \( M_1 = N_1, \ldots, M_r = N_r \) also shows that if one starts with a single equation to be solved, one must then solve a system of equations.

The elimination of a variable is nothing but the ubiquitous 'by replacing \( x \) in equation (3) by the its value found in (6), the result is ... ', valid reasoning in any algebra.

The only transformation which differentiates \( T[X] \) and \( T^r[X] \) is the failure by cycle, when an \textit{occur check} \( x \in \text{var}(M) \) is positive.
Proposition 6.13

- Relation $\leadsto$ is well-founded.
- If $E \leadsto \bot$, then $E$ is not unifiable.
- If $E \leadsto E' \neq \bot$, and if no transformation is applicable to $E'$, then $E'$ is a system equivalent to $E$, in solved form, and such that $\text{var}(E') = \text{var}(E)$, and $\theta_{E'}$ is a most general idempotent unifier of $E$.

Proof According to lemma 6.12, all the transformations considered preserve the set of solutions. Note also that the set of variables of the system is also preserved by the first four transformations.

The termination is proven by supposing that the algorithm does not terminate in failure. A 'solved' variable is a variable $x$ which has only one occurrence in $E$, and that in the left-hand side. Use $\nu$ for the number of unsolved variables at a given step of the algorithm. It is clear that, at each elimination, the substituted variable $x$ goes from the status of unsolved variable to that of solved variable, since the occur check has been done. Hence each $\leadsto_{\text{elim}}$ decreases $\nu$. It is also clear that the status of solved variable is acquired permanently, and that the other transformations cannot increase $\nu$. Furthermore, each decomposition or erasure decreases the total size of $E$, while each inversion leaves it untouched. Write $\tau$ for its size. Finally, write $\delta$ for the number of 'inverted' equations of the form $M = x$ with $M$ non-variable. Each inversion decreases $\delta$. Hence, each transformation reduces the triplet $(\nu, \tau, \delta)$ using the lexical order over $\mathbb{N}^3$. Since this order is well-founded, the relation $\leadsto$ is also well-founded, and so only a finite number of transformations can be done.

If the algorithm terminates in failure, then $E$ is not unifiable, according to the lemma.

If no transformation is applicable to the result $E'$, then $E'$ is in solved form. □

This proposition allows the design of a 'non-deterministic' unification algorithm (Herbrand's algorithm, rediscovered by Martelli and Montanari). As in the case of the $\lambda$-calculus or of rewriting systems, there are rules, applicable in an undetermined order; here, systems of equations play the role of terms. This process can therefore be seen as a 'computation by reduction', which satisfies the strong normalization property (there is no confluence, but that is not important), which allows a don't care non-determinism. There are, however, two other hidden properties, of soundness ($\theta_{E'}$ is a unifier) and of completeness (every unifier is an instance of $\theta_{E'}$), which are much more difficult to show for other unification problems. Furthermore, there are failure cases, which yield an immediate response here (non-unifiable system). In other situations, backtracking would have to be done to try other transformations: this is don't know non-determinism. It is in fact the first example of 'computation by resolution', another computation paradigm which will be studied in the next chapter. Proposition 6.13 can be stated under the form:

Theorem 6.14 If a system of equations $E$ is unifiable, there exists $\theta \in U(E)$ such that
\( \sigma \in \text{U}(E) \) if and only if \( \sigma \geq \theta \);
- \( \text{dom}(\theta) \cup \text{im}(\theta) = \text{var}(E) \);
- \( \theta \) is idempotent.

Rational unification

The search for solutions in \( T^r[X] \) is distinguished by the absence of the occur check. The concept of solved form is therefore different.

A system of equations \( E \) is in rational solved form if it is in reduced form and it does not include any cycle \( x_{i_1} = x_{i_2}, \ldots, x_{i_n} = x_{i_1} \).

If the occur check is omitted from \( \rightsquigarrow_{\text{elim}} \) and the failure transformation \( \rightsquigarrow_{\text{cycle}} \) is omitted, then relation \( \rightsquigarrow \) is no longer well-founded:

\[
\{ x = f(x), y = x \} \rightsquigarrow_{\text{elim}} \{ x = f(x), y = f(x) \} \rightsquigarrow_{\text{elim}} \{ x = f(x), y = f(f(x)) \} \rightsquigarrow \ldots.
\]

Uncontrolled, elimination is dangerous, though useful. For example, if a system \( \{ x = f(a, y), x = f(z, b) \} \) is obtained, \( x \) must be eliminated to allow the continuation with \( \{ x = f(a, y), f(a, y) = f(z, b) \} \) by decomposition, \( \ldots \). But one can always start by applying decompositions, inversions and erasures. Colmerauer then proposed another transformation, confrontation, or merge (see Figure 6.4), and he limited elimination of \( x \) to the case of an equation \( x = y \), where \( x \) and \( y \) are distinct variables, a condition which is a degenerate form of the occur check! The choice of equation conserved by a merge is made using an order over the equations.

<table>
<thead>
<tr>
<th>merge:</th>
<th>( E \cup { x = M, x = N } ) if ( M \notin X ) and ( x = N \rightarrow M = N )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \rightsquigarrow_{\text{conf}} E \cup { x = M, M = N } )</td>
</tr>
<tr>
<td>coalescence:</td>
<td>( E \cup { x = y } ) if ( y \neq x )</td>
</tr>
<tr>
<td></td>
<td>( \rightsquigarrow_{\text{elim}} E[x := y] \cup { x = y } )</td>
</tr>
</tbody>
</table>

**Figure 6.4** Rational unification

To handle rational unification, the six transformations (erasure, inversion, decomposition, conflict, merge and coalescence) constitute a complete system which also satisfies the termination property.

6.2.4 Unification algorithms

The unification algorithms used by most Prolog interpreters are really inefficient. Most use Robinson's algorithm, which represents a system of equations as a stack, and which operates on the equation at the top of the stack unless no rule applies to
it: it has an exponential maximal complexity, as is shown by the following example:

\[ M_p = f(x_1, x_2, \ldots, x_p), \]
\[ N_p = f(g(x_0, x_0), g(x_1, x_1), \ldots, g(x_{p-1}, x_{p-1})). \]

The image of \( x_p \), which is a term of size \( 2^{p+1} - 1 \), with \( 2^p \) occurrences of variable \( x_0 \), is constructed explicitly, then searched to ensure that it does not contain \( x_p \). The explicit construction and the search lead to exponential complexities: spatial and temporal respectively.

It was first supposed that omitting the occur check would avoid the exponential temporal complexity. But the unification of

\[ M'_p = f(g(x_0, x_0), g(x_1, x_1), \ldots, g(x_{p-1}, x_{p-1}), y_1, \ldots, y_p, y_p), \]
\[ N'_p = f(x_1, \ldots, x_p, g(y_0, y_0), g(y_1, y_1), \ldots, g(y_{p-1}, y_{p-1}), x_p), \]

begins by constructing the images of \( x_p \) and \( y_p \), which are terms similar to those of the preceding example, then it must traverse them to unify \( y_p \) and \( x_p \), which requires exponential time to show \( 2^p \) times that \( x_0 = y_0 \).

This problem occurs because of an unshared implementation of terms. Corbin and Bidoit [26] showed that an implementation of terms by dags reduced the complexity to \( O(p^2) \), where \( p \) is the number of symbols to unify in the terms.

Unification is in fact a linear problem, as was shown by Paterson and Wegman [117], with an asymptotically linear algorithm which is not efficient for typical cases. Pseudo-linear algorithms have been given by Huet [67] and Martelli and Montanari [100], Escalada-Imaz and Ghallab [41] proposed an algorithm in the same spirit as that of Huet, but using smaller data structures.

As in logic programming, it is often important to fail as soon as possible. Unification algorithms privilege one of the two possible causes of failure: conflict or cycle. The Peterson–Wegman algorithm fails quickly in the case of cycle—a quite rare case in practice. Those of Huet and Escalada-Imaz–Ghallab privilege failure by conflict by delaying the occur check. The latter two adapt readily to rational terms. Those of Colmerauer and Martelli–Rossi [101] are explicitly designed for rational terms.

**Systems of multiequations** Martelli and Montanari structure a system of equations using multiequations by grouping terms by classes for the equivalence generated by the equations of the system.

**Example** The system \( \{x = z, y = f(x, x), f(x, a) = f(y, z), z = y\} \) defines two classes of terms: \( \{x, y, z, f(x, x)\} \) and \( \{f(x, a), f(y, z)\} \). It is convenient to distinguish the variables by writing the two multiequations \( \{x, y, z\} = \{f(x, x)\} \) and \( \{w\} = \{f(x, a), f(y, z)\} \), where \( w \) is a new variable. \( \square \)

A **multiequation** is an expression \( V = S \), where \( V \) is a non-empty set of variables and \( S \) is a multiset of non-variable terms. A **merge** of two multiequations \( V = S \)
and \( V' = S' \) such that \( V \cap V' \neq \emptyset \) consists of replacing these two multiequations by \( V \cup V' = S \cup S' \). The right-hand side of a multiequation is to be decomposed as soon as it contains several terms. Martelli and Montanari suggested that several decomposition steps be combined by computing the common part \( C(S) \) and the border \( F(S) \) of a multiset \( S \) of terms.

**Example** The common part of \( S = \{ f(x, g(x), k(y)), f(h(z), t, k(z)) \} \) is the term \( C(S) = f(x, t, k(y)) \). Its border is the system \( (F(S)) \) formed of three multiequations: \( \{x\} = \{h(z)\}, \{t\} = \{g(x)\}, \{y, z\} = \emptyset \). However, \( \{f(x, b), f(a, a)\} \) does not have a common part.

If the common part does not exist, there is failure by conflict. A decomposition step consists of replacing a multiequation \( V = S \) by \( V = \{C(S)\} \) and by all the multiequations of \( F(S) \).

It is useful to work with two systems \( T \) and \( U \). Initially, to unify terms \( M_1 \) and \( M_2 \), containing variables \( x_1, \ldots, x_n \), one defines

\[
U = \{\{u\} = \{M_1, M_2\}\}, \quad T = \{\{x_1\} = \emptyset, \ldots, \{x_n\} = \emptyset\},
\]

where \( u \) is a new variable.

The objective is to obtain \( U = \emptyset \) by transferring the multiequations of \( U \) to \( T \). During the decomposition of \( V = S \), the principle is to place \( V = \{C(S)\} \) in \( T \), possibly after a merge, and to add \( F(S) \) to \( U \). A multiequation of \( U \) is selected and it is then merged as much as possible with the multiequations of \( T \). If the result contains at least two terms on the right-hand side, it is decomposed as indicated; otherwise, it is placed in \( T \).

**Example**

\[
U = \{\{u\} = \{f(x, g(x), k(y)), f(h(z), t, k(z))\}, \{v\} = \{x, h(y)\}\} \quad T = \{\{x\} = \emptyset, \{y\} = \emptyset, \{z\} = \emptyset, \{t\} = \emptyset\}
\]

\( \leadsto \) \( U = \{\{x\} = \{h(z)\}, \{t\} = \{g(x)\}, \{y, z\} = \emptyset, \{v\} = \{x, h(y)\}\} \)

\( \leadsto \) \( T = \{\{u\} = \{f(x, t, k(y))\}, \{x\} = \{h(z)\}, \{t\} = \{g(x)\}, \{y, z\} = \emptyset\} \)

\( \leadsto \) \( U = \{\{v\} = \{h(y)\}\} \quad T = \{\{u\} = \{f(x, t, k(y))\}, \{x\} = \{h(z)\}, \{t\} = \{g(x)\}, \{y, z\} = \emptyset\} \)

\( \leadsto \) \( U = \{\{y, z\} = \emptyset\} \quad T = \{\{u\} = \{f(x, t, k(y))\}, \{v, x\} = \{h(z)\}, \{t\} = \{g(x)\}, \{y, z\} = \emptyset\} \)

\( \leadsto \) \( U = \emptyset \quad T = \{\{u\} = \{f(x, t, k(y))\}, \{v, x\} = \{h(z)\}, \{t\} = \{g(x)\}, \{y, z\} = \emptyset\} \).

The final system \( T \), in which the equations containing \( u \) and \( v \) can be ignored, is not in solved form: \( x \) must still be eliminated. Nevertheless this ‘triangular’ form
can be considered satisfactory as it is more compact than a solved form. Differentiating eliminations allows one to handle unification in $T'[^X]$ without worrying about cycles. However, if the search is restricted to solutions in $T[^X]$, tests must be made to ensure that there are no cycles. The simple occur check is not only inefficient, but insufficient, so long as the eliminations have not been done (for example: $x = f(y), y = g(x)$). The existence of cycles is tested by associating a counter to each multiequation, of $T$ only, if the test is final, enumerating the number of occurrences of variables of its left-hand side appearing in the other equations of $T$ and $U$. Those counters are modified each time the system is transformed. The value of the counter is written in square brackets.

**Example** For the preceding system, the initial counters are:

$$
U = \left\{ \begin{array}{l}
\{u\} = \{f(x), g(x), h(y), f(h(z), t, k(z))\}, \\
\{v\} = \{x, h(y)\}
\end{array} \right\},
$$

$$
T = \left\{ \begin{array}{l}
[3]\{x\} = \emptyset, \\
[2]\{y\} = \emptyset, \\
[2]\{z\} = \emptyset, \\
[1]\{t\} = \emptyset
\end{array} \right\}.
$$

A system is in solved form if and only if all of its counters are zero. Conversely, if there is no cyclic dependency, there is at least one multiequation $[c]V = S$ whose counter $c$ is zero, and the variables of $V$ can be eliminated—without making any substitution, since they have no occurrences elsewhere. One can then ignore this multiequation, thereby decrementing the counters of the variables appearing in its right-hand side $S$.

**Example (cont.)** The final system, without its auxiliary variables $u$ and $v$, is:

$$
T = \{ \begin{array}{l}
[1]\{x\} = \{h(z)\}, \\
[0]\{t\} = \{g(x)\}, \\
[1]\{y, z\} = \emptyset.
\end{array} \}.
$$

Start by eliminating $t$, then $x$, whose counter has become zero, and $y, z$ last, giving systems $T$:

$$
\begin{array}{l}
[0]: \{x\} = \{h(y)\} \\
[0]^1:\{t\} = \{g(x)\} \\
[1]: \{y, z\} = \emptyset
\end{array} \quad \begin{array}{l}
[0]^2: \{x\} = \{h(y)\} \\
[0]^1:\{t\} = \{g(x)\} \\
[0]^3: \{y, z\} = \emptyset.
\end{array}
$$

Written as exponent of the counter is the order in which the multiequations are eliminated. This order allows one to construct the solved form explicitly: first $y = z$, then $x = h(y) = h(z)$, then $t = g(x) = g(h(z))$. □

If it is important to fail as early as possible by cycle detection, this simple computation can be done without having reached the final form of the system: a
multi-equation having been selected in \( U \), then merged with multi-equations in \( T \),
the decomposition is done only if the associated counter is zero.

The representative of a variable Escalada-Imaz and Ghalab's algorithm
constructs a graph whose vertices are tagged by terms. To each variable \( x \) corresponds
a vertex \( u_x \). Every connected component corresponds to a multi-equation of the
form \( \{ x \} = \{ M \} \), or else \( \{ x \} = \emptyset \): it contains at most one non-variable vertex,
and, if it contains one, has no cycle. The initial graph \( G_0 \) is formed of one vertex
per variable, with no edges.

In a given configuration of the graph, to each variable \( x \) is associated its representative \( \text{rep}(x) \): if \( u_x \) does not have a successor, then \( \text{rep}(x) = x \). Otherwise,
by following the path starting at \( u_x \), \( \text{rep}(x) \) is the last variable if the path does
not contain any cycles, or the last variable visited, should there be a cycle. To
find this representative practically, a marking of visited vertices is necessary. This
marking is also useful when the graph is traversed from one connected component
to another, to simulate the elimination of a variable.

There are four possibilities, which are written in the following manner:

\[
\begin{align*}
x: & \quad u_x \text{ has no successor;} \\
x \Rightarrow \text{rep}(x): & \quad u_{\text{rep}(x)} \text{ has no successor;} \\
x \Rightarrow \text{rep}(x) \rightarrow X: & \quad u_{\text{rep}(x)} \text{ has a successor } u_X, \text{ with } X \text{ non-variable;} \\
x \Rightarrow \text{rep}(x) \circ: & \quad \text{the graph has a cycle, the successor of } u_{\text{rep}(x)} \text{ is the vertex} \\
& \quad \text{of a variable already traversed, starting from } x.
\end{align*}
\]

Write \( x \ldots \) for the first of these cases and \( x \Rightarrow \text{rep}(x) \ldots \) for one of the last three
cases.

Transformations operate over pairs \((E, G)\) of a system of equations and a graph.
Decomposition is only used for the system of equations:

\[
\begin{align*}
\text{decomposition: } & (E \cup \{ f \bar{M} = f \bar{N} \}, G) \xrightarrow{\text{dec}} (E \cup \{ \bar{M} = \bar{N} \}, G). \\
\text{conflict: } & (E \cup \{ f \bar{M} = g \bar{N} \}, G) \xrightarrow{\text{conflict}} \perp, \text{if } f \neq g.
\end{align*}
\]

When an equation \( x = M \) is selected in \( E \), there is, depending on the case, either
graph construction or merging.

If \( M \) is non-variable, there are three cases, depending on \( x \):

\[
\begin{align*}
x & \\
x \Rightarrow \text{rep}(x) & \\
x \Rightarrow \text{rep}(x) \circ & \\
x \Rightarrow \text{rep}(x) \rightarrow X & \\
x \rightarrow M &
\end{align*}
\]

If \( M \) is a variable \( y \), there are three more cases, depending on \( y \) (plus the symmetric
cases in \( x/y \):

\[
\begin{align*}
\text{x} & \rightarrow \text{rep}(x) \quad \text{y} \ldots \\
\text{x} & \rightarrow \text{rep}(x) \cup \text{y} \ldots \\
x \rightarrow \text{rep}(x) \rightarrow X & \quad y \rightarrow \text{rep}(y) \rightarrow Y \\
\overline{x \rightarrow \text{rep}(x) \rightarrow y} & \quad \overline{x \rightarrow \text{rep}(x) \rightarrow y \rightarrow \text{rep}(y) \rightarrow Y} \\
\end{align*}
\]

and \( E \cup \{ x = M \} \sim E \cup \{ X = Y \} \).

In each case where a representative must be computed, a box \( \overline{x \ldots z} \) indicates that the path \( x \ldots z \) must be compressed, i.e., replaced by a tree of depth 1 and root \( z \), the other vertices on the path being linked to \( z \). In the cases where a cycle \( \cup \) is detected, it must be unwound.

By applying these transformations a finite number of times, the pair \( (E, G_0) \) is transformed into \( (\emptyset, G) \).

**Example** Suppose two terms

\[
\begin{align*}
M & \equiv f(h(x_1, x_2, x_3), h(x_6, x_7, x_8), x_3, x_6), \\
N & \equiv f(h(g(x_4, x_5), x_1, x_2), h(x_7, x_8, x_5), g(x_5, a), x_5),
\end{align*}
\]

must be unified. The processing of equation \( M/1 = N/1 \) gives a first component of the graph:

\[
x_3 \rightarrow x_2 \rightarrow x_1 \rightarrow g(x_4, x_5).
\]

Equation \( M/2 = N/2 \) gives the cycle

\[
x_6 \rightarrow x_7 \rightarrow x_8 \cup [x_6].
\]

Equation \( M/3 = N/3 \) requires finding the representative of \( x_3 \), which is \( x_1 \), compressing the path \( x_3 \ldots x_1 \), which after merging the equation gives \( g(x_4, x_5) = g(x_5, a) \), which creates a third component:

\[
x_4 \rightarrow x_5 \rightarrow a.
\]

The last equation \( x_6 = x_5 \) requires finding the representative of \( x_6 \), which is \( x_8 \), compressing the path \( x_6 \ldots x_8 \), and merging the second and third components. The resulting graph has two connected components:

\[
\begin{array}{c}
x_3 \\
\downarrow \\
x_1 \rightarrow g(x_4, x_5) \\
\uparrow \\
x_2
\end{array} \quad \begin{array}{c}
x_6 \\
\downarrow \\
x_8 \rightarrow x_5 \rightarrow a \\
\uparrow \\
x_7 \quad x_4
\end{array}
\]

The graph \( G \) having been constructed, the absence of cycle must still be tested, and the unifier \( \theta \) must be computed, i.e., for each variable \( x \), \( \theta(x) \). To each vertex
of the graph is associated a value \( \text{val}(u) \), which can be a (pointer to a) term, the symbols \( \text{nil} \) or \( \text{cycle} \): this value is used to memoize the computed results and to mark the traversed vertices for the detection of cycles. Initially, \( \text{val}(u_x) = \text{nil} \) for all variables:

- if \( \text{val}(u_x) = M \): return \( M \);
- if \( \text{val}(u_x) = \text{cycle} \): halt, failure by cycle;
- otherwise, find \( \text{rep}(x) \); let \( t \) be a local variable:
  - if \( \text{val}(u_{\text{rep}(x)}) = M \): let \( t := M \);
  - if \( \text{val}(u_{\text{rep}(x)}) = \text{cycle} \): halt, failure by cycle;
  - otherwise, in cases \( x \rightarrow \text{rep}(x) \) and \( x \rightarrow \text{rep}(x) \odot \), let \( t := \text{rep}(x) \);
  - if \( x \rightarrow \text{rep}(x) \rightarrow M \), with \( M \) closed, let \( t := M \);
  - if \( x \rightarrow \text{rep}(x) \rightarrow M \), with \( M \) not closed, then:
    * mark each vertex \( u \) of the path \( u_x \ldots u_{\text{rep}(x)} \) by \( \text{val}(u) := \text{cycle} \),
    * compute \( \theta(y) \) for each \( y \in \text{var}(M) \),
    * let \( t := \theta(M) \);

for each vertex \( u \) of the path \( u_x \ldots u_{\text{rep}(x)} \), let \( \text{val}(u) := t \) and return \( t \).

Example (cont.) The computation of \( \theta(x_2) \) determines the representative of \( x_2 \) \( (x_1) \), marks the vertices \( x_2 \) and \( x_1 \) by \( \text{cycle} \), and computes \( \theta(x_4) \) and \( \theta(x_5) \), both equaling \( a \). After this computation, the vertices \( x_2 \) and \( x_1 \) receive the value \( g(a, a) \), and \( x_4, x_5 \) the value \( a \). The graph need not be traversed to compute \( \theta(x_1), \theta(x_4) \) and \( \theta(x_5) \).

### 6.2.5 Critical pairs

The Knuth–Bendix confluence criterion, an application of unification, is important to the study of rewriting systems.

Let \( P \) and \( P' \) be two terms. Term \( P' \) is superposable upon \( P \) at occurrence \( u \in \mathcal{O}(P) \) if \( P(u) \notin X \) and if \( P/u \) and \( P' \) are unifiable.

Let \( P \rightarrow Q \) and \( P' \rightarrow Q' \) be two separate rewriting rules, i.e., their sets of variables are disjoint. The rewriting rule \( P' \rightarrow Q' \) is superposable upon \( P \rightarrow Q \) at occurrence \( u \in \mathcal{O}(P) \) if \( P' \) is superposable upon \( P \) at \( u \). For every unifier \( \sigma \) of \( P/u \) and \( P' \), there are the two reductions \( \sigma P \overset{u}{\rightarrow} \sigma Q \) and \( \sigma P \overset{u}{\rightarrow} \sigma P[u \leftarrow \sigma Q'] \).

**Definition 6.8** The pair \( (\theta Q, \theta P[u \leftarrow \theta Q']) \) is a critical pair of \( \mathcal{R} \), when \( \theta \) is a most general unifier of \( P/u \) and \( P' \).

Note that there are finitely many critical pairs in a finite rewriting system.
Example  The associativity rule \((xy)z \rightarrow x(yz)\) superposes upon itself at occurrence 1, xy and \((x'y')z'\) being unifiable, giving the two reductions:
\[
\begin{align*}
((x'y')z')z & \overset{*}{\rightarrow} (x'y')(z'z), \\
((x'y')z')z & \overset{1}{\rightarrow} (x'(y'z'))z
\end{align*}
\]
hence the critical pair \(((x'y')(z'z), (x'(y'z'))z)\).

The existence of critical pairs is a sign of the ‘ambiguity’ of a rewriting system.

Theorem 6.15 (Knuth–Bendix, Huet) If \(M_1 \leftarrow M \rightarrow M_2\), then

- either there exist reductions \(M_1 \overset{*}{\rightarrow} M' \overset{*}{\leftarrow} M_2\)
- or there exists a critical pair \((C_1, C_2)\) of \(R\), a context \(K[\_]\) and a substitution \(\mu\) such that \(M_i \equiv K[\mu C_i]\) for \(i = 1, 2\).

Proof Let \(M \overset{u_1,v_1}{\rightarrow} M_1\) and \(M \overset{u_2,v_2}{\rightarrow} M_2\). Distinguish the relative positions of occurrences \(u_1\) and \(u_2\).

- If \(u_1\) and \(u_2\) are independent, then \(M_1\) and \(M_2\) rewrite to a term \(M'\) by the reductions \(M_1 \overset{u_2,v_2}{\rightarrow} M'\) and \(M_2 \overset{u_1,v_1}{\rightarrow} M'\).
- If \(u_1 \leq u_2\), it can be supposed that \(u_1 = \varepsilon\). Then \(M \equiv \sigma_1 P_1\), hence \(M \overset{\leq}{\rightarrow} \sigma_1 Q_1\), and \(M/u_2 \equiv \sigma_2 P_2\), \(M \overset{u_2}{\rightarrow} M[u_2 \leftarrow \sigma_2 Q_2]\). There are two cases according to the nature of \(u_2\).
  - If \(u_2\) is an internal occurrence of \(P_1\), i.e. \(u_2 \in \mathcal{O}(P_1)\) and \(P_1(u_2) \notin X\). Suppose that the rules \(\rho_1\) and \(\rho_2\) are separate. Since \((\sigma_1 P_1)/u_2 \equiv \sigma_2 P_2\), the substitution \(\sigma\), sum of \(\sigma_1\) and \(\sigma_2\), is a unifier of \(P_1/u_2\) and \(P_2\). By assumption on \(u_2\), \(\rho_2\) superposes itself on \(\rho_1\), which yields the critical pair \((C_1, C_2)\) where \(C_1 \equiv \theta Q_1\) and \(C_2 \equiv \theta(P_1[u_2 \leftarrow Q_2])\), \(\theta\) being a most general unifier of \(P_1/u_2\) and \(P_2\). Let \(\mu\) be a substitution such that \(\sigma = \mu \theta\). Then \(M_1 \equiv \sigma_1 Q_1 \equiv \mu C_1\) and \(M_2 \equiv M[u_2 \leftarrow \sigma_2 Q_2] \equiv \mu C_2\).
  - If \(u_2 \in \mathcal{O}(\sigma P_1)\) is not an internal occurrence of \(P_1\), then \(u_2 = w'u',\) with \(w' \in \mathcal{O}(P_1), P_1/w' \equiv x \in X,\) and \(u' \in \mathcal{O}(\sigma(x))\). Let \(W_1\) be the set of occurrences of \(x\) in \(P_1\), and \(U_2 = \{wu'; w \in W_1\} \subset \mathcal{O}(M)\). Application of rule \(\sigma P_2 \rightarrow \sigma Q_2\) to occurrence \(u_2\), then to other occurrences, pairwise independent, of \(U_2,\) yields successively:
\[
M \equiv \sigma P_1 \overset{u_2}{\rightarrow} M_2 \equiv (\sigma P_1)[u_2 \leftarrow \sigma Q_2], \overset{*}{\rightarrow} M'_2 \equiv (\sigma P_1)[U_2 \leftarrow \sigma Q_2].
\]
Let \(\sigma'\) be the substitution obtained by replacing \(\sigma(x)\) by \(\sigma'(x) \equiv \sigma(x)[u' \leftarrow \sigma Q_2]\).

Then \(M'_2 \equiv \sigma' P_1\). Rule \(P_1 \rightarrow Q_1\) can then be applied:
\[
M'_2 \equiv \sigma' P_1 \overset{*}{\rightarrow} M' \equiv \sigma' Q_1.
\]
$M'$ can also be obtained from $M_1$,
\[
M_1 \equiv \sigma Q_1 \xrightarrow{\ast} (\sigma Q_1)[U_1 \leftarrow \sigma Q_2] \equiv \sigma' Q_1,
\]
where $U_1$ is the set of occurrences of $\sigma P_2$ in $\sigma Q_1$. Hence $M_1 \xrightarrow{\ast} M' \xleftarrow{\ast} M_2$. This case is illustrated by Figure 6.5.

\[\text{Figure 6.5 Proof of the Knuth–Bendix theorem}\]

**Corollary 6.16** A rewriting system is locally confluent if and only if every critical pair is confluent.

Combining Newman’s lemma with the preceding corollary yields D. E. Knuth and P. B. Bendix’s original statement:

**Corollary 6.17** A noetherian rewriting system is confluent if and only if every critical pair is confluent.

This theorem will be used in the third part of the chapter to normalize equational proofs by ‘completing’ a rewriting system. Another application of the concept of critical pair is the Huet–Lévy confluence criterion, a simplified form of which is given here: a rewriting system whose left members are linear terms (at most one occurrence of each variable) and which has no critical pairs is confluent.
6.3 Equational proofs

6.3.1 Equations and rewriting

Since a set of equations $E$ defines an equivalence relation over terms, the problem $E \vdash M = N$ can be viewed as determining if $M$ and $N$ belong to the same equivalence class. Should there be a canonical representative for each class, and if for each element one can associate the canonical representative of its class, then the equivalence problem simply consists of determining the identity of canonical representatives. For example, to decide whether two fractions are equivalent, it suffices to simplify them, then to test the equality of numerators and denominators.

This aim is fulfilled by a rewriting system, assuming it satisfies the weak normalization property (every term has a normal form), that it be confluent (hence that the normal form be unique) and that there exists an effective strategy to attain the normal form of a term. This last condition is satisfied if the system is noetherian. A confluent and noetherian rewriting system is convergent or canonical.

To use a canonical rewriting system $R$ in equational proofs, the following properties must be satisfied:

- $R$ is sound relative to $E$ if $M \xrightarrow{\ast} R N$ implies $M =_E N$;
- $R$ is complete relative to $E$ if $M =_E N$ implies the syntactic equality of the normal forms $\overline{M}$ and $\overline{N}$ for $R$;
- $R$ is canonical relative to $E$ if it is convergent, sound and complete relative to $E$.

Given $E$, a canonical $R$ relative to $E$ is sought.

**Example** The axioms of group theory are:

$$
\begin{align*}
\text{(groups)} & \quad \begin{cases} 
ex &= x \\
x^{-1}x &= e \\
(xy)z &= x(yz).
\end{cases}
\end{align*}
$$

A canonical rewriting system for this theory is formed by the following ten rules:

$$
\begin{align*}
ex & \to x & xe & \to x \\
x^{-1}x & \to e & xx^{-1} & \to e \\
(xy)z & \to x(yz) & e^{-1} & \to e \\
x^{-1}(xy) & \to y & x(x^{-1}y) & \to y \\
(x^{-1})^{-1} & \to x & (xy)^{-1} & \to y^{-1}x^{-1}.
\end{align*}
$$

It is easy to construct a sound system, simply by orienting the equations of $E$ into rules. Such a system will rarely be complete. For example, the natural orientation of the axioms for groups, $ex \to x$, $x^{-1}x \to e$, $(xy)z \to x(yz)$ is not complete, since
the term \( xx^{-1} \) is irreducible, although \( xx^{-1} = e \) is a semantic consequence of the axioms. It is noetherian, but not confluent, since critical pairs exist. A sound and complete system would be obtained by taking the two possible orientations of each relation, the two relations \( \leftarrow \rightarrow_\mathcal{R} \) and \( \equiv_\mathcal{E} \) then being identical. But such a system is never noetherian, since there are cycles \( P \rightarrow Q \rightarrow P \). An intermediate solution is therefore required.

**Mixed proofs** The Knuth–Bendix completion procedure attempts to transform a system of equations \( \mathcal{E}_0 \) into a rewriting system \( \mathcal{R} \) which is convergent, sound and complete relative to \( \mathcal{E} \). Since the procedure constructs \( \mathcal{R} \) step by step, it operates over pairs \( (\mathcal{E}, \mathcal{R}) \) starting from \( (\mathcal{E}_0, \varnothing) \), and it succeeds if it reaches \( (\varnothing, \mathcal{R}_\infty) \). At each step, \( \mathcal{R} \) is sound relative to \( \mathcal{E}_0 \) and is noetherian. Better, all the \( \mathcal{R} \) are included in the same well-order.

The resulting proofs will be **mixed**, using a pair \( \Gamma = (\mathcal{E}, \mathcal{R}) \) of sets of equations and of rules. Write \( \pi :_\Gamma M = N \) if \( \pi \) is a derivation of \( M = N \) in \( \Gamma \). Rather than use the equational inference system freely, only ‘linearized’ derivations will be used (this is an easy result of proof normalization): a derivation of \( M \) is represented by a sequence of terms \( \pi = (M_0 \equiv M, M_1, \ldots, M_p \equiv N) \), with, for each pair \( (M_i, M_{i+1}) \), \( 0 \leq i < p \), an ‘elementary’ derivation of \( M_i \rightarrow_\mathcal{R} M_{i+1} \), or of \( M_i \leftarrow_\mathcal{E} M_{i+1} \). Each of these elementary derivations must be specified by a rule or an equation, the occurrence where it is applied and the matching substitution.

### 6.3.2 Knuth–Bendix completion

**Complexity of proofs** Two proofs of \( M = N \) are represented in Figures 6.6 and 6.7. A proof by confluence is simpler than an arbitrary proof.

If \( \mathcal{E} \) is empty, and if \( \mathcal{R} \) is confluent, the Church–Rosser property states that any proof of \( M = N \) can be transformed into a proof by confluence. This lemma is a result of proof normalization. Similar results have already been evoked: cut-elimination in sequent calculus and normalization in natural deduction. In the last
\[ M \equiv N_0 \]
\[ N_1 \quad N_2 \quad N_3 \quad N_4 \quad N_5 \equiv N \]

Figure 6.7 A proof by confluence of \( M = N \)

case, the proofs were typed \( \lambda \)-terms, by the Curry–Howard correspondence, and they were normalized by contracting the redexes (i.e., an introduction followed by an elimination).

Here, the normalization of the proof of an equation consists essentially of replacing an effluence \( M_1 \leftarrow M_0 \rightarrow M_2 \) by a confluence \( M_1 \xrightarrow{\pi} M' \xleftarrow{\pi} M_2 \). This is not possible if \( E \) is non-empty and if \( \mathcal{R} \) is not confluent. Bachmair, Dershowitz and Hsiang’s idea was to see the completion procedure as a transformation of the axioms system in order to enable the writing of more and more normal proofs. It is well known that the same theory can have, depending on the set of axioms chosen, proofs of various complexities: for example, the equational axiom \( x^{-1}x = e \) allows simpler proofs than \( \forall x \exists y (yx = e) \).

The complexity \( \|\pi\| \) of an equational proof \( \pi \) is defined uniquely through its ‘shape’. The complexity of an elementary proof is a multiset of terms:

\[
\| M \rightarrow_{\mathcal{R}} N \| = \{ M \}.
\]
\[
\| M \leftarrow_{\mathcal{E}} N \| = \{ M, N \}.
\]

The complexity of a proof \( (M_0, \ldots, M_p) \) of \( M_0 = M_p \) is the multiset of the complexities of the elementary proofs making it up:

\[
\| M_0, \ldots, M_p \| = \{ \| M_0 \|, \| M_1 \|, \ldots, \| M_{p-1} \|, \| M_p \| \}.
\]

**Example** The complexity of the proof in Figure 6.6 is

\[
\{\{ M_0, M_1 \}, \{ M_1 \}, \{ M_3 \}, \{ M_4 \}, \{ M_4 \}, \{ M_5, M_6 \}, \{ M_6 \}, \{ M_7 \}, \{ M_9 \} \}.
\]

That of the proof by confluence in Figure 6.7 is:

\[
\{\{ N_0 \}, \{ N_1 \}, \{ N_4 \}, \{ N_4 \}, \{ N_5 \} \}.
\]

Note that an equational proof has a complexity which lies between that of a proof by effluence and a proof by confluence.
Since a reduction order (i.e., a well-founded order compatible and stable under substitutions) $\rightarrow$ has been defined over terms, it is extended to multisets of terms, then to the multisets of those multisets. Define $\pi \rightarrow^p \pi'$ if $||\pi|| \not\geq ||\pi'||$: it is a well-founded order, since it is constructed using extensions to multisets of the well-founded order $\rightarrow$.

Normalization of proofs Normalizing a proof requires contraction schemes which reduce its complexity. For example, a proof $\ldots M \Leftarrow M \ldots$, using axiom $P = P$, can be simplified to $\ldots M \ldots$, which is $\not\rightarrow^p \ldots M \Leftarrow M \ldots$. This axiom can even be removed without affecting the equality $\equiv$. In other cases, either $\mathcal{E}$ or $\mathcal{R}$ will have to be extended before proofs can be simplified.

The idea is to define a family of transformations $(\mathcal{E}, \mathcal{R}) \rightsquigarrow (\mathcal{E}', \mathcal{R}')$ such that:

- if $\pi : \Gamma M = N$ is a non-normal proof, there exists a transformation $\Gamma \rightsquigarrow \Gamma'$ and a proof $\pi' : \Gamma' M = N$ such that $\pi \rightarrow^p \pi'$;
- for every transformation $\Gamma \rightsquigarrow \Gamma'$ and every $\pi : \Gamma M = N$, there exists $\pi' : \Gamma' M = N$ such that $\pi \not\rightarrow^p \pi'$.

Once these transformations have been chosen, a completion procedure for $\mathcal{E}_0$ will construct a sequence $\Gamma_0, \Gamma_1, \ldots$ of pairs $(\mathcal{E}_i, \mathcal{R}_i)$, with $\Gamma_0 = (\mathcal{E}_0, \emptyset)$, such that for every non-normal proof $\pi : \Gamma_i M = N$, there exists $\pi' : \Gamma_j M = N$ for $j \geq i$ such that $\pi \not\rightarrow^p \pi'$:

First group of transformations The first group operates over the structure of proofs:

| orientation | $\mathcal{E} \cup \{P = Q\}, \mathcal{R} \rightsquigarrow \mathcal{E}, \mathcal{R} \cup \{P \rightarrow Q\}$ if $P \not\rightarrow Q$ | $\mathcal{E} \cup \{P = Q\}, \mathcal{R} \rightsquigarrow \mathcal{E}, \mathcal{R} \cup \{Q \rightarrow P\}$ if $Q \not\rightarrow P$ |
| erasure | $\mathcal{E} \cup \{M = M\}, \mathcal{R} \rightsquigarrow \mathcal{E}, \mathcal{R}$ |
| effluence | $\mathcal{E}, \mathcal{R} \rightsquigarrow \mathcal{E} \cup \{M_1 = M_2\}, \mathcal{R}$ if $M_0$ | $M_1 \Downarrow M_2$ |
| normalization | $\mathcal{E} \cup \{M = N\}, \mathcal{R} \rightsquigarrow \mathcal{E} \cup \{M' = N'\}, \mathcal{R}$ if $M \not\leftrightarrow^p \mathcal{R} M'$ and $N \not\leftrightarrow^p \mathcal{R} N'$ |

Note first that

- each of the transformations preserves equality $\leftrightarrow_{\mathcal{E}, \mathcal{R}}$;
- the rewriting systems are all noetherian for the same order $\rightarrow$; and
- the transformations can only make $\mathcal{R}$ grow.
Equational proofs 243

Each transformation $\Gamma \rightarrow \Gamma'$ defines a transformation over proofs, associating with each proof $\pi$ of $M = N$ in $\Gamma$ a proof $\pi'$ of the same equation in $\Gamma'$, obtained by 'contracting' a subproof (analogous to the contraction of a redex in the $\lambda$-calculus). If $\pi$ does not use an axiom affected by the transformation, then $\pi' = \pi$. Otherwise, an equational proof

$$\ldots C[\theta P] \leftrightarrow C[\theta Q] \ldots$$

is replaced, for each transformation, by:

- erasure:
  $$\ldots C[\theta P] \ldots$$

- normalization:
  $$\ldots C[\theta P] \rightarrow C[\theta P'] \leftrightarrow C[\theta Q] \ldots$$

- orientation:
  $$\ldots C[\theta P] \rightarrow C[\theta Q] \ldots$$

and an effluence proof $M_1 \leftarrow M_0 \rightarrow M_2 \ldots$ by the equational proof $M_1 \leftrightarrow M_2$. The proof $\pi'$ obtained is strictly simpler, since:

$$\{ \ldots \{ P, P \} \ldots \} \Rightarrow \{ \ldots \},$$

$$\{ \ldots \{ M, N \} \ldots \} \Rightarrow \{ \ldots \{ M \}, \{ M', N \} \ldots \},$$

since $M \succ M'$,

$$\{ \ldots \{ P, Q \} \ldots \} \Rightarrow \{ \ldots \{ P \} \ldots \},$$

$$\{ \ldots \{ M_0 \}, \{ M_0 \} \ldots \} \Rightarrow \{ \ldots \{ M_1, M_2 \} \ldots \}.$$  

In all cases, $\pi \succ^p \pi'$.

What is required is to strictly reduce the complexity of $\pi$ if $\pi$ is not a proof by confluence. This is possible by supposing that at least one equation used by $\pi$, both sides of which are distinct and irreducible by $R$, is orientable. Sure enough, at least one equational subproof of $\pi$ can be contracted, by erasure, normalization or orientation of an equation and every subproof by effluence can be transformed into an equation.

This last transformation is, however, unnecessary. Consider a subproof $M_1 \leftarrow M_0 \rightarrow M_2$ of $\pi$. If there exists a proof by confluence $M_1 \leftarrow M_0 \leftarrow M_2$ in $R$, it is strictly simpler, since $M_0$ is strictly greater than each of the terms which appear in the reductions $M_1 \leftarrow M_0 \leftarrow M_2$, by assumption on the reduction order $\succ$. In this case, no transformation of the axioms is necessary. Otherwise, according to the Knuth–Bendix theorem, then $M_1 \equiv C[\theta P]$, $M_2 \equiv C[\theta Q]$, where $(P, Q)$ is a critical pair of $R$. After adding the equation $P = Q$ to $E$, $C[\theta P] \leftarrow M_0 \rightarrow C[\theta Q]$ can be replaced by $C[\theta P] \leftrightarrow C[\theta Q]$, which is strictly simpler, since $M_0 \succ M_1, M_2$.

**Fairness** An equation (or rule, or critical pair) is called persistent if it appears in all of the $\Gamma_i$ above a certain rank. A sequence $(\Gamma_n)_{n \geq 0}$ is fair for the first group of transformations if no equation is persistent, and if every critical pair created is later transformed into an equation. In other words, every equation of $E_i$ will later be erased, normalized or oriented, and every critical pair created by the sequence will ultimately be transferred to the equational part, and then erased, normalized or oriented. Fairness is an important property of processes, as are confluence and termination, in the case of non-determinism, to avoid the possibility of a choice being offered an infinite number of times without it ever being taken.
Proposition 6.18 If the sequence \((\Gamma_n)_{n \geq 0}\) is fair and if \(M =_{\Gamma_i} N\), there exists a step \(k \geq i\) such that \(M = N\) admits a proof by confluence in \(\mathcal{R}_k\).

The proof of a stronger theorem will be given later.

Corollary 6.19 If the sequence \((\Gamma_n)_{n \geq 0}\) is fair, then the rewriting system \(\mathcal{R}_\infty = \bigcup_{n \geq 0} \mathcal{R}_n\) is sound, canonical and complete for \(\mathcal{E}_0\).

Proof The system \(\bigcup_{n \geq 0} \mathcal{R}_n\) is sound relative to \(\mathcal{E}_0\), since each \(\mathcal{R}_n\) is. It is noetherian since the \(\mathcal{R}_n\) are uniformly noetherian, i.e., for every \(n\), \(M \rightarrow_{\mathcal{R}_n} N\) implies \(M \succ N\), the reduction order \(\succ\) being independent of \(n\).

If \((P, Q)\) is a critical pair of \(\bigcup_{n \geq 0} \mathcal{R}_n\), there exists an \(i\) such that \((P, Q)\) is a critical pair of \(\mathcal{R}_i\), and the proposition shows that \((P, Q)\) join for an \(\mathcal{R}_k\), hence \(\bigcup_{n \geq 0} \mathcal{R}_n\). Confluence is a result, according to the Knuth–Bendix theorem, and so \(\mathcal{R}_\infty\) is convergent.

If \(M =_{\mathcal{E}_0} N\), the proposition shows that there exists a \(k\) such that \(M =_{\mathcal{R}_k} N\), hence \(M =_{\mathcal{R}_\infty} N\). It follows that the normal forms of \(M\) and \(N\) in \(\mathcal{R}_\infty\) are identical. This proves the completeness of \(\mathcal{R}_\infty\) relative to \(\mathcal{E}_0\). □

A completion procedure for \(\mathcal{E}\) which constructs a fair sequence is therefore a semi-decision procedure. Let \(M\) and \(N\) be two terms. Since each system \(\mathcal{R}_i\) is noetherian and finite, let \(M_i\) (respectively \(N_i\)) be the finite set of irreducible terms obtained by reduction of \(M\) (respectively \(N\)) in \(\mathcal{R}_i\), and \(P_i = M_i \cap N_i\). If \(M =_{\mathcal{E}} N\), there exists an \(i\) and a term \(P\) such that \(M \rightarrow P \Leftarrow N\) in \(\mathcal{R}_i\), hence \(P\) is non-empty. The semi-decision procedure consists in computing \(P_i\) until \(P_i \neq \emptyset\): it halts if and only if \(M =_{\mathcal{E}} N\).

Inter-simplifications The first group of transformations allows an inefficient, non-deterministic completion procedure. It is preferable to simplify the created rules on the fly. Otherwise, the probable result will be a redundant system.

A rule \(P \rightarrow Q\) is reduced in \(\mathcal{R}\) if \(P\) is irreducible for \(\mathcal{R} \setminus \{P \rightarrow Q\}\), and \(Q\) is irreducible for \(\mathcal{R}\). A rewriting system \(\mathcal{R}\) is reduced if each of its rules is reduced. What can one do with an unreduced system?

If \(Q \vdash Q'\) (in particular \(Q \succ Q'\)), the rule \(P \rightarrow Q\) should be replaced by \(P \rightarrow Q'\), which is simpler.

If \(P\) is reducible by a rule \(R \rightarrow S\), then a subterm of \(P\) is an instance of \(R\): \(P/u \equiv \theta R\). The critical pair \((Q, P[u \leftarrow \theta S])\) is therefore obtained, but it is a particular case of computing critical pairs, where the substitution \(\theta\) links only variables of \(R\). It follows that the rule \(R \rightarrow S\), which is applicable each time that \(P \rightarrow Q\) is applicable, is simpler than \(P \rightarrow Q\).

The search for reduced systems leads to the use of the strict order \(\succ^R\) over rules, defined by: \(P \rightarrow Q \succ^R R \rightarrow S\) if \(P \equiv R\) and \(Q \succ S\), or if a subterm of \(P\) is an instance of \(R\) and \(P \neq R\). It is a well-founded order, since matching and the relation ‘is a subterm of’ are well-founded. A system is reduced as soon as every rule is minimal under \(\succ^R\).
A second group of transformations is therefore used. It operates by intersimplification of the rules to obtain a reduced system:

\[
\begin{align*}
\text{inter-r} & : \mathcal{E}, \mathcal{R} \cup \{P \rightarrow Q\} \leadsto \mathcal{E}, \mathcal{R} \cup \{P \rightarrow Q'\} \quad \text{if } Q \triangleleft_{\mathcal{R}} Q' \\
\text{inter-l} & : \mathcal{E}, \mathcal{R} \cup \{P \rightarrow Q\} \leadsto \mathcal{E} \cup \{P' = Q\}, \mathcal{R} \quad \text{if } P \overset{\rho}{\rightarrow} P' \quad P \rightarrow Q \triangleright_{\mathcal{R}} \rho
\end{align*}
\]

Note that one can always reduce the right-hand side of a rule, while the reduction of a left-hand side transforms the rule into an equation, which will often be deleted and sometimes oriented in the opposite direction to the original rule. As for the first group of transformations, these allow the contraction of subproofs.

A proof by rewriting \(\pi\),

\[
\ldots C[\theta P] \rightarrow C[\theta Q] \ldots
\]

is replaced, for each transformation, by \(\pi'\):

\[
\begin{align*}
\text{inter-r:} & \quad \ldots C[\theta P] \rightarrow C[\theta Q'] \triangleleft C[\theta Q] \ldots \\
\text{inter-l:} & \quad \ldots C[\theta P] \triangleright C[\theta P'] \leftrightarrow C[\theta Q'] \ldots
\end{align*}
\]

However, \(\pi \triangleright^p \pi'\) does not necessarily hold in the order on proofs. This order will be refined to consider not only the shape of the proofs, but also their contents, by using the order \(\triangleright^R\) over the rules.

The fine complexity of an elementary proof is a pair formed of a multiset and a rule or _ (irrelevant value):

\[
\begin{align*}
|| M \overset{\rho}{\rightarrow} N || & = (\{\{M\}\}, \rho), \\
|| M \leftrightarrow_{\varepsilon} N || & = (\{\{M, N\}\}, \_).
\end{align*}
\]

These pairs are compared using the lexical ordering, using the extension to multisets of the order \(\triangleright\) over terms in the first component, and the order \(\triangleright^R\) over the rules in the second component. The fine complexity of a proof \((M_0, \ldots, M_p)\) of \(M_0 = M_p\) is still the multiset of the complexities of the elementary proofs making it up:

\[
|| M_0, \ldots, M_p || = \{|| M_0, M_1 ||, \ldots, || M_{p-1}, M_p ||\}.
\]

The fine complexities of proofs are compared by extension to multisets of the order \(\triangleright\) over pairs. Define \(\pi \triangleright^P \pi'\) if \(|| \pi || \triangleright || \pi' ||\). It is also a well-founded order, since it is constructed by lexical product and extensions to multisets of the well-orders \(\triangleright\) and \(\triangleright^R\). It is then easy to show:

**Proposition 6.20**

- \(\pi \triangleright^P \pi'\) implies \(\pi \triangleright^P \pi'\);
- if \(\pi\) is transformed to \(\pi'\), then \(\pi \triangleright^P \pi'\);
• if \( \pi \) uses an unredused rule, then \( \pi \) can be transformed into \( \pi' \) such that \( \pi \succ^P \pi' \).

When the two groups of transformations are used jointly, transfers between equations and rules are made in two directions. Since certain rules disappear by inter-simplification, the definition of a fair sequence of transformations must be modified. A sequence \((\Gamma_n)_{n \geq 0}\) is fair for the two groups of transformations if no equation is persistent, if every rule whose right member is reducible is simplified at a later step, and if every persistent critical pair is transformed into an equation at a later step. It is unnecessary to handle a critical pair between equations which will disappear.

**Theorem 6.21** If the sequence \((\Gamma_n)_{n \geq 0}\) is fair and if \( M \equiv_{\Gamma_i} N \), there exists a step \( k \geq i \), such that \( M = N \) admits a proof by confluence in \( R_k \) using only reduced rules.

**Proof** By well-founded induction over \( \succ^P \). Let \( \pi :_{\Gamma_i} M = N \), containing an equational subproof or a subproof by effluence, or using an unredused rule. An equational subproof uses an equation which, by fairness, will have disappeared in \( \Gamma_{j_1} \), \( j_1 \geq i \). A subproof by effluence can be transformed into a subproof by confluence in \( \Gamma_i \), or else comes out of a critical pair of \( R_i \). If a critical pair of \( R_i \) is not persistent, it will disappear in \( \Gamma_{j_2} \), \( j_2 > i \) by inter-simplification of one of the rules which created it by superposition. Otherwise, it is transferred into the equational part of \( \Gamma_{j_3} \), \( j_3 > i \). If \( \pi \) uses a rule whose left member is reducible, the corresponding critical pair will be handled as previously. If it is the right member which is reducible, it will be simplified in \( \Gamma_{j_4} \), \( j_4 > i \).

In all cases, \( \pi \) is transformed into \( \pi' \), in a \( \Gamma_j \), with \( j \geq i \) and \( \pi \succ^P \pi' \). By the inductive hypothesis, there exists a step \( k \geq j \) such that \( M = N \) admits a proof by confluence in \( R_k \). \( \square \)

### 6.3.3 An example: the equational theory of groups

A completion procedure takes as input a set \( E_0 \) of equations and a reduction order \( \succ \), and generates a sequence \((\Gamma_n)_{n \geq 0}\) of mixed systems, starting from \( \Gamma_0 = (E_0, R_0) \). The elementary operations used are:

- **superposition**\((R_i, R_j, u)\): superpose the left member of \( R_i \) on that of \( R_j \) at occurrence \( u \) and produce the corresponding critical pair;
- **normalization**\((R_i, \ldots)\): normalize the two members of a given equation by applying the rules \( R_i, \ldots \) of the current rewriting system;
- **erasure**\((R_i)\): erase equation \( R_i \);
- **orientation**: transform a given equation into a rule by comparing its two rules by \( \succ \);
- **left-simplification**\((R_i, R_j)\): transform the rule \( R_i \) into an equation after having reduced its left member by \( R_j \).
The example of the equational theory of groups is the best known. The order of reduction used is an rpo order, the product being greater than the constant \( e \), and having a lexical status. Here is the trace of a completion procedure, starting from the three usual equations, which are oriented in the natural manner. The complete system is formed of rules 1, 2, 3, 4, 8, 10, 11, 12, 13 and 16.

1: \( ex \rightarrow x \)
2: \( x^{-1}x \rightarrow x \)
3: \((xy)z \rightarrow x(zy)\)

\begin{align*}
\text{superposition}(2, 3, 1): & \quad x^{-1}(xy) = ey \\
\text{normalization}(1): & \quad x^{-1}(xy) = y \\
\text{orientation}: & \\
\end{align*}

4: \( x^{-1}(xy) \rightarrow y \)

\text{superposition}(1, 4, 2): \quad y = e^{-1}y

5: \( e^{-1}y \rightarrow y \)

\text{superposition}(2, 4, 2): \quad e = (x^{-1})^{-1}e

6: \( (x^{-1})^{-1}e \rightarrow e \)

\text{superposition}(5, 4, 2): \quad y = (e^{-1})^{-1}y

7: \( (e^{-1})^{-1}y \rightarrow y \)

\text{superposition}(2, 7, e): \quad e^{-1} = e

8: \( e^{-1} \rightarrow e \)

\text{left-simplification}(5, 8): \quad ey = y
\text{normalization}(1): \quad y = y
\text{erasure}(5)

\text{left-simplification}(7, 8): \quad e^{-1}y = y
\text{normalization}(8, 1): \quad y = y
\text{erasure}(7)

\text{superposition}(4, 4, 2): \quad xy = (x^{-1})^{-1}y

9: \( (x^{-1})^{-1}y \rightarrow xy \)

\text{left-simplification}(6, 9): \quad xe = x
\text{erasure}(6)

\text{orientation}: \\

10: \( xe \rightarrow x \)

\text{superposition}(9, 10, e): \quad (x^{-1})^{-1} = xe
\text{normalization}(10): \quad (x^{-1})^{-1} = e
\text{orientation}: \\

11: \( (x^{-1})^{-1} \rightarrow x \)

\text{left-simplification}(9, 11): \quad xy = xy
\text{erasure}(9)
248  Equational logic

\[ \text{superposition(11, 2, 1): } e = xx^{-1} \]
orientation:

\[ \text{superposition(11, 4, 1): } y = x(x^{-1}y) \]
orientation:

\[ \text{superposition(12, 3, e): } x(yxy)^{-1} = e \]
orientation:

\[ \text{superposition(14, 4, 2): } y(xy)^{-1} = x^{-1}e \]
normalization(10):
\[ y(xy)^{-1} = x^{-1} \]
orientation:

\[ \text{left-simplification(14, 15): } xx^{-1} = e \]
normalization(12):
\[ e = e \]
erasure(14)

\[ \text{superposition(15, 4, 2): } (yx)^{-1} = x^{-1}y^{-1} \]
orientation:

\[ \text{left-simplification(15, 16): } y(y^{-1}x^{-1}) = x^{-1} \]
normalization(13):
\[ x^{-1} = x^{-1} \]
erasure(15).

Exercises

1. Let \( p, q : A \to B \) be two morphisms. Show that the set of \( x \in A \) such that \( p(x) = q(x) \) is a subalgebra \( E \) of \( A \), and that every morphism \( r : X \to A \) such that \( p \circ r = q \circ r \) factors through the inclusion \( E \hookrightarrow A \).

2. Compare the subalgebras of \( A \) and the congruences of \( A \times A \).

3. Let \( p, q : A \to B \) be two morphisms. Define a quotient \( l : B \to C \), such that every morphism \( r : B \to X \) satisfying \( r \circ p = r \circ q \) factors through \( l \).

4. Determine the class of models for each of the following axioms:
\[ x = x, \quad x = y, \quad f(x) = f(y), \quad f(x) = y. \]

5. Prove the properties of the Galois correspondences, equations (6.1)–(6.8).

6. Show that every variety is stable under isomorphism, subalgebras, quotients and products.

\footnote{Hints to the exercises labeled with a \( \star \) can be found at the end of the book.}
7. Determine whether the following classes are varieties: finite groups, infinite groups, abelian groups, rings of characteristic \( p \) and rings of characteristic dividing \( p \).

8. Derive the rules \( (R) \) and \( (T) \) from the equality axioms in NK:

\[
(R) : \quad \forall \bar{x}(M = M') \quad \frac{\forall \bar{x}(M = M')}{\forall \bar{x}(M = M'')}.
\]

\[
(T) : \quad \frac{\forall \bar{x}(M = M')}{\forall \bar{x}(M = M'')}.
\]

9. Show that the rule \( (\text{cong}) \) is derivable from \( (T) \), \( (\text{subst}) \) and the unary rules of context passing:

\[
(((f, i)_{1 \leq i \leq r}) : \quad f x_1 \ldots x_{i-1} M_i x_{i+1} \ldots x_r = f x_1 \ldots x_{i-1} N_i x_{i+1} \ldots x_r; \quad f \in \Sigma_r.
\]

10. Derive the weakening rule, equation (6.9), of the guarded equations from the \( (\text{subst}) \) rule.

11.\* Show that \((a, b; a^3 = b^2 = e, ab = ba^2)\) and \((u, v; u^2, v^2, (uv)^3)\) are two presentations of the same group. Which one?

12.\* Construct a model of \((+)\) where addition is not associative.

13.\* Show that \( T_{\Sigma/\Sigma^0} \circ T_{\Sigma^0/\Sigma^0} = T_{\Sigma/\Sigma^0}. \) What are the values of \( T_{\Sigma/\Sigma}, T_{\Sigma/\Sigma^0} \) and \( T_{\Sigma^0/\Sigma^0}[T_{\Sigma^0}]? \)

14. Show by a counterexample that the assumption ‘\( \Sigma \) non-linear’ is necessary to obtain the second point of proposition 6.7.

15.\* Show, by completion, that the equation \( x(x^{-1}y) = y \) is a consequence of \( x^{-1}(xy) = y \) (without any other axioms).

16.\* Construct by completion a canonical system for idempotent semigroups, with axioms \((xy)z = x(yz)\) and \(xx = x\).

17.\* Let \( E \) be a system of equations. Show that \( E \cup \{ M_1 \neq N_1, \ldots, M_p \neq N_p \} \) is unsolvable if and only if there exists \( i \) such that \( E \cup \{ M_i \neq N_i \} \) is unsolvable. A substitution \( \sigma \) is the solution of the inequality \( M \neq N \) if \( \sigma(M) \neq \sigma(N) \).

18.\* Show by an example how to associate a pure \( \lambda \)-term \( M \) with a simple equation between simple types with variables, so that \( M \) is typable if and only if this system of equations has a solution.
Bibliographic notes

The modules of Standard ML are described in [111]. For universal algebra, a
standard reference is Cohn's book [22]. The use of categories in computer science
is described in [5].

Unification is the theme of volume 8 (1989) of the Journal of Symbolic Com-
putation. In it can be found a general presentation by Siekmann [137]. Another
survey is [86]. Robinson's algorithm is presented in [128], those of Huet in [67], of
Paterson and Wegman in [117], of Martelli and Montanari in [100], of Corbin and
Bidoit in [26], of Martelli and Rossi in [101], and of Escalada-Imaz and Ghallab
in [41].

The proof of the Knuth-Bendix theorem comes from Huet [68]. The Knuth-
Bendix completion procedure is given by its authors in [87]. A complete proof for
it is given in [69]. The presentation given here comes from Bachmair, Dershowitz
and Hsiang's article [6].
Chapter 7

Resolution

7.1 Logic programs

Prolog was first implemented at the University of Marseille in 1973, under the direction of A. Colmerauer [23]; the first logical interpretation of the Prolog mechanism was given by R. Kowalski in 1974 [88]. Logic programming developed through the conjunction of the two events. Although logic programming is not simply Prolog, and Prolog programs are not always logical, logic programming will first be presented through 'pure' Prolog programs. A few typical examples are given here to show the role of unification.

7.1.1 A Prolog interpreter

Notation  The Edinburgh syntax will be used. Terms are written using parenthesized notation. They are formed of variables, identifiers beginning with a capital letter, and of symbols, identifiers starting with a small letter. Numbers are considered symbols of zero arity. So \( r(a, x) \) is used instead of the usual \( R(a, x) \) of logic. The queries and rules are terminated by a ".". Most of the examples are written in C-Prolog\(^1\) or NU-Prolog\(^2\).

The Marseille syntax, used by Prolog–II+,\(^3\) and occasionally used here, is very different. Examples presented will use one-character identifiers for variables and identifiers with at least two characters for signature symbols: hence \( rr(aa, x) \) corresponds to the \( r(a, x) \) of Edinburgh. Queries and rules are terminated by a ";".

\(^1\)EdCAAD, University of Edinburgh, Department of Architecture.

\(^2\)Machine Intelligence Project, Department of Computer Science, University of Melbourne, Australia.

\(^3\)Groupe d'Intelligence Artificielle de Luminy, Université d'Aix-Marseille II, distributed by PrologIA
Unification A Prolog interpreter consists mainly of a unification machine. A program does not have to be written before a query such as
\[ ?- \ f(X, g(X), Y) = f(h(Z), T, Z). \]
is made. Prolog variables are unknown in the mathematical sense, and the interpreter attempts to give them a value. An interpreter would give the following solution to the above query:
\[ \sim T = g(h(Z)), \]
\[ Z = Z, \]
\[ Y = Z, \]
\[ X = h(Z) \]
However, Prolog does not evaluate terms, unlike functional language interpreters:
\[ ?- 2 + 2 = 3 + 1. \]
\[ \sim \text{fail} \]
This non-evaluation and the answer fail are typical of Prolog, where failure does not necessarily mean an error.

Data structures The data manipulated by Prolog are first-order terms.
Types can be defined, along with their constructors, but the concept of type is not explicit in Prolog: it is simply a way of reading certain terms. For example, the unary integers can be defined using the constants nat, 0 and s, and the definite program
\[
\text{nat}(0).
\]
\[
\text{nat}(s(X)) :- \text{nat}(X).
\]
which states that 0 is an integer and that \( s(X) \) is an integer if \( X \) is an integer. Note that these constants were not declared, and that the interpreter infers their arity, just as the ML interpreter infers the type of expressions. The set of the two rules constitutes the definition of predicate nat/1. The /1 indicates that nat is of arity 1. Several predicates can use the same symbol, so long as they are distinguished by their arity. There is therefore no arity checking akin to ML's type checking.

A set of predicate definitions constitutes a logic program. Once supplied with this program, the interpreter can respond to queries:
\[ ?- \text{nat}(s(0)). \]
\[ \sim \text{true} \]
Should there be several solutions, they are given successively in response to a ';' RETURN typed by the user. The interpreter stops looking in response to a simple 'RETURN'. It enumerates the integers in response to the query:
\[ ?- \text{nat}(X). \]
\[ \sim X = 0 \ ; \text{RETURN} \]
\[ X = s(0) \quad ; \text{RETURN} \]
\[ X = s(s(0)) \quad \text{RETURN} \]

Other recursive data structures, more useful than the integers, such as lists and trees, can be defined similarly.

**Relations** Here is a program, using the same integers, defining addition:
\[
p(X, 0, X).
p(X, s(Y), s(Z)) \quad :- \quad p(X, Y, Z).
\]

Even if the values of a *function* are to be computed, a *relation* must be programmed, with an argument for the result. So ordinary computations such as
\[
?- \quad p(s(0), s(0), Z).
\]

\[ Z = s(s(0)) \]

can be done, but also queries asking for the arguments of a relation such as:
\[
?- \quad p(X, Y, s(s(0))).
\]

\[ X = s(s(0)), \]
\[ Y = 0 \quad ; \]
\[ X = s(0), \]
\[ Y = s(0) \quad ; \]
\[ X = 0, \]
\[ Y = s(s(0)) \quad ; \]
\textbf{fail}

The final fail means that there are no more solutions.

A request can be composed of several literals. The search for \( X \) and \( Y \), such that \( X + Y = 4 \) and \( X + 2 = Y \), can be written by the query:
\[
?- \quad p(X, Y, s(s(s(s(0))))), \quad p(X, s(s(0)), Y).
\]

\[ Y = s(s(s(0)))) , \]
\[ X = s(0) \]

Even though it is not visible, a standard Prolog interpreter starts by computing a solution for the first literal, then checks if it is a solution to the second literal. Here, it will first find \( X = 4, Y = 0 \), which does not satisfy \( p(X, s(s(0)), Y) \). It will then return to the first literal to examine the other solutions. Thus \( X = 3, Y = 1 \), then \( X = 2, Y = 2 \) are obtained, but they are still not solutions to the second literal. Finally, \( X = 1, Y = 3 \) is a solution to both literals, the result is obtained, and the interpreter continues to search. The declarative reading of the query, where a system of equations is to be solved, and the operational reading just outlined are very different. This search mechanism is the strength of Prolog in situations where an exhaustive search of graphs, state spaces and similar structures is necessary, since it need not be programmed explicitly. This strength is also a weakness of Prolog, since it is too easy to generate inefficient programs.
Lists  As in Lisp and ML, lists form a data structure with a predefined syntax. Table 7.1 indicates the correspondence between the notation in the three languages.

<table>
<thead>
<tr>
<th>Prolog</th>
<th>Standard ML</th>
<th>Lisp</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a,b,c]</td>
<td>[a,b,c]</td>
<td>(a b c)</td>
</tr>
<tr>
<td>[a</td>
<td>L]</td>
<td>a::L</td>
</tr>
<tr>
<td>[a,b,c</td>
<td>L]</td>
<td>a::b::c::L</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
<td>()</td>
</tr>
</tbody>
</table>

Table 7.1 Lists in different languages

Numerous programs manipulating lists come from the Lisp tradition, such as the concatenation of two lists, the reversal of a list and quicksort:

```prolog
append([], Y, Y).
append([H|T], Y, [H|TY]) :- append(T, Y, TY).
reverse([], []).
reverse([H|T], Y) :- reverse(T, RT), append(RT, [H], Y).
```

The predicate quicksort/2 sorts a list of numbers by decomposing it into two lists of numbers smaller than or equal and larger than the first element (the 'pivot'), respectively, using the predicate partition/4, and calls itself recursively for each of the sublists.

```prolog
quicksort([], []).
quicksort([H|T], Y) :-
    partition(T, H, Littles, Bigs),
quicksort(Littles, S_Littles),
quicksort(Bigs, S_Bigs),
    append(S_Littles, [H|S_Bigs], Y).
```

```prolog
partition([], _, [], []).
p
partition([H|T], P, [H|Littles], Bigs) :-
    H =< P,
    partition(T, P, Littles, Bigs).
p
partition([H|T], P, Littles, [H|Bigs]) :-
    H > P,
    partition(T, P, Littles, Bigs).
```

The set, more exactly the sorted list, of answers to a query is obtained using the predefined predicate solutions/3 of NU-Prolog:

```prolog
solutions([X|Y], p(X, Y, s(s(0))), L).
```

```prolog
L = [[] | s(s(0))], [s(0) | s(0)], [s(s(0)) | 0]]
```
The logic variable and unification. The data structures of functional programming contain 'values'. Here, they can contain constants and 'variables'. A closed term is completely known. If it contains variables, it is a partially specified object. The interpreter will make it 'rise' in the lattice of terms by assigning values to the variables so that it satisfies the query in which it occurs, and will make it rise the least amount possible so that it remains the most general possible.

Terms can be used, as for the concrete types of ML, to define structured values. A term `rectangle(width(2), length(3))`, or simply `rectangle(2,3)`, represents a particular rectangle, and `rectangle(X,Y)` an arbitrary rectangle. But no ordinary language can be used to specify squares as constrained rectangles. In Prolog, `rectangle(X,X)` is an arbitrary square.

The essential mechanism is unification, which, by itself, does a number of basic functions which are handled by distinct mechanisms in traditional languages:

- parameter passing, both input and output; unification is more powerful than call by pattern matching, since it eliminates the difference between input and output parameters;
- data destructuring (access to the fields in records), but without mutable objects, the modification of fields is not efficient;
- data construction, in a manner similar to C's `malloc`;
- variable assignment, similar to the 'single-assignment' variables of functional programming languages;
- testing the value of variables, as for call by pattern matching.

D-lists. D-lists (from difference lists) are an original data structure, typical of logic programming, first used by Colmerauer, even before Prolog came to be, in his research on syntactic analysis, and can be used to represent term contexts. A context is a term with a 'hole', written '?', ready to receive a graft of a term. It is tempting to represent such a hole by a variable. Doing that would have no meaning in functional programming, but it is possible in logic programming, even though it is inefficient, since the graft needs to go through the entire term to reach the variable. In sequential programming, the problem would be handled by holding a pointer to the hole, as in the standard implementation of files. The logic variable allows such an access: all the occurrences of the same variable denote the same object. Hence, to reach the end of the list `[a,b,c,d]` quickly, it suffices to have a variable `X` having an occurrence at the end of the list, and another at the `surface'`, which is immediately accessible. This can be done by using, for example, the infix constant `-` to form the term: `[a,b,c,d|-X]`-`X` represents the context `[a,b,c,d]?`.

A pair of lists `L, L'` is a D-list if `L = [L_1, ... , L_p| M]` and `L'` is unifiable with `M`. In that case, if `θ` is the most general unifier, then the pair `L, L'` represents the list context `[θL_1, ... θL_p]?`.

Example. The D-lists `[a,b,c,d]^[c,d], [a,b,c]^-[c], [a,b,c|T]^[X|T]`, and `[X,b,a|T]^-[X|T]` all represent the context `[a,b]?`. The simplest representations
are \([a,b]-[c,d]\), of no interest, and \([a,b|x]-x\), the 'standard' representation. But
\([a,b]-[c,d]\) is not a D-list, since \([c,d]\) does not unify with any suffix of \([a,b]\).

With this data structure, concatenation is implemented using a single unification,-
by the program:

```prolog
append_d1(L-M,M-N,L-N).
```

To concatenate the lists \([a,b]\) and \([c,d]\), query

```prolog
?- append_d1([a,b|x]-x,[c,d|y]-y,Z).
```

is solved by unifying \(M\) with \([c,d|y]\), \(N\) with \(Y\), \(L\) with \([a,b,c,d|y]\) and \(Z\) with
\([a,b,c,d|y]-y\). In fact, concatenation is often done implicitly, without even calling
\(append_d1\).

**Occur check** Most Prolog implementations do not implement unification in \(T[X]\)
completely, for unconvincing efficiency reasons. Only Prolog–II+ implements real
unification, but in \(T^r[X]\), and not in \(T[X]\), and so one can program with rational
terms, i.e., cyclical objects. In the simplest case

```prolog
?- X = f(X).
```

\(~X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(~C
```

C–Prolog’s answer is incorrect in the term algebra, but correct if it is interpreted
as a rational term. Prolog–II responds better:

```prolog
eq(x, ff(x));
\(~x=ff(*1)
```

In other cases, such as

```prolog
?- f(X,g(X),X) = f(g(Y),Y,Y).
```
a standard interpreter would give no response, since it loops! But in Prolog–II:

```prolog
> eq(ff(x,gg(x),x), ff(gg(y),y,y));
\(~x=gg(gg(*2)), y=gg(gg(*2))
```

If the values of variables are not asked for, there is no way of knowing if a bad
unification took place:

```prolog
test :- p(X,X).
p(X,f(X)).
?- test.
\(~true
```

Prolog gives an answer which is logically incorrect. This lack of occur check takes
away substantially from the logical credibility of Prolog, even if in practice it occurs
rarely.
D-lists are, however, a major exception, with seemingly useful examples giving incorrect answers. The following query, which asks whether the list \([a]\) is the result of concatenating two empty lists, succeeds!

\[-\text{append}_d\text{L}-L,M-M,[a\mid N]-N\].

The program

\[
\text{rep}(L-E,X) \leftarrow \text{append}(X,E,L).
\]

computes the list represented by a D-list. However, the following query loops with an erroneous answer, even though the solution \(X = [a,b]\) is expected:

\[-\text{rep}([a,b\mid E]-E,X).\]

7.1.2 Views of logic programming

Logic programming is attractive because of the many ways in which it can be viewed: logical, procedural, rewriting, equation resolution, theorem-proving, networks of communicating processes, etc.

*Logical formulas*

Kowalski’s view [88] initiated logic programming. A *literal* is an atom (positive literal) or the negation of an atom (negative literal). A formula is *clausal* if it is of the form \(\forall \overrightarrow{\varepsilon} \varphi^o\), where \(\varphi^o\) is a disjunction of literals. The universal closure of a disjunction of literals, of which at most one is an atom, is called a *Horn clause*. It is standard to distinguish the following three forms,

\[
\begin{align*}
\forall \overrightarrow{\varepsilon} A, \\
\forall \overrightarrow{\varepsilon} (A \lor \neg A_1 \lor \cdots \lor \neg A_p), \\
\forall \overrightarrow{\varepsilon} (\neg L_1 \lor \cdots \lor \neg L_q),
\end{align*}
\]

where the \(A, A_i, L_i\) are atoms, classically equivalent to:

\[
\begin{align*}
\forall \overrightarrow{\varepsilon} A & \quad \text{unit clause, or fact,} \\
\forall \overrightarrow{\varepsilon} (A_1 \land \cdots \land A_p \Rightarrow A) & \quad \text{definite clause, or rule,} \\
\neg \exists \overrightarrow{\varepsilon} (L_1 \land \cdots \land L_q) & \quad \text{negative clause, or goal.}
\end{align*}
\]

Note that a unit clause is a particular case of a definite clause, when \(p = 0\). The Prolog notations for fact and rule are, respectively:

\[
A.
\]

\[
A :- A_1, \ldots , A_p.
\]

If \(C\) is the definite clause \(A :- A_1, \ldots , A_p\), write \(C^+\) for the atom \(A\), and \(C^-\) for the set of literals \(A_1, \ldots , A_p\). \(C^+\) is the head of \(C\), and \(C^-\) is its body. A definite program is a sequence of definite clauses. Its logical meaning is the conjunction of the clauses which make it up. Note that this logical meaning is not affected
by the order or repetition of literals and clauses, so that bodies and programs can be considered as sets rather than sequences. The implicit universal quantification makes every variable local to the clause in which it appears: there are no ‘global variables’ in Prolog.

The Prolog interpreter responds to queries of the form

\[ \neg L_1, \ldots, L_q, \]

where the logical meaning is that of formula

\[ \exists \bar{x}(L_1 \land \ldots \land L_q), \]

which is the negation of a goal.

It is useful to extend the notation used for terms, occurrences and subterms to queries. If \( R \) is the query \( \neg L_1, \ldots, L_q \), write \( R / i \) for literal \( L_i \), \( R(i) \) for the head symbol of \( L_i \), and \( R \setminus i \) for the query \( \neg L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_q \), the result of erasing \( L_i \). For \( q = 0 \), an empty query is obtained, written \( \top \), which is equated with the connective \( \top \) (true). To avoid confusion, when \( R \) is the query \( \neg L_1, \ldots, L_q \), \( R \) is also used for the formula \( L_1 \land \ldots \land L_q \), and \( \forall(\neg R) \), or \( R^\top \), for the goal \( \forall \bar{x}(\neg L_1 \lor \ldots \lor \neg L_q) \).

A Horn formula is a conjunction of Horn clauses. The conjunction of program rules and of a goal is therefore a Horn formula. The importance of the class of Horn clauses will be explained later.

Submitting a query \( R \) to a program \( P \), consists, from a logical point of view, of proving the existential formula \( \exists \bar{x}R \) from \( P \), i.e., of refuting \( P \land R^\top \). A solution to this query is a substitution \( \sigma \) such that \( P \not\models \sigma(R) \). The interpreter proves \( \exists \bar{x}R \) in a constructive manner by generating solutions.

This interpretation of programs and queries is only declarative, and gives no interpretation of the execution of programs as proof procedures. Natural deduction could be used instead of methods particular to clauses, such as the resolution principle and its refinements. Kowalski introduced SLD-derivation as a particular case of the resolution principle, thereby giving an operational semantics to logic programming. This subject will be treated in depth.

**Intuitionistic sequents**

The definite clause \( A :- A_1, \ldots, A_p \) can be seen not only as a formula \( \forall \bar{x}(A_1 \land \ldots \land A_p \Rightarrow A) \), but also as a sequent \( A_1, \ldots, A_p : A \) composed of atomic formulas, which has the same declarative meaning. On the other hand, a query \( \neg L_1, \ldots, L_q \) corresponds to a sequent \( L_1, \ldots, L_q : \). In both cases they are sequents of the intuitionistic calculus (at most one formula in the right member of the sequent) restricted to atomic formulas. We therefore have a good idea of what logic to use. Recall that the rules of the sequent calculus are divided into the structural rules (rearranging of formulas), the logical rules (for connectives and quantifiers), the cut rule and the identity axiom. The logical rules were proposed by Gentzen, as were the rules of NK, to make the meaning of the logical symbols explicit. But
it is possible to do logic without using any logical symbols! Formulas without
logical symbols are atoms. Natural deduction can do practically nothing as it
only manipulates atoms. However, the sequent calculus can work with sequents
formed from atoms, and there are (structural, identity and cut) rules using no logic
symbols, allowing a very combinatoric reasoning with axioms.

With only the identity axiom, the Hauptsatz shows that the cut can be elimi-
nated, but then only structural properties can be proven (for example, to derive
A, B : A). If other axioms Γ : Δ formed of axioms are allowed, the cuts can no
longer be eliminated; they are essential. This is what happens in logic program-
ming, where axioms are definite clauses. A proof in LJ, from the axioms and all of
their instances, may use no other rule than the cut rule, one of the premises being
the instance of an axiom.

\[
\text{(cut): } \frac{A_1, \ldots, A_p : A}{A_1, \ldots, A_p, B_2, \ldots, B_q : A}
\]

Therein lies the principle of SLD-resolution.

Rewrite rules

This is Colmerauer's view, which inspired the Prologs of Marseille. The rules
are written \( A \rightarrow A_1, \ldots, A_p \); and read from left to right, as in \('A\) rewrites to
\( A_1, \ldots, A_p\)', instead of \('A_1, \ldots, A_p\) imply \('A\'). A query is a sequence of terms—the
difference between function and relation symbols is eliminated—to be erased using
the rules of the program. There is a striking analogy with context-free grammars
where non-terminals must be eliminated by rewriting. A rule can be applied if a
system of equations, which grows as the literals are erased, can be solved. There is
no reason to limit oneself to equations between terms. Prolog–II+ treats equations
and inequalities \((\neq)\) over rational terms. Other languages work with other domains
(Booleans, finite sets, rational, algebraic or real numbers): logic programming then
becomes constraint programming. In these languages, such as Prolog–III, CHIP,
or the family of CLP (Constrained Logic Programming), the interpreter is capable
of solving systems of equations and inequalities \((\neq)\) between rational terms or
Booleans, and inequalities \((\leq)\) between integers or reals.

\[
\begin{align*}
    f(x, g(a)) &= f(g(x), y) \\
    x &\neq y, \\
    2n_1 + 3n_2 &> 14 \\
    n_1 - n_2 &= 7, \\
    b_1 \lor b_2 \land b_3 &= 1 \\
    b_1 \lor b_2 &\neq b_3,
\end{align*}
\]

\(x, y\) rational terms
\(n_1, n_1\) integers
\(b_1, b_2, b_3\) Boolean.

7.2 Semantics of definite programs

There is now a well-established logical basis, from the works of Kowalski, Apt, van
Emden, Clark, Lassez and Maher, which is synthesized in Lloyd’s book [95]. The
logical approach can be split into two parts. The declarative semantics determines what can be computed. The operational semantics describes what can be executed.

### 7.2.1 Declarative semantics of definite programs

It is doubtful whether one can consider the predicate calculus as a programming language: a formula computes nothing. However, it is possible to isolate fragments, containing a subset of the predicate calculus, a semantics and appropriate proof methods, in which one can compute effectively. Even though several fragments have been considered, only the 'Horn' fragment has been developed into a real programming language. Two essential ideas are implemented: given the problem $\varphi \vdash \psi$, program $\varphi$ specifies a search for a proof of $\psi$, and from this proof a result is extracted. The presentation will begin by justifying Horn clauses.

**Horn clauses and product algebras**

Just as in equational theories, Horn clauses satisfy an important stability property, which explains their use in programming: stability under product.

First, the definition of the product $\mathcal{M} = \prod_{i \in I} \mathcal{M}_i$ of a family of $\Sigma$-algebras $\mathcal{M}_i$, already given for the functional case, is extended to predicate calculus signatures by defining $R^\mathcal{M} = \prod_i R^{\mathcal{M}_i}$, for every relational symbol $R$: for an atom $A$ and a valuation $\bar{\xi} : X \to \prod_i \mathcal{M}_i$ of components $\xi_i : X \to \mathcal{M}_i$, then $\bar{\xi}(A) = 1$ if $\xi_i(A) = 1$ for every $i$.

**Proposition 7.1** The class of models of a set of Horn clauses is stable under product.

**Proof** Let $(\mathcal{M}_i)_i$ be a family of models of a Horn clause $C \equiv A \lor \neg A_1 \lor \ldots \lor \neg A_p$ and $\bar{\xi} = (\xi_i)_i : X \to \prod_i \mathcal{M}_i$ a valuation. It must be shown that $C^\mathcal{M}(\bar{\xi}) = 1$, i.e., $A^\mathcal{M}(\bar{\xi}) = 1, A_1^\mathcal{M}(\bar{\xi}) = 0, \ldots, A_p^\mathcal{M}(\bar{\xi}) = 0$. Suppose that $A^\mathcal{M}(\bar{\xi}) \neq 1$. This implies that there exists $i$ such that $A_i^\mathcal{M}(\xi_i) = 0$. Since $\mathcal{M}_i \models C, A_i^{\mathcal{M}_i}(\xi_i) = 1, A_1^{\mathcal{M}_i}(\xi_i) = 0, \ldots, A_p^{\mathcal{M}_i}(\xi_i) = 0$. There therefore exists $k_i$ such that $A_{k_i}^{\mathcal{M}_i}(\xi_i) = 0$, which implies $A_{k_i}^\mathcal{M}(\bar{\xi}) = 0$, hence $C^\mathcal{M}(\bar{\xi}) = 1$.

If $\Gamma$ is a set of Horn clauses and $\mathcal{M}_i \models \Gamma$ for every $i$, then $\mathcal{M}_i \models C$ for every $i$ and every $C \in \Gamma$. Hence, by the above, $\prod_i \mathcal{M}_i \models C$ for every $C$, i.e., $\prod_i \mathcal{M}_i \models \Gamma$. □

An arbitrary clause does not satisfy this property, as the following property proves. Let $C \equiv p(x) \lor q(x)$, and the algebras $\mathcal{M}_1, \mathcal{M}_2$ be of domain $\{a, b\}$, $p^{\mathcal{M}_1}(a) = q^{\mathcal{M}_1}(b) = 1$, $p^{\mathcal{M}_2}(b) = q^{\mathcal{M}_2}(a) = 0$, and the contrary for $\mathcal{M}_2$. Then $\mathcal{M}_1, \mathcal{M}_2 \models C$, but $\mathcal{M}_1 \times \mathcal{M}_2 \not\models C$. The converse of proposition 7.1 is true: it is the McKinsey theorem.

**Extraction of a result**

One of the underlying ideas of logic programming is to consider a computation as the extraction of a result from a proof. For example, from a derivation $\exists x \psi$, 

a term \( t \) would be extracted such that \( \psi[x := t] \) were derivable. The Herbrand theorem leans in this direction.

Consider two propositions, one universal \( \varphi = \forall \bar{x} \varphi^o \), and the other existential \( \psi = \exists \bar{x} \psi^o \), written in prenex form for simplicity. The formula \( \varphi \Rightarrow \psi \) being existential, the existential version of the Herbrand theorem states the equivalence between \( \vdash \varphi \Rightarrow \psi \) and the existence of a domain \( D \) such that \( \vdash \operatorname{Exp}(\varphi \Rightarrow \psi, D) \).

Recall that the formula \( \operatorname{Exp}(\varphi, D) \) is the expansion of \( \varphi \) over \( D \), a finite set of closed terms. By the definition of the expansion,

\[
\operatorname{Exp}(\varphi \Rightarrow \psi, D) = \operatorname{Exp}(\varphi, D) \Rightarrow \operatorname{Exp}(\psi, D).
\]

Since \( \varphi \) is universal (lemma 5.12), \( \vdash \varphi \Rightarrow \operatorname{Exp}(\varphi, D) \), hence \( \vdash \varphi \Rightarrow \operatorname{Exp}(\psi, D) \). By making the expansion explicit, closed substitutions \( \theta_i \) are obtained such that

\[
\varphi \vdash \theta_1 \psi^o \lor \ldots \lor \theta_n \psi^o.
\]

According to the strong version of the Herbrand theorem—not proven here—these substitutions can be extracted effectively from a derivation of \( \psi \)—if one can be constructed, which the theorem does not indicate. Furthermore, the result obtained is a disjunction, and one cannot necessarily conclude that \( \varphi \vdash \theta_i \psi^o \) for some \( i \), which would really allow one to refer to the result of a computation. Otherwise, there is an indeterminate result.

**Example** Let \( \varphi = p(a) \lor p(b) \), and \( \psi = \exists x \operatorname{Exp}(x) \). Then \( \vdash \varphi \Rightarrow \psi \), but there is no term \( t \) such that \( \vdash \varphi \Rightarrow p(t) \).

This situation is not satisfactory, although it does lead the calculus to nondeterminism in an interesting manner. If determinate results are wanted, then either the logic must be changed to a more constructive logic or the class of formulas must be restricted. Standard logic programming plays both games: its proofs use intuitionistic logic and its formulas are restricted to Horn clauses. More precisely, with respect to the above situation, logic programming handles problems \( \varphi \vdash \psi \) where \( \varphi^o \) is a conjunction of definite clauses, and \( \psi \) is the existential closure of a conjunction of atoms, i.e., the negation of a negative clause.

**Theorem 7.2 ([48])** Let \( \varphi \) be a universal Horn proposition, and let there be \( n \) existential propositions

\[
\psi_i = \exists \bar{x} \psi_i^o, \quad \psi_i^o = (B_i^1 \land \ldots \land B_i^{p_i}), \quad i = 1, \ldots, n,
\]

where the \( B_i^k \) are atoms:

1. \( \varphi \vdash \psi_1 \lor \ldots \lor \psi_n \) if and only if there exist \( i \) such that \( \varphi \vdash \psi_i \);
2. \( \varphi \vdash \psi_i \) if and only if there exists a closed term \( t \) such that \( \varphi \vdash \psi_i^o[x := t] \).

**Proof** (1) By contradiction: Suppose \( \vdash \varphi \Rightarrow (\psi_1 \lor \ldots \lor \psi_n) \) and \( \varphi \not\vdash \psi_i \) for every \( i \). There therefore exist algebras \( M_i \) such that \( M_i \vdash \varphi \) and \( M_i \not\vdash \neg \psi_i \). Since \( \neg \psi_i \) is equivalent to \( \forall \bar{x} (\neg B_i^1 \lor \ldots \lor \neg B_i^{p_i}) \), \( M_i \not\vdash \neg \psi_i \) means that for every \( \xi_i : X \rightarrow M_i \) there is an index \( k_i \) such that \( (\neg B_i^{k_i})^{M_i}(\xi_i) = 1 \), i.e., \( (B_i^{k_i})^{M_i}(\xi_i) = 0 \).
Let $\mathcal{M} = \prod_{1 \leq i \leq n} \mathcal{M}_i$. Then $(B^\mathcal{M}_i)(\xi) = 0$, by the definition of the product algebra, hence $(\neg B^\mathcal{M}_i)(\xi) = 1$, for every $\xi : X \to M$. This proves that $\mathcal{M} \vDash \neg \psi_i$ for every $i$. Furthermore, by using the property of stability under product of Horn clauses, $\mathcal{M} \vDash \varphi$. It follows that $\mathcal{M} \vDash \varphi \land \neg \psi_1 \land \ldots \land \neg \psi_n$, which contradicts $\mathcal{M} \vDash \varphi \Rightarrow (\psi_1 \lor \ldots \lor \psi_n)$.

(2) According to the Herbrand theorem, if $\varphi \vDash \psi_i$, there exist closed substitutions $\theta_j$ such that $\varphi \vDash \theta_j \psi^\varphi \lor \ldots \lor \theta_n \psi^\varphi$. This implies, by (1), that $\varphi \vDash \theta_j \psi^\varphi$ for a $j$. 

This theorem allows one to give a declarative semantics to existential queries. What are computable are the substitutions. A closed solution of a query $\exists \vec{x} \psi$ is a closed substitution $\sigma$ such that $P \vdash \sigma(\psi)$.

As in similar situations, such as unification and typing in $\lambda$-calculus, rather than enumerating the set of solutions, the most general of the possible solutions is sought. In natural deduction, all the derivations considered have their assumptions among the propositions forming the program $P$. If there is a derivation of $p(t)$ for a closed term $t$, one can infer $\exists x p(x)$ by losing the information contained in $t$, which is certainly not in the spirit of logic programming. In the case of a non-closed term $t$, for example $f(a, z)$, one can always infer $\exists x p(x)$ from $p(f(a, z))$, but, better still, one can infer $\forall x p(f(a, z))$ by $\forall$-introduction, variable $z$ not being free in any assumption since all the assumptions are propositions. The semantics of a query must therefore be changed by searching for $\sigma$ with values in $T[x]$ such that $P \vdash \forall(\sigma \psi)$. This appears surprising, since the result is a universal formula in the form of a query, while to apply the Herbrand theorem, what is needed is 'universal $\vdash$ existential'. However, the $\forall$ only plays a superficial role, stating that solutions are unique up to renaming of variables.

**Definition 7.1** The solutions to query $R$ for program $P$ are the substitutions $\sigma : \text{var}(R) \to T[x]$ such that $P \vdash \forall(\sigma R)$. Their set is written $[R]_P$.

**Examples** Let $P \equiv p(a) \land \forall x(p(x) \Rightarrow (f(x)))$, the program

\[
\begin{align*}
p(a) . \\
p(f(x)) & : = p(x) .
\end{align*}
\]

Then $[\exists x p(x)]_P = \{ [f^n(x)] ; n \geq 0 \}$.

For a closed term $A$, either $[A]_P = \emptyset$, if $P \not\vdash A$, or $[A]_P = \{ 1 \}$, if $P \vdash A$, where $1$ is the unit substitution.

**Herbrand models**

It would be tempting to associate with a set of Horn clauses the product of 'all' its models, but it is a gigantic algebra! However, for universal formulas, the satisfiability problem can be restricted to Herbrand algebras. They are algebras whose restriction to the functional part of the signature is the algebra of closed terms.

A Herbrand algebra $\mathcal{H}$ is defined by its *fact base*, a subset $B(\mathcal{H})$ of the set $At_\Sigma$ of closed terms (called Herbrand base): $R^\mathcal{H}(t_1, \ldots, t_n) = 1$ if $Rt_1 \ldots t_n \in B(\mathcal{H})$. 
Furthermore, if $\mathcal{H}$ and $\mathcal{H}'$ are two Herbrand algebras, there exists an algebra morphism from $\mathcal{H}$ to $\mathcal{H}'$ if and only if $B(\mathcal{H}) \subseteq B(\mathcal{H}')$. This morphism is then unique, i.e., the identity, which makes $\mathcal{H}$ a subalgebra of $\mathcal{H}'$ (see exercise 4).

A product of a family of Herbrand algebras $(\mathcal{H}_i)_{i \in I}$ is not a Herbrand algebra since its domain is $(T_\Sigma)^I$ and not $T_\Sigma$. But the diagonal $D_I$ of this product has an algebraic structure, isomorphic to the Herbrand algebra defined by the intersection of the fact bases $B(\mathcal{H}_i)$:

$$B(D_I) = \bigcap_{i \in I} B(\mathcal{H}_i).$$

Since only Herbrand algebras are being considered, the correct notion of product is not the ordinary product of algebras, but, rather, the intersection of the bases of facts, which satisfies the same algebraic properties as a product. Because of all these good properties, Herbrand algebras are equated with subsets of $\text{At}_\Sigma$, as are products of Herbrand algebras with intersection in $\mathcal{P}(\text{At}_\Sigma)$.

Every definite program has a Herbrand model: $\text{At}_\Sigma$ itself satisfies all the definite clauses in an obvious way. Starting from a family of Herbrand models of a set $\Gamma$ of Horn clauses, it is known (proposition 7.1) that a product of models is a model. The diagonal of the product is also a model, since a subalgebra of a model of a universal theory is a model of it. From an algebraic point of view, the category of Herbrand models of a set of Horn clauses has products. In particular, it has a terminal object, which is $\text{At}_\Sigma$. It also has an initial object, which is defined by the intersection of the Herbrand models of $\Gamma$.

Let $P$ be a definite program, i.e., a finite set of definite clauses. Write $\mathcal{M}_P$ for the initial Herbrand model, the intersection of all the Herbrand models of $P$.

**Proposition 7.3** $\mathcal{M}_P = \{ A \in \text{At}_\Sigma; P \models A \}$.

**Proof** $P$ and $A$ being universal, $P \models A$ if and only if $P \Rightarrow A$ is satisfied by all the Herbrand algebras, i.e., $A$ is true in all the Herbrand models of $P$, which means that $A$ belongs to the intersection of these models, which is $P$. 

---

**The $T_P$ operator**

R. Kowalski and M. van Emden associate with a definite program $P$ a monotone operator $T_P$ over $\mathcal{P}(\text{At}_\Sigma)$. Each clause $A : = A_1, \ldots , A_p$ of $P$ generates the inference rules over $\text{At}_\Sigma$,

$$\frac{\theta A_1 \ldots \theta A_p}{\theta A},$$

for all closed substitutions $\theta$. To this inference system corresponds a monotone operator $T_P : \mathcal{P}(\text{At}_\Sigma) \rightarrow \mathcal{P}(\text{At}_\Sigma)$,

$$T_P(\mathcal{H}) = \{ \theta C^+; \theta : X \rightarrow T_\Sigma \text{ and } \mathcal{H} \models \theta C^- \}.$$

Since the rules only have a finite number of premises, the operator $T_P$ is finitary, hence continuous, and its least fixpoint is therefore

$$\mu T_P = T_P^\omega = \bigcup_{n \geq 0} T_P^n(\emptyset).$$
Lemma 7.4 A Herbrand algebra $\mathcal{H}$ is a model of $P$ if and only if $T_P(\mathcal{H}) \subseteq \mathcal{H}$.

Proof Let $C$ be the clause $A : - A_1, \ldots, A_p$. $\mathcal{H} \models C$ if for every valuation $\xi$ taking values in $T_\Sigma$, $A_1^\mathcal{H}(\xi) = 1, \ldots, A_p^\mathcal{H}(\xi) = 1$ imply $A^\mathcal{H}(\xi) = 1$. But, $A_i^\mathcal{H}(\xi) = 1$ means $\xi(A_i) \in \mathcal{H}$. Hence $\mathcal{H} \models C$ if and only if $T_P(\mathcal{H}) \subseteq \mathcal{H}$.

Proposition 7.5 $M_P = \text{Ind}(T_P) = T_P^{\omega}$.

Proof Since $T_P$ is continuous, $\text{Ind}(T_P) = T_P^{\omega}$. By definition,

$$\text{Ind}(T_P) = \bigcap\{\mathcal{H}; T_P(\mathcal{H}) \subseteq \mathcal{H}\}.$$  

But, $T_P(\mathcal{H}) \subseteq \mathcal{H}$ means that $\mathcal{H}$ is a model of $P$ by lemma 7.4, and $M_P$ is the intersection of the models of $P$.

The operator $T_P$ therefore allows the enumeration of the consequences of $P$ by ‘forward chaining’. It is an effective method of computation, but not efficient, since the essential aspect is to allow proofs by induction over $P$. Being monotone, $T_P$ also has a greatest fixpoint $\mu T_P$, obtained by transfinite induction of $T_P$ starting from above (i.e., $\text{At}_\Sigma$): $\mu T_P = T_P^{\omega}$ for a countable ordinal $\alpha$, which can be strictly greater than $\omega$.

Searching for a proof

The standard interpretation of logic programming is in conflict with the Curry–Howard correspondence, according to which a program corresponds to a proof and its specification corresponds to a formula. Here, a ‘program’ is a formula! But this nomenclature is in conflict with ordinary programming, since these programs are not executable. A query is needed to produce a result, although a program could be run by forward chaining, without a query, to obtain the set of its logical consequences. Rather, a logic program should be seen as a specification of the search for a proof, and it is normal that this specification be done with a formula. It is queries which play the role of programs, viewed as terms to be reduced. In fact, for a Curry–Howard correspondence of logic programs to exist would probably require unifying logic programming and functional programming, and, for this to be possible, would require a radical change of viewpoint.

Miller, Nadathur and Scedrov [109] have proposed a characterization of the deduction relations, adapted to logic programming.

A program specifies the search for a proof (not how the search must be done, but what the properties characterizing the search for a proof are) and must allow the extraction of a result from the proof:

\begin{align*}
(\land) & \quad P \vdash \varphi \land \psi \text{ if and only if } P \vdash \varphi \text{ and } P \vdash \psi; \\
(\lor) & \quad P \vdash \varphi \lor \psi \text{ if and only if } P \vdash \varphi \text{ or } P \vdash \psi;
\end{align*}
\((\Rightarrow)\) \(P \vdash \varphi \Rightarrow \psi\) if and only if \(P, \varphi \vdash \psi\);

\((\exists)\) \(P \vdash \exists x \varphi\) if and only if there exists a term \(t\) such that \(P \vdash \varphi[x := t]\);

\((\forall)\) \(P \vdash \forall x \varphi\) if and only if \(P \vdash \varphi[x := y]\) for every bound variable \(y\) in \(P\) or \(\forall x \varphi\).

These conditions are satisfied by Prolog, but only vacuously in the case of \((\forall)\), \((\Rightarrow)\) and \((\forall)\) since queries do not use these connectives. Any fragment of first-order logic satisfying these requirements might be a candidate for extending logic programming.

### 7.2.2 Operational semantics of definite programs

The operational semantics of definite programs is given by SLD-resolution: it is the logical meaning given to them by Kowalski, as a refinement of the resolution principle, introduced by Robinson in 1965 [128], and since studied extensively as a method for automatic theorem-proving. SLD means Selection, Linear, Definite. Discovered and implemented independently by Colmerauer in the first Prolog, SLD-resolution has other non-logical readings, being similar to a form of rewriting controlled by equation resolution.

The resolution rule

In clausal logic, and in the restricted framework of logic programming, the resolution rule is (the formulas are implicitly universally quantified):

\[
\begin{align*}
\neg L_1 \lor \ldots \lor \neg L_q & \quad A \Leftarrow A_1 \lor \ldots \lor A_p \\
\neg \theta L_1 \lor \ldots \lor \neg \theta L_{s-1} \lor \neg \theta A_1 \lor \ldots \lor \neg \theta A_p \lor \neg \theta L_{s+1} \lor \ldots \lor \neg \theta L_q & \\
\hline
\neg \theta L_1 \lor \ldots \lor \neg \theta L_q
\end{align*}
\]

where the \(L_i\), \(A\) and \(A_j\) are atoms and \(\theta\) is a most general unifier of \(L_s\) and \(A\). Using the language of definite programs, if \(R\) is the query \(\neg L_1, \ldots, L_q\) and \(C\) is the clause \(A \Leftarrow A_1, \ldots, A_p\), the application of this rule consists of globally instantiating the query \(R\) with \(\theta\), of deleting the selected literal \(L_s\) and replacing it with the body of the clause, also instantiated with \(\theta\):

\[
\begin{align*}
\neg L_1, \ldots, L_q & \quad A \Leftarrow A_1, \ldots, A_p \\
\hline
\neg \theta L_1, \ldots, \theta L_{s-1}, \theta A_1, \ldots, \theta A_p, \theta L_{s+1}, \ldots, \theta L_q
\end{align*}
\]

From a logical point of view, it is easy to derive this rule, for example in natural deduction, or more directly in the sequent calculus. Hence,

\[
\begin{align*}
\neg L_1 \lor \ldots \lor \neg L_q, \\
A \Leftarrow A_1 \lor \ldots \lor A_p, \\
\vdash \neg \theta L_1 \lor \ldots \lor \neg \theta L_{s-1} \lor \neg \theta A_1 \lor \ldots \lor \neg \theta A_p \lor \theta L_{s+1} \lor \ldots \lor \theta L_q,
\end{align*}
\]

and by contraposition

\[
C \vdash \forall \exists \theta(L_1 \land \ldots \land L_{s-1} \land A_1 \land \ldots \land A_p \land L_{s+1} \land \ldots \land L_q) \Rightarrow \exists \exists(L_1 \land \ldots \land L_q).
\]
There is even a strong soundness result, true in minimal logic, which has been expected since theorem 7.2.

Lemma 7.6 If $\sigma L_s = \sigma C^-$, then

$$C \vdash_M \forall (\sigma (L_1 \land \ldots \land L_{s-1} \land C^- \land L_{s+1} \land \ldots \land L_q)) \Rightarrow \forall (\sigma (L_1 \land \ldots \land L_q)).$$

Proof The notation is simplified to show in natural deduction that

$$C \vdash_M \forall (\sigma C^- \land \sigma L') \Rightarrow \forall (\sigma L \land \sigma L')$$

as soon as $\sigma L = \sigma C^+$:

$$\frac{\forall (C^- \Rightarrow C^+)}{\sigma C^- \Rightarrow \sigma C^+} \quad \frac{\forall (\sigma C^- \land \sigma L')}{\sigma C^-} \quad \frac{\forall (\sigma C^- \land \sigma L')}{\sigma C^- \land \sigma L'}$$

$$\frac{\sigma C^+ \quad \sigma C^- \land \sigma L'}{\sigma L \land \sigma L'} \quad \frac{\forall (\sigma C^- \land \sigma L')}{\forall (\sigma L \land \sigma L')}.$$

Constraint resolution
This inference will be described as a procedure to solve equations, à la Colmerauer, by making the environment explicit, as was done for functional programming. The rules of a program are used to construct incrementally a system of equations whose solution will be that of an initial query. A solution state is a pair $E \cdot R$ formed of a system $E$ of equations to solve and a query $R$ to delete. These states must be transformed, starting from $\emptyset \cdot R_0$, to obtain a state $E_n \cdot T$, the solutions of $E_n$ being the solutions of $R_0$. There are, a priori, two ways to advance towards a solution: transform a subset of the query into equations, and transform the system of equations into a more solved form.

Example Consider the addition program

\begin{align*}
  p(U, 0, U). \\
  p(U, s(V), s(W)) & : p(U, V, W).
\end{align*}

and the query

?- p(X, s(Y), s(s(0))).

By using the second rule, and then the first,

\begin{align*}
  \emptyset \cdot p(X, s(Y), s(s(0))) \\
  \rightarrow_P \{p(X, s(Y), s(s(0))) = p(U_1, s(V_1), s(W_1))\} \cdot p(U_1, V_1, W_1) \\
  \rightarrow_P \{p(X, s(Y), s(s(0))) = p(U_1, s(V_1), s(W_1)), p(U_1, V_1, W_1) = p(U_2, 0, U_2)\} \cdot T
\end{align*}
A solution for the last system is \( X = U_1 = U_2 = W_1 = s(0), Y = V_1 = 0 \), which answers the query \(- \ p(X, s(Y), s(s(0))) \) with the substitution \([X \ Y s(0) 0]\).

It is, however, awkward to form these systems blindly, waiting for \( T \) to be derived before starting to solve them. It is better to add the equation \( L = A \) to \( E \) only if the system \( E \cup \{ L = A \} \) has a solution. This general scheme forms the basis of ‘constraint programming’. In the particular case where the equations are solved over a syntactic domain, each system can be transformed into solved form before adding a new equation.

**Example (cont.)** The first system obtained,

\[
\{ p(X, s(Y), s(s(0))) = p(U_1, s(V_1), s(W_1)) \},
\]

is transformed into the equivalent system, in solved form:

\[
\{ X = U_1, Y = V_1, W_1 = s(0) \}.
\]

Does the second equation, \( p(U_1, V_1, W_1) = p(U_2, 0, U_2) \), have a solution in this ‘environment’? This question is equivalent to solving \( p(U_1, V_1, s(0)) = p(U_2, 0, U_2) \), which means solving \( U_1 = U_2 = s(0), V_1 = 0 \).

The only pairs \( E \cdot R \) to be considered are therefore those where \( E \) is in solved form, i.e., is an idempotent substitution \( \rho \). Furthermore, only those bindings in \( \rho \) which are of interest need be preserved, i.e., those pertaining to the variables of the original query. The others can be eliminated by applying them to the current query. One can then suppose the \( \text{dom}(\rho) \) and \( \text{var}(R) \) are disjoint.

**Example (cont.)**

\[
\{ X = U_1, Y = V_1, W_1 = s(0) \} \cdot p(U_1, V_1, W_1)
\]

\[
\equiv \{ X = U_1, Y = V_1 \} \cdot p(U_1, V_1, s(0)).
\]

If the query has several literals, as in

\(- \ p(X, Y, s(s(s(s(0))))), p(X, s(s(0)), Y).\)

there is another degree of freedom: will the equations be added one by one by selecting one literal each time, or will the equations defined by all the literals be formed before they are solved? In the following example, the second rule is applied to each of the two literals, and the system is then solved globally:

\[
\varnothing \cdot p(X, Y, s(s(s(0)))) \Rightarrow p(X, s(s(0)), Y).
\]

\[
\rightarrow^P \{ p(X, Y, s(s(s(0)))) = p(U_1, s(V_1), s(W_1)) , p(X, s(s(0)), Y) = p(U_2, s(V_2), s(W_2)) \}
\]

\[
\cdot p(U_1, V_1, W_1), p(U_2, V_2, W_2)
\]

\[
\rightarrow^P \{ Y = s(V_1) \} \cdot p(X, V_1, s(s(s(0)))) , p(X, s(0), V_1).
\]
This situation is comparable to that of choosing a redex to be contracted in the \( \lambda \)-calculus. The \textit{confluence} property showed that this choice, so long as it remains free at each step, does not affect the result. However, the reduction strategy was not neutral with respect to termination. Here there is a \textit{switching} property, to be proven later on, which follows from the fact that a system of equations can be solved in any order. It is easier to work by selecting a single literal each time, and one can even impose a selection rule for all queries—SLD-resolution—which will be defined using the following concepts.

\textbf{Definition 7.2}

- A \textit{(logical) environment} \( \rho \) is an idempotent substitution. Write 1 for the identity substitution of empty domain.
- A \textit{query} is a list of atoms, written \( \mathcal{L} \): \( L_1, \ldots, L_q \). The empty query is written \( \top \).
- A \textit{solution state} is a pair \( \rho \cdot R \), where \( \rho \) is an environment, \( R \) is a query and \( \text{dom}(\rho) \cap \text{var}(R) = \emptyset \). A state is \textit{initial} if it is of the form \( 1 \cdot R \) and is a \textit{success state} if it is of the form \( \rho \cdot \top \).

The component \( \rho \) represents a partial computation of the solution, and the derived query \( R \) indicates in which direction \( \rho \) must grow to reach a solution.

Normalizing the resolution procedure in this way is a simplification which will have to be abandoned in less sequential implementations, where a unification can be ‘suspended’ so long as a certain condition is not satisfied. The most general unifier cannot then be computed immediately, and the bindings are only ‘broadcast’ to certain variables.

\textbf{SLD-derivations}

Let \( P \) be a definite program, i.e., a list of definite clauses. The program \( P \) defines a transition relation between states tagged by pairs \( (s, C) \), where \( s \) is a literal occurrence and \( C \) a clause. This relation will be written \( \rightarrow_P \), or simply \( \rightarrow \).

A \textit{separate variant} of a clause is a clause renamed using variables hereto unused: sometimes phrases such as ‘separate from \( \text{var}(R) \)’, or simply ‘separate from \( R \)’, will be used. In what follows, let \( V \) be a fixed set of variables. It will be used as the set of variables of an initial query.

\textbf{Definition 7.3} Let \( \rho \cdot R \) be a state with \( R \neq \top \); \( R/s = L_s \) be the literal of \( R \) at occurrence \( s \); and \( C \) be a variant of a clause of \( P \), separate from \( R \) and \( V \), with body \( C^- = A_1, \ldots, A_p \) and head \( C^+ \). Form the system \( \rho \cup \{ R/s = C^+ \} \), equating the substitution \( \rho \) to the system in reduced form which defines it and for which it is a most general unifier.

If the system has no solution, there does not exist a tagged transition \( (s, C) \) from \( \rho \cdot R \).

Otherwise, every solution of \( \rho \cup \{ R/s = C^+ \} \) is a solution of \( \rho \), hence an instance \( \mu \rho \) of \( \rho \). By the separation assumption on \( C \), and by the definition of states, \( \rho \) does
not operate over $R/s = C^+$. Hence $\mu$ is a solution of $R/s = C^+$. So a most general unifier of $\rho \cup \{ R/s = C^+ \}$ is of the form $\theta \rho$, where $\theta$ is a most general unifier of $R/s = C^+$. Then define $\rho \cdot R \xrightarrow{C} P \rho' \cdot R'$, where

$$\rho' = \theta \rho,$$
$$R' = \theta L_1, \ldots, \theta L_{s-1}, \theta A_1, \ldots, \theta A_p, \theta L_{s+1}, \ldots, \theta L_q.$$  

If $q = 1$ and $p = 0$ let $R' = T$; it is a success state. The literals $\theta A_1, \ldots, \theta A_p$ are introduced by this transition, and $\theta L_i, i \neq s$ is the residue of $L_i$.

Note that this concept of residue is much simpler than that of $\beta$-reduction, where there can also be reduction and multiplication of redexes.

This rule is written in an abbreviated manner:

$$\frac{\{ R/s = C^+ \} \xrightarrow{\star} \theta}{\rho \cdot R \rightarrow_P \theta \rho \cdot \theta(R[s \leftarrow C^+])}.$$  

Some of the terminology of reduction relations (from Chapter 2) will be used.

An (SLD-)derivation is a finite or infinite sequence

$$\rho_0 \cdot R_0 \xrightarrow{s_0, C_0} \cdots \xrightarrow{s_{n-1}, C_{n-1}} \rho_n \cdot R_n \xrightarrow{s_n, C_n} \cdots$$

Note that each $\rho_i \cdot R_i$ of this derivation is of the form:

$$\theta_1 \cdots \theta_i \rho_0 \cdot \theta_i \cdots \theta_1 \rho_0 R_0[s_0 \leftarrow C_0] \cdots [s_{i-1} \leftarrow C_{i-1}].$$

A derivation can be infinite. If it is not, it succeeds if it halts in a success state $\rho_n : T$, where $n$ is the length of the derivation.

**Definition 7.4** The result substitution of a successful derivation $1 \cdot R_0 \xrightarrow{\star} \rho_n \cdot T$, or answer to $R_0$, is the restriction to $\text{var}(R_0)$ of $\rho_n$. The result query is $\rho_n R_0$.

In the definition of a resolution step, it would be possible to restrict the environment component of each state to $V$, but the following presentation would be unnecessarily complicated.

**Fundamental properties**

The $\rightarrow_P$ relation satisfies several important properties. The logical ones are strong (constructive) soundness and completeness, and the others are particular to this form of computation: generalization and switching.

**Soundness**

**Theorem 7.7 (Strong soundness)** If $\rho \cdot R \xrightarrow{\star} P \rho' \cdot R'$, then $P \vdash \forall(R') \Rightarrow \forall(\rho' R)$. In particular, the result substitution $\rho$ of a successful derivation is a solution of $R$. 

Proof. By induction on the length of the derivation. The null length derivation case is trivial. Let

$$\rho \cdot R \xrightarrow{s, C} \theta \rho \because \theta \rho R[s \leftarrow C'] \xrightarrow{*} \rho'/\theta \rho \cdot R'$$

be a derivation. By the inductive hypothesis,

$$P \vdash_M \forall(R') \Rightarrow \forall(\rho'/\theta \rho R[s \leftarrow C']).$$

Furthermore, $\theta$ is a solution of $R/s = C^+$, as is $\rho'/\theta$. Lemma 7.6 applies to the first transition, giving

$$P \vdash_M \forall(\rho'/\theta \rho R[s \leftarrow C']) \Rightarrow \forall(\rho'/\theta \rho R).$$

By transitivity, $P \vdash_M \forall(R') \Rightarrow \forall(\rho'/\theta \rho R)$.

Note that the soundness of SLD-resolution does not depend on the choice of most general unifier: every solution to the equations $R/s = C^+$ is satisfactory.

This result justifies the concept of a qualified answer to a query, which can be used for the analysis of programs (debugging), or, in the case of a failed derivation, to find the last constructed state. If $1 \cdot R \xrightarrow{\rho} R'$, then the pair $(\rho, R')$ is a qualified (or conditional) answer: if $\forall(R')$ is true, $\rho$ is a solution.

By ignoring the result $\rho$, the theorem states that a successful derivation of $R$ constitutes a proof of $P \vdash \exists \exists R$, i.e. it is a refutation of $P \land \neg (\exists \exists R)$, or a refutation of the clausal formula $P \land R'$. By omitting the program, a successful derivation from $1 \cdot R$ is called a refutation of the goal $R'$.

A successful derivation of $R$ computes a solution to $R$ by successive approximations, using the increasing sequence $R \leq \rho_1 R \leq \rho_2 R \leq \ldots \leq \rho_n R$ until $\rho_n$ is a solution.

Expansions of a query. A successful derivation provides a solution to a query. An arbitrary finite derivation can only provide a conditional solution. An infinite derivation allows a finer and finer approximation, assuming it is 'fair'.

Definition 7.5 A derivation is fair if it is finite, or if for every $n$ and every literal $L$ appearing in it, $L$ or one of its residues is ultimately selected.

The Herbrand expansion $\operatorname{Exp}(\exists(R), T_\Sigma)$ would be the conjunction of all the closed instances of the literals of $R$, which is infinite if $T_\Sigma$ is infinite. Instead, the set of instances is used, written $\operatorname{Exp}(R)$: it is the expansion of query $R$. Since the formula is existential, $\operatorname{Exp}(R) \vdash_M \exists(R)$.

Proposition 7.8

1. If the successful derivation $1 \cdot R \xrightarrow{\rho_m} \top$ is of length $m$, $\operatorname{Exp}(\rho_m R) \subseteq T_P^{1m}$.
2. If the infinite derivation $1 \cdot R \rightarrow \ldots \rightarrow \rho_n R_n \rightarrow \ldots$ is fair, then for every $m$ there exists $n$ such that $\operatorname{Exp}(\rho_n R) \subseteq T_P^{1m}$.
Proof

(1) By induction on $m$.
If $m = 0$, then $R \equiv T$ and $\rho_0 = I$. Hence $\text{Exp}(T) = \emptyset = T_P^{10}$.
If $m > 0$, consider the derivation of length $m$,

$$1 \cdot R \xrightarrow{s,C} \theta \cdot \theta R[s \leftarrow C^-] \xrightarrow{\ast} \rho \theta \cdot T.$$ 

Then $\text{Exp}(\rho \theta R) = \text{Exp}(\rho \theta (R \setminus s)) \cup \text{Exp}(\rho \theta C^+)$. By the inductive hypothesis applied to the successful derivation from $\theta R[s \leftarrow C^-$], and by the monotonicity of $T_P$,

$$\text{Exp}(\rho \theta (R \setminus s)) \subseteq \text{Exp}(\rho \theta R[s \leftarrow C^-]) \subseteq T_P^{1m-1} \subseteq T_P^{1m}$$

$$\text{Exp}(\rho \theta C^-) \subseteq \text{Exp}(\rho \theta R[s \leftarrow C^-]) \subseteq T_P^{1m-1},$$

$T_P$ can be applied to obtain $C^+$ from $C^-$,

$$\text{Exp}(\rho \theta C^+) \subseteq T_P(T_P^{1m-1}) = T_P^{1m}.$$

It follows that $\text{Exp}(\rho \theta R) \subseteq T_P^{1m}$.

(2) By induction on $m$.

The case $m = 0$ is trivial: $\text{Exp}(R) \subseteq T_P^{10} = \text{At}_E$.
Let $m > 0$. Let $i$ be one of the literal occurrences of $R$. By the fairness assumption, there exists a $p$ such that $R/i$ is selected at the $p$-th step of the derivation,

$$1 \cdot R \xrightarrow{s} \rho_p \cdot \rho_p R_p \xrightarrow{s,C} \theta \rho_p \cdot \theta \rho_p R_p[s \leftarrow C^-],$$

with $\theta \rho_p R_p[i] = \theta C^+$. By the inductive hypothesis applied to the (also fair) derivation from $(p + 1)$-th step and to $m - 1$, there exists $q$ such that

$$\theta \rho_p \cdot \theta \rho_p R_p[s \leftarrow C^-] \xrightarrow{\ast} \rho'_q \theta \rho_p \cdot R'_{p+1+q},$$

$$\text{Exp}(\rho'_q \theta \rho_p R_p[s \leftarrow C^-]) \subseteq T_P^{1m-1}.$$

It follows that:

$$\text{Exp}(\rho'_q \theta \rho_p C^-) \subseteq \text{Exp}(\rho'_q \theta \rho_p R_p[s \leftarrow C^-]) \subseteq T_P^{1m-1},$$

$$\text{Exp}(\rho'_q \theta \rho_p (R/i)) \subseteq T_P(\text{Exp}(\rho'_q \theta \rho_p C^-)) \subseteq T_P(T_P^{1m-1}) = T_P^{1m}.$$

Let $n_i = p + 1 + q$, so that $\text{Exp}(\rho_n(R/i)) \subseteq T_P^{1m}$, with $\rho_n = \rho'_q \theta \rho_p$. Let $n = \max_i n_i$; since $\rho_n \leq \rho_n$, $\text{Exp}(\rho_n(R/i)) \subseteq \text{Exp}(\rho_n(R/i))$, hence:

$$\text{Exp}(\rho_n R) = \bigcup_i \text{Exp}(\rho_n(R/i)) \subseteq T_P^{1m}.$$

Since $\text{Exp}(A) = \{A\}$ if $A$ is a closed atom, the following corollary is obtained.

Corollary 7.9 Let $A \in \text{At}_E$.

- If $A$ has a successful derivation, then $A \in T_P^{1w}$.
- If $A$ has a fair infinite derivation, then $A \in T_P^{1w}$. 
Switching  So the derivations $1 \cdot R \xrightarrow{\star P} \rho \cdot \top$ must be constructed. This solution search is non-deterministic: at each step, a literal and a clause must be selected. The following theorem ensures that the non-determinism in the choice of literal does not matter: this is called don’t care non-determinism.

Theorem 7.10 (Switching) If in a query $R$, the literals $L_{s_1}$ and the residue of $L_{s_2}$ are successively deleted, they can also be deleted in the reverse order, and the derived states are the same up to renaming of variables.

Proof Suppose that the literals $L_{s_1}$ and $L_{s_2}$ of $R$ are selected in the following order:

\[
\frac{1}{P} \rho \cdot \ldots, L_{s_1}, \ldots, L_{s_2}, \ldots,
\]

\[
\frac{2}{P} \theta_1 \rho \cdot \theta_1(\ldots, C_1^-, \ldots, L_{s_2}, \ldots),
\]

\[
\frac{2}{P} \theta_2 \theta_1 \rho \cdot \theta_2 \theta_1(\ldots, C_1^-, \ldots, C_2^-, \ldots).
\]

One can start by selecting $L_{s_2}$ in $R$, to show that $\theta_2 \theta_1 L_{s_2} = \theta_2 \theta_1 C_2^+$ holds (since, by assumption, $R$, $C_1$ and $C_2$ are separate, hence $\theta_1$ leaves $C_2^+$ invariant): $\theta_2 \theta_1$ is a solution to the equation $L_{s_2} = C_2^+$, which therefore has a most general unifier $\theta_2'$, and there exists a substitution $\mu_2$ such that

\[
\theta_2 \theta_1 = \mu_2 \theta_2'.
\]  \tag{7.1}

The selection of $L_{s_2}$ in $R$ succeeds:

\[
\rho \cdot \ldots, L_{s_1}, \ldots, L_{s_2}, \ldots,
\]

\[
\frac{2}{P} \theta_2' \rho \cdot \theta_2'(\ldots, L_{s_1}, \ldots, C_2^-, \ldots).
\]

One can then select $\theta_1' L_{s_1}$. Because $\theta_1 L_{s_1} = \theta_1 C_1^+$, from (7.1), and by the separation assumption, it follows that:

\[
\mu_2 \theta_2' L_{s_1} = \theta_2 \theta_1 L_{s_1} = \theta_2 \theta_1 C_1^+ = \mu_2 \theta_2' C_1^+ = \mu_2 C_1^+.
\]

Hence, $\mu_2$ is a solution to $\theta_2' L_{s_1} = C_1^+$, which therefore has a most general unifier $\nu_1'$, and there exists a substitution $\nu_1$ such that

\[
\mu_2 = \nu_1 \theta_1'.
\]  \tag{7.2}

Hence the selection of $\theta_2' L_{s_1}$ succeeds in turn:

\[
\rho \cdot \ldots, L_{s_1}, \ldots, L_{s_2}, \ldots,
\]

\[
\frac{2}{P} \theta_2' \rho \cdot \theta_2'(\ldots, L_{s_1}, \ldots, C_2^-, \ldots),
\]

\[
\frac{1}{P} \theta_1' \theta_2' \rho \cdot \theta_1' \theta_2'(\ldots, C_1^-, \ldots, C_2^-, \ldots).
\]

The two derived queries,

\[
\theta_2 \theta_1 \rho \cdot \theta_2 \theta_1(\ldots, C_1^-, \ldots, C_2^-, \ldots),
\]
Semantics of definite programs 273

and

\[ \theta'_1 \theta'_2 \rho \cdot \theta'_1 \theta'_2 (\ldots, C_1^-, \ldots, C_2^-, \ldots), \]

must still be shown to be variants.

So, by equations (7.1) and (7.2): \( \theta_2 \theta_1 = \mu_2 \theta'_2 = \nu_1 \theta'_1 \theta'_2 \), i.e., \( \theta_2 \theta_1 \geq \theta'_1 \theta'_2 \).

Conversely, the reasoning leading to \( \mu_2 \) and \( \nu_1 \) can be done symmetrically. First,

\[ \theta'_1 \theta'_2 \mu_{s_1} = \theta'_1 C_1^+ = \theta'_1 \theta'_2 C_1^+; \quad \theta_1 \theta_2 \text{ is a solution of } L_{s_1} = C_1^+, \]

hence

\[ \theta'_1 \theta'_2 = \mu_1 \theta_1 \quad (7.3) \]

for a substitution \( \mu_1 \). Second, from \( \theta'_2 \mu_{s_2} = \theta'_2 C_2^+ \), from (7.3), and by the separation assumption, it follows that:

\[ \mu_1 \theta_1 \mu_{s_2} = \theta'_1 \theta'_2 \mu_{s_2} = \theta'_1 \theta'_2 C_2^+ = \mu_1 \theta_1 C_2^+ = \mu_1 C_2^+, \]

i.e., \( \mu_1 \) is a solution of \( \theta_1 \mu_{s_2} = C_2^+ \), hence

\[ \mu_1 = \nu_2 \theta_2 \quad (7.4) \]

for a substitution \( \nu_2 \). Therefore, from (7.3) and (7.4), \( \theta'_1 \theta'_2 = \mu_1 \theta_1 = \nu_2 \theta_2 \theta_1 \), i.e., \( \theta'_1 \theta'_2 \geq \theta_2 \theta_1 \).

To conclude, \( \theta_2 \theta_1 \) and \( \theta'_1 \theta'_2 \) are variants.

**Remark** The switching property is weaker than confluence. Let \( P \) be the program \( p(a,a), p(b,b) \). From the same query, the two derivations:

\[ 1 \cdot p(a,X), p(b,X) \xrightarrow{1,1} p(a,b), \]

\[ 1 \cdot p(a,X), p(b,X) \xrightarrow{2,2} p(a,b), \]

are not confluent.

However, the choice of clause is done by a don't know non-determinism. Since all the possible successful derivations must be accounted for, they are organized in the form of a tree.

A search state is a pair \((\rho \cdot R, s)\), where \( \rho \cdot R \) is a resolution state and \( s \) is a literal occurrence in \( R \). When \( R = \top \), there is no occurrence to indicate. A search (or resolution) tree for \( R_0 \) is a tree whose nodes are tagged by the search states. The initial state \( 1 \cdot R_0 \) appears at the root and a non-terminal \((\rho \cdot R, s)\) has as child all of the \( \rho' \cdot R' \) such that \( \rho \cdot R \xrightarrow{C,\rho} \rho' \cdot R' \) for each clause \( C \) of \( P \). Since \( P \) is a list of definite clauses, the tree is ordered. A terminal node is either a success state \( \rho \cdot \bot \), or a failure state \((\rho \cdot R, s)\) in the case where there is no transition \( \rho \cdot R \xrightarrow{C,\rho} \ldots \), i.e., for each clause \( C \), equation \( R/s = C^+ \) has no solution. The same query can have search trees of many forms, finite or infinite. Figure 7.3 on page 277 shows a 'standard' search tree for a program computing paths in a graph.
Completeness  It will be shown that all the solutions of a query \( R \) can be obtained by constructing an arbitrary search tree for \( R \).

Lemma 7.11 (Generalization) If \( 1 \cdot \lambda R \xrightarrow{*} \rho \cdot \rho \lambda R' \), then there exists a substitution \( \sigma \leq \rho \lambda \) and a derivation \( 1 \cdot R \xrightarrow{*} \sigma \cdot \sigma R' \).

Proof  By induction on the length of the derivation from \( 1 \cdot \lambda R \).

The property is trivial for a derivation of length zero, with \( \sigma = 1 \leq \lambda \).

Consider a derivation

\[
1 \cdot \lambda R \xrightarrow{*} \rho \cdot \rho \lambda R' \xrightarrow{s,C} \theta \rho \cdot \theta \rho \lambda R'[s \leftarrow C'].
\]

By the inductive hypothesis, there exists a derivation

\[
1 \cdot R \xrightarrow{*} \sigma \cdot \sigma R',
\]

with \( \sigma \leq \rho \lambda \), i.e., \( \rho \lambda = \mu \sigma \), for a particular substitution \( \mu \). Rename the variables of \( C \) outside the domain of \( \mu \), so that \( \mu C^+ = C^+ \). From, \( \theta \rho \lambda(R/s) = \theta C^+ = \theta \mu C^+ \), it follows that the equation \( \sigma(R'/s) = C^+ \) has a solution, \( \theta \mu \). If \( \theta' \) is a most general solution for it, \( \theta \mu = \nu \theta' \) for a particular substitution \( \nu \). By equation (7.5), the derivation can therefore be extended by

\[
\sigma \cdot \sigma R' \xrightarrow{s,C} \theta' \sigma \cdot \theta' \sigma R'[s \leftarrow C'],
\]

with \( \theta \rho \lambda = \theta \mu \sigma = \nu \theta' \sigma \geq \theta' \sigma \).

For the \( R' = \top \) case, there is

\[
\square
\]

Corollary 7.12 If \( 1 \cdot \lambda R \xrightarrow{*} \rho \cdot \top \), then \( 1 \cdot R \xrightarrow{*} \sigma \cdot \top \) for a \( \sigma \leq \rho \lambda \).

The following lemma states the completeness of SLD-resolution, restricted to closed atoms.

Lemma 7.13 If \( L \in M_P \), there exists a successful derivation \( 1 \cdot L \xrightarrow{*} 1 \cdot \top \).

Proof  From proposition 7.5, \( M_P \) is the subset of \( At_\Sigma \) inductively defined by \( P \), which allows a proof by induction over \( P \).

If \( L \equiv \theta A \) is an instance of a unit clause of \( P \), then \( 1 \cdot L \xrightarrow{\downarrow} 1 \cdot \top \).

If \( L \equiv \theta A \), for a clause \( A : = A_1, \ldots, A_p \), and if \( P \vdash \theta A_1, \ldots, \theta A_p \), there exists, by the inductive hypothesis, successful derivations \( 1 \cdot \theta A_i \xrightarrow{*} 1 \cdot \top \) for each \( i \). By combining these successful derivations in an arbitrary order, one obtains:

\[
1 \cdot \theta A_1, \theta A_2, \ldots, \theta A_p
\]

\[
\xrightarrow{*} 1 \cdot \theta A_2, \ldots, \theta A_p
\]

\[
\xrightarrow{*} 1 \cdot \theta A_p
\]

\[
\xrightarrow{*} 1 \cdot \top.
\]
Let $\theta' = \theta \mid \text{var}(A)$. It is the most general unifier of $\theta A$ and $A$. So there is a transition

$$1 \cdot \theta A \rightarrow \theta' \cdot \theta' A_1, \ldots, \theta' A_p.$$  

Since $\theta' \leq \theta$, by the generalization lemma, there exists a derivation

$$1 \cdot \theta' A_1, \ldots, \theta' A_p \xrightarrow{\ast} 1 \cdot \top,$$

and by combining these derivations,

$$1 \cdot \theta A \rightarrow \theta' \cdot \theta' A_1, \ldots, \theta' A_p \xrightarrow{\ast} \theta' \cdot \top,$$

where the substitution $\theta'$ has as domain a set of variables not appearing in $\theta A$, and so can replaced by $1$ in the final result. □

This lemma can be extended to $L \in \text{At}_{\Sigma}[X]$, by considering the variables of $L$ as constants, which means working in $\Sigma \cup \text{var}(L)$ instead of $\Sigma$:

Lemma 7.14 Let $L \in \text{At}_{\Sigma}[X]$. If $P \models L$, there exists a successful derivation $1 \cdot L \xrightarrow{\ast} 1 \cdot \top$.

Theorem 7.15 (Strong completeness) If $\theta \in [R]_P$, there exists a successful derivation $1 \cdot R \xrightarrow{\ast} \rho \cdot \top$ such that $\rho \leq \theta$.

**Proof** By definition, $\theta \in [R]_P$ means that $P \models \theta R$. If $R \equiv L_1, \ldots, L_q$, it follows that $P \models \theta L_i$ for every $i$. From lemma 7.14, there exists for each $i$ a successful derivation $1 \cdot \theta L_i \xrightarrow{\ast} 1 \cdot \top$. Combining these successful derivations yields: $1 \cdot \theta R \xrightarrow{\ast} 1 \cdot \top$. From the generalization lemma, there exists a successful derivation $1 \cdot R \xrightarrow{\ast} \rho \cdot \top$, with $\rho \leq \theta$. □

To obtain the existence of a result $\rho \leq \theta$ in an arbitrary search tree, it suffices to apply the switching theorem.

Figure 7.1 illustrates the completeness theorem by representing, for a query $R$, three answers $\theta R$, $\theta' R$ and $\theta'' R$, partial solutions $\rho_1 R$ and $\rho_2 R$, and a solution $\sigma R$ which is an instance of one of the answers.

### 7.3 Control structures

Going from the general framework of logic programming to Prolog consists of choosing a selection rule and a search strategy. Once those choices are known, a programmer can think procedurally, and even ignore the logical basis.

#### 7.3.1 Selection and searching

A *selection rule* is a mapping associating with each derivation $1 \cdot R \xrightarrow{\ast} \rho \cdot R'$ a literal occurrence in $R'$. Each selection rule $S$ defines a unique search tree, whose
internal nodes are tagged by \((\rho \cdot R, S(\varpi))\), where \(\varpi\) is the derivation corresponding to the path from the root of the tree to the node.

By the switching theorem, adopting a selection rule is not a restriction: if there is a successful SLD-derivation for \(R\), there is also a successful derivation using \(S\).

The standard Prolog rule selects the left literal of each query: \(R/1\). It is a local rule: its result depends only on the final state of the derivation. Note that if a query reappears in a derivation, it is always the same literal which is selected. Although an important simplification, it tends to—when combined with depth-first traversal of the tree—create infinite loops, as does call by value in functional programming.

Although an important simplification, this rule tends to create infinite branches in the search tree. Any traversal of such a tree implies the non-termination of the query-solving; moreover, an 'eager' traversal, which steadily follows an infinite branch, prevents any answer from being issued, as does call by value in functional programming. Completeness can only be guaranteed by traversing the search tree in an exhaustive manner, to enumerate all of the answers: this is performed, for instance, by breadth-first traversal.

A search strategy is necessary to define how the search tree is to be traversed. In Prolog, the standard strategy is 'depth-first'. This choice allows one not to construct the entire tree, and to use a stack whose elements are the successively pushed states of a branch of the tree. They are popped while backtracking.

This choice of search strategy makes the completeness property of SLD-resolution irrelevant for the user: by traversing the tree depth-first, no solution to the right of an infinite branch will be found. To avoid this situation, the order of clauses and of literals within a clause can sometimes be changed.

### 7.3.2 The activation-box model

L. Byrd introduced an execution model which is used by most Prolog systems in trace and debug mode.

To each node \((\rho \cdot R, s)\) of the search tree corresponds a call \(R/s\) of the procedure
associated with the predicate $R(s)$ (in the standard case, $s = 1$). Each call $R/s$ is represented as an activation box with four gates: call, exit, redo and fail, through which the control flow passes, as shown in Figure 7.2. For a given box, the trace of the flow belongs to the language defined by the rational expression \( \text{call}(\text{exit}|\text{redo})^*\text{fail} \).

call the (initial) activation of the procedure, before unification with a clause head; a number is assigned to the box;
exit successful exit from the procedure, all the calls created after the call entry were successful; corresponds to a partial derivation of the form $(\rho \cdot R, s) \xrightarrow{*} (\emptyset \rho \cdot \emptyset (R \setminus s), ..)$;
redo call reentering the (already active) procedure triggering a unification attempt of $R/s$ with a new clause of the procedure;
fail (final) exit from the procedure.

![Figure 7.2 An activation box](image)

![Figure 7.3 A standard search tree](image)

**Graphs** The search tree in Figure 7.3 is a standard example from the relational calculus. A predicate \( \text{arc}/2 \), specifying a graph, is given extensionally, for example:

\[
\begin{align*}
\text{arc}(a, b). \\
\text{arc}(b, c). \\
\text{arc}(a, d).
\end{align*}
\]
The predicate path/2 states the existence of a (directed) path between two vertices of the graph:

\[
\text{path}(U, U).
\]
\[
\text{path}(U, W) \leftarrow \text{arc}(U, V), \text{path}(V, W).
\]

In this definition, the order of the rules, and for the second rule, the order of the literals in its right-hand side, affect the results:

?\- \text{path}(a, Y).

The search tree is finite. The execution is visible in trace mode:

\(\sim\)

(0) 0 call: \text{path}(a, \_VGML)?
(0) 0 exit: \text{path}(a, a)?

\(Y = a;\)
(0) 0 redo: \text{path}(a, \_VGML)?
(1) 1 call: \text{arc}(a, \_VGTL)?
(1) 1 exit: \text{arc}(a, b)?
(2) 1 call: \text{path}(b, \_VGML)?
(2) 1 exit: \text{path}(b, b)?
(0) 0 exit: \text{path}(a, b)?

\(Y = b;\)
(0) 0 redo: \text{path}(a, \_VGML)?
(2) 1 redo: \text{path}(b, \_VGML)?
(3) 2 call: \text{arc}(b, \_VHBX)?
(3) 2 exit: \text{arc}(b, c)?
(4) 2 call: \text{path}(c, \_VGML)?
(4) 2 exit: \text{path}(c, c)?
(2) 1 exit: \text{path}(b, c)?
(0) 0 exit: \text{path}(a, c)?

\(Y = c;\)
(0) 0 redo: \text{path}(a, \_VGML)?
(2) 1 redo: \text{path}(b, \_VGML)?
(4) 2 redo: \text{path}(c, \_VGML)?
(5) 3 call: \text{arc}(c, \_VHJY)?
(5) 3 fail: \text{arc}(c, \_VHJY)?
(4) 2 fail: \text{path}(c, \_VGML)?
(3) 2 redo: \text{arc}(b, \_VGML)?
(3) 2 fail: \text{arc}(b, \_VHBX)?
(2) 1 fail: \text{path}(b, \_VGML)?
(1) 1 redo: \text{arc}(a, \_VGTL)?
(1) 1 exit: \text{arc}(a, d)?
(2) 1 call: \text{path}(d, \_VGML)?
(2) 1 exit: \text{path}(d, d)?
(0) 0 exit: \text{path}(a, d)?

\(Y = d;\)
(0) 0 redo: path(a, _VGML)?
(2) 1 redo: path(d, _VGML)?
(3) 2 call: arc(d, _VHbx)?
(3) 2 fail: arc(d, _VHbx)?
(2) 1 fail: path(d, _VGML)?
(1) 1 redo: arc(a, _VGTW)?
(1) 1 fail: arc(a, _VGTW)?
(0) 0 fail: path(a, _VGML)?

fail.

The integer appearing in parentheses uniquely designates a predicate call. The
following integer is the number of ancestors in its derivation. Variables starting
with a _ are newly generated variables. Each line corresponds to the input or
output of a process through one of the four gates.

Should the second rule be changed to be left-recursive,

\[
\text{path}(U, W) :- \text{path}((U, V), \text{arc}(V, W)).
\]

the search tree has an infinite branch formed of queries

\[
\text{path}(a, V_n), \text{arc}(V_n, V_{n-1}), \ldots, \text{arc}(V_2, V_1), \text{arc}(V_1, Y)
\]
such that the other branches (always to its left) fail as soon as \( n > 2 \), since there is
no path of length \( \geq 2 \) in this example. After having enumerated the four solutions,
the interpreter chooses the infinite branch and stays there. If the two clauses
defining path are permuted, the tree is transformed by reflection, and the infinite
branch is now leftmost, so the interpreter will give no answer.

### 7.3.3 Cuts

It is the cut which ensured Prolog’s success, and it is as controversial as the GOTO
and COMMON of FORTRAN. Attempts have been made to replace it with more elabo-
rate control structures (coroutines in Prolog-II+ and NU-Prolog), more symmetric
ones (commitment in concurrent programming), or more logical ones (Lloyd’s
declarative pruning procedure).

The syntax for a cut is !, and it can appear anywhere in the body of a rule. It
has no declarative meaning. The choice of word, cut, is perhaps not the best, since
the basic mechanism of Prolog is the cut of the sequent calculus.

The cut is a global control structure, which only affects the search tree. At the
local transition level, a cut succeeds as soon as it is selected.

In the following discussion, the environment component will be omitted, as it
plays no role. Let \( R_0 \rightarrow R_1 \) be a transition such that the left literal of \( R_1 \) triggers
a rule \( A :- \ldots, !, \ldots \). Then \( R_0 \rightarrow R_1 \rightarrow (\ldots, !, \ldots), R_2 \). If there exists a
derivation

\[
R_0 \rightarrow R_1 \rightarrow (\ldots, !, \ldots), R_2 \xrightarrow{!}, R_n
\]
(all the queries to the left of ! have been deleted), then the search continues normally below !, $R_n$, whose unique descendent is $R_n$. If the standard traversal backtracks to $R_n$, then the control immediately passes to $R'_m$ which is the query derived from $R_0$ which follows $R_1$, according to the standard selection rules. The preceding clauses between $R_1$ and $R_n$ are not active: the ! therefore prunes certain branches of the tree, and short-circuits the normal traversal by passing from $R_n$ to $R'_1$ (Figure 7.4). Should one of the literals to the left of the ! not succeed, the search continues normally.

**Example** In the first position of a rule $A : !, \ldots$, the rules which follow which could be activated are not:

- $\text{member}(X, [X|\_]) : !.$
- $\text{member}(X, [\_|Y]) : \text{member}(X, Y).$

Although all the solutions to a query can be found, it sometimes occurs that only the existence of a solution is necessary: the search can then be stopped as soon as the first solution is found.

A cut is called *green* if it does not prune any branches leading to solutions; otherwise, it is *red*. A green cut is logically correct: it is a way to limit the don’t know non-determinism of Prolog and to make programs more efficient. A red cut modifies the declarative meaning of programs. Unfortunately, the ‘color’ of a cut cannot be determined statically. Modern Prologs offer constructions which respect the declarative meaning of programs and fulfill some of the functions of the cut.

**Example** Two vertices a and b of a graph can be shown to be in the same connected component with predicate $\text{path}(X,L,Y)$:

- $\text{path}(A, [], A).$
- $\text{path}(A, [B|L], C) : \text{arc}(A,B), \text{path}((B,L,C)).$
which constructs all of the paths \( L \) leading to \( X \) from \( Y \). It is silly to simply call \( \text{path}(a, L, b) \), since the values of \( L \) are of no interest. Rather than use a cut, it is preferable to call the explicitly quantified query:

\[
?\neg \text{ some } L \text{ path}(a, L, b).
\]

which is possible in NU-Prolog.

### 7.3.4 Non-standard control

Several Prologs do not use the standard control and give ways of specifying the conditions under which a literal will be selected. IC-Prolog offers variable annotations, MU-Prolog the \textit{wait}, NU-Prolog the \textit{when} and Prolog-II+ the \textit{freeze}. In addition, concurrent logic programming offers new control structures (guards, protected variables).

\textit{Coroutines}

In NU-Prolog, it is in general the left literal which is selected. However, if certain conditions are not satisfied, this selection is suspended, and another literal is selected. The suspended literal will be awoken as soon as the conditions are satisfied, through the resolution of the other literals which will have ‘broadcast’ their value to those variables. This mechanism is employed when a predicate is the object of a suspension declaration (when), for certain predefined predicates, for evaluating predicates and for the treatment of negation and quantifiers. The basic idea is to suspend a call if it is not ‘safe’.

Often, for the same program, certain queries have a satisfactory behavior, while others do not.

\textit{Example}  The list concatenation program

\[
\begin{align*}
\text{append}([], Y, Y). \\
\text{append}([H | T], Y, [H | TY]) & : \text{ append}(T, Y, TY).
\end{align*}
\]

works correctly for queries concatenating two lists or computing the decompositions of the third argument, as in

\[
?\neg \text{ append}([a], [b], Z)
\]

or

\[
?\neg \text{ append}(X, Y, [1, 2, 3])
\]

but give an infinite number of results in response to the query

\[
?\neg \text{ append}(X, [a], Z)
\]

\[
\sim X = [], \\
Z = [a]
\]
\( \lnot X = [U_1, \ldots, U_n], \)
\[ Z = [U_1, \ldots, U_n, a]; \ldots \]

But it is difficult to master the form of a call to append when the literal appears in
the middle of a computation. Consider, for example, the program to concatenate
three lists:

\[
\text{append3}(X, Y, Z, XYZ) :- \\
\quad \text{append}(X, Y, XY), \\
\quad \text{append}(XY, Z, XYZ).
\]

A query to decompose a list immediately introduces an \texttt{append} with three variables:

\[
\text{append3}(X,Y,Z,[a,b,c]) \rightarrow \text{append}(X,Y,XY), \text{append}(XY,Z,[a,b,c])
\]

which will get lost in an infinite branch once the ten decompositions of \([a,b,c]\)
have been made. In this case, the second literal is safe and should be selected
before the first. The standard rule is avoided by using a \texttt{when} declaration:

\[
?\leftarrow \text{append}(A, B, C) \text{ when } A \text{ or } C.
\]

which suspends a selection of \texttt{append} so long as its first and third arguments are
variables, and so allows it if the first or third is not a variable (the predicate \texttt{append}
is predefined with this declaration).

\[
\square
\]

\textbf{Fairness}

A selection rule is fair if all of the derivations which it defines are fair. The standard
rule is not fair: with the left-recursive clause \( p :- p, q, \) there is an unfair infinite
derivation \( p \rightarrow p, q \rightarrow p, q, q \rightarrow \ldots, \) since the literal \( q \) is never selected. The
search tree is infinite. It will be shown that fairness is a crucial property in the
theoretical treatment of negation.

Here is a fair rule: select at each step the leftmost residual redex, and if there are
none, select one of the introduced radicals. For this rule, the derivation \( p \rightarrow p, q \)
fails immediately if there is no head clause \( q \) and the search tree is finite.

\textbf{Nondeterminism}

The don't know non-determinism of Prolog appears because of the choice of clause,
and from the existence of multiple solutions to a query.

For a given program, the predicates can be distinguished according to their
degree of activity. An active predicate tends to instantiate its arguments, hence to
produce solutions. A passive predicate acts as a test which checks that a relation
is satisfied or fails without instantiating its arguments, hence without producing
any value. The same predicate can be active and passive.

\[
\text{member}(X, [X|L]). \\
\text{member}(X, [X|L]) :- \text{member}(X,L). \\
\text{intersection}(L,M) :- \text{member}(X,L), \text{member}(X,M).
\]
In the definition of intersection, the first call member(X,L) is active and binds X successively to each element of L, while member(X,M) is passive, simply testing if the value bound to X is an element of M. This is the 'generate and test' principle, whose abuse ensured the success of Prolog as a declarative language, as well as its undeserved reputation for inefficiency.

Unlike the definitions using call by pattern matching in functional programming, a logic program can contain several clauses whose heads unify with a literal. In certain situations, this non-determinism is inevitable. A typical example is that of the comparison of two binary trees (with leaves nil and nodes tagged by atoms) up to reflections:

iso(nil,nil).
iso(t(L1,X,R1),t(L2,X,R2)) :- iso(L1,R1), iso(L2,R2).
iso(t(L1,X,R1),t(L2,X,R2)) :- iso(L1,R2), iso(L2,R1).

It is impossible to determine which clause leads to a solution, so long as the solution is not known! The search tree must therefore be constructed, even though only one solution is sought.

There are other circumstances where the construction of a tree can be avoided.

Definition 7.6 A predicate p is called

- **locally deterministic** if for every literal L, there is at most one rule head defining p which unifies with L;
- **deterministic** if every search tree with root p is linear.

A query about a deterministic predicate can be executed more efficiently. The preprocessor nab of NU-PROLOG is capable of adding when declarations to force a predicate to be locally deterministic, by suspending the selection of an insufficiently instantiated literal which would unify with several clause heads.

**Example** The following program computes the sum of (numeric) tags of the preceding binary trees, after having transformed them into lists by flatten:

```
sum_tree(T,N) :- flatten(T,L); sum_list(L,N).
?- flatten(t(A, B, C), D) when A.
flatten(t(t(XX,XU,XY),U,Y),L) :- flatten(t(XX,XU,t(XY,U,Y)),L).
flatten(t(nil,U,Y),[U|L]) :- flatten(Y,L).
flatten(nil,[]).
sum_list(L,S) :- sum_list1(L,0,S).
?- sum_list1(A, B, C) when A.
sum_list1([],S,S).
sum_list1([N|L],A,S) :- plus(A,N,AN), sum_list1(L,AN,S).
```

Note that the ?- is part of the program. The when declaration over sum_list1 requires the instantiation of the first argument, which makes it locally deterministic. Since plus is deterministic, sum_list1 is as well, hence sum_list too. flatten
is, due to its when declaration. So sum_tree is deterministic since it is locally
deterministic (only one clause) and it calls two deterministic clauses.

Concurrent logic programming
Concurrent Prologs introduce two new structures: guards and variable protection.
Unification is subject to restriction in order to fill two new necessary functions,

- synchronization, and
- communication

between the different processes. The flow of data is no longer bidirectional. The
differences between the existing dialects CP (Concurrent Prolog), Flat CP, GHC
(Guarded Horn Clauses), Flat GHC, PARLOG, etc., are mostly about the different
restrictions imposed on unification.

7.3.5 Evaluating predicates

It was stated right from the beginning that terms are not evaluated by the inter-
preter, unlike in functional programming. A real programming language must
implement numeric computations efficiently and so it is out of the question to im-
plement numbers as terms. Every Prolog contains an applicative subset, which
evaluates arithmetic expressions, without resolution, using the underlying machine
arithmetic. This applicative subset is interfaced with resolution by the ‘evaluating
predicates’, such as the order (=<, <, ...) and assignment (is) predicates.

Example The literal \( M \leq N \) succeeds if \( M \) and \( N \) are closed arithmetic ex-
pressions, and if their evaluation produces numbers \( m \) and \( n \) such that \( m \leq n \). If \( M \)
or \( N \) is not an arithmetic expression, \( M \leq N \) fails. Finally, if they are arithmetic
expressions, but one of the terms contains variables, the evaluation does not take
place. Standard Prolog interpreters then terminate by triggering an error. Hence

\[
?- X = 1, X + 1 =< 3.
\]

\[ \checkmark X = 1 \]

succeeds, while the inverse order of the literals provokes an error:

\[
?- X + 1 =< 3, X = 1.
\]

\[ \checkmark Error \text{ in arithmetic expression: not a number} \]

\[
(1) 1 \text{ Fail: } _0+1=<3
\]

\[ \text{no} \]

This behavior calls into question the switching property of pure logic program-
ning: resolution is subordinate to the applicative subset, and unification appears
to be another form of assignment, with the problems of uninitialized variables in sequential programming!

The only solution is to suspend the evaluation of non-closed arithmetic expressions until the variables are bound to numeric values. So the left literal selection rule is not used, even though the basis is still SLD-resolution (switching lemma).

The literal could be moved to the right of the query to ensure that it is sufficiently instantiated at the end. However, tests should be done as soon as possible to reduce the search space. A compromise consists of suspending the evaluation of the literal and waking it up as soon as its variables are completely instantiated by the resolution of the other literals of the query. It is this mechanism of coroutine which is implemented in NU-Prolog, where there is a satisfactory behavior:

?- X + 1 =< 3, X = 1.
\[ X = 1 \]

The predicate plus of NU-Prolog is suspended so long as one of its arguments is a variable. If both of its arguments are integers, then the third can be computed.

These evaluating predicates only play a passive role: they check without producing values themselves (the value is produced by the evaluator, the predicate tests it). The coroutine mechanism does not suffice to guarantee completeness, since the two literals of each of the queries

?- X =< 3, X >= 5.
?- X =< 3, 1 =< X.

are suspended indefinitely, even though the first query should fail logically and the second should succeed. One can envisage active predicates which restrict the search space of solutions: the answer to the first query should be \( X \in \emptyset \), which is equivalent to a failure, without examining other literals, and the answer to the second should be \( X \in [1,3] \). To have active predicates, SLD-resolution must be extended to the resolution of equations over \( T[X] \), by a mechanism of resolution and constraint propagation. The only constraint that Prolog handles correctly is equality between terms. The relations \( \neq \) over \( T_\Sigma \), \( = \) and \( \leq \) over numbers are not treated in a 'logical' manner.

The only applicative part of Prolog, limited to the evaluation of arithmetic expression and to a few other elementary data types, is not sufficiently powerful as a programming language. It has neither the functional \( \lambda \), nor the sequential iteration. It must therefore be combined with SLD-resolution to simulate these constructions. So one writes programs such as:

\[
\begin{align*}
\text{f}(0,1). \\
\text{f}(N,P) & :- N > 0, N1 \text{ is } N - 1, \text{f}(N1,P1), P \text{ is } N \ast P1.
\end{align*}
\]

The literal \( M \) is \( N \) succeeds if the evaluation of \( N \) is allowed and produces a value \( n \), and if \( M \) unifies with \( n \). Normally, this is restricted to arithmetic expressions, so \( n \) is a number, but other types of evaluation could be possible, such as \([X,Y] \text{ is } [1+1,2\ast3]\). Here is the resolution of the query \( ?- \text{f}(3,P) \), omitting the
test \( N > 0 \):

\[
1 \cdot f(3, P) \quad \rightarrow \quad 1 \cdot N_1 \text{ is } 3 - 1, f(N_1, P_1), P \text{ is } 3 \cdot P_1 \\
\rightarrow \quad 1 \cdot f(2, P_1), P \text{ is } 3 \cdot P_1 \\
\star \quad 1 \cdot f(0, P_3), P_2 \text{ is } 1 \cdot P_3, P_1 \text{ is } 2 \cdot P_2, P \text{ is } 3 \cdot P_1 \\
\rightarrow \quad 1 \cdot P_2 \text{ is } 1 \cdot 1, P_1 \text{ is } 2 \cdot P_2, P \text{ is } 3 \cdot P_1 \\
\star \quad \left[\begin{array}{c} P \\ 6 \end{array}\right], \top.
\]

7.4 Negation

SLD-resolution is defined in intuitionistic logic, and can ever be defined in minimal logic. Since the latter has no rule for negation, the logical interpretation of programs offers no clue for the treatment of negation. How can one, with the same programs, state, prove and use negative facts. And what kind of negation? That of minimal logic, where \( \neg \varphi \) is an abbreviation for \( \varphi \Rightarrow \bot \), the constant connective \( \bot \) being defined by no rule? that of intuitionistic logic, which remains constructive? or that of classical logic?

7.4.1 Negation as failure

From the model point of view, the answer is easy: since each definite program \( P \) has a model \( \models \) in which all the atoms are true, it is impossible to deduce \( P \models \neg A \), whatever the concept of deduction \( \models \) which satisfies the soundness property. To avoid this problem, two paths are possible:

- extend \( P \) into a stronger theory \( \overset{\cdot}{P} \), leaving the framework of definite clauses, to obtain \( \overset{\cdot}{P} \models \neg A \), where \( \models \) is as usual;
- replace \( \models \) by a stronger relation \( \models \), leaving the usual logic, to obtain \( P \models \neg A \).

Before choosing one of these paths, consider what happens in practice.

The standard implementation of negation consists of defining a predicate \( \text{not} \) as follows (using \( X \) as a literal metavariable):

\[
\text{not}(X) :- X, !, \text{fail}.
\]

\[
\text{not}(\_).
\]

Given a query \( \text{not}(R) \), if the query \( R \) succeeds, \( \text{not}(R) \) fails, since \( \text{fail} \) always fails and the ! prevents the activation of \( \text{not}(\_). \) If \( R \) fails, then \( \text{not}(R) \) succeeds with the second clause. Negation is interpreted as \textit{failure}. However, logically incomprehensible programs can result: although the query

\[
?- X = a, \text{not}(X = b).
\]

\[
\not~X = a
\]
succeeds as one would expect,

\(?- \text{not}(X = b), \ X = a.\)

fails after having unified X with b, traversed the \(!\) and encountered the \text{fail} (literal \(X = a\) is not even examined). To avoid this problem, \text{not} should only be applied to queries without variables or to queries in which the variables have all been instantiated beforehand, a condition which is not statically verifiable.

This interpretation of negation was formalized in database theory, where the \textit{closed world assumption} states that any fact not explicitly affirmed is considered to be false. It is tempting to extend this assumption into an inference rule allowing the deduction of \(\neg \varphi\) if \(\varphi\) is not deducible from the data:

\[
\text{(CW)}: \quad \frac{\Gamma \vdash \varphi}{\Gamma \vdash \neg \varphi}.
\]

Up to now, for each deductive relation \(\vdash\) (natural deduction, equational logic, SLD-resolution), inference rules over the formulas have allowed the construction of derivations, which are concrete objects, and \(\Gamma \vdash \varphi\) stated the existence of a derivation of \(\varphi\) with assumptions in \(\Gamma\). The relation was local to \(\Gamma\). Here, rule \(\text{(CW)}\) cannot be used to construct the derivations of formulas. It modifies \(\vdash\), which loses its local qualities, since \(\Gamma, \varphi \vdash \neg \varphi\) should not be inferrable, even if \(\Gamma \vdash \neg \varphi\) were obtained. Furthermore, as soon as a logical system is powerful enough, \(\Gamma \not\vdash \varphi\) is not even semi-decidable. An inference rule whose premises are not semi-decidable is of no \textit{operational} interest.

This solution, which replaces classical deduction by a non-local deductive relation, can also be stated by extending the program: to prove \(\neg A\), just add \(\neg A\) to \(P\)! Replace \(P\) by

\[
\bar{P} \equiv P \cup \{-A; A \in \text{At}_\Sigma \text{ and } P \not\vdash A\}.
\]

For every closed atom \(A\), then either \(\bar{P} \vdash A\) or \(\bar{P} \vdash \neg A\). However, the theory \(\bar{P}\) is not recursively enumerable in general (see Chapter 7), which means it has no \textit{declarative} value. If a recursive \(\bar{P}\) is wanted, preferably finite, then there will be negations \(\neg A\) such that \(P \not\vdash A\) cannot be proven. The declarative reading gives three values: some atoms are true (\(P \vdash A\)), others are false (one can prove \(\neg A\)), the others are unknown. The latter class should be reduced as much as possible.

7.4.2 Decidable refutation of atoms

Let \(P\) be a definite program. It is not possible to prove \(\neg A\) for all \(A \notin T_P^{1}\omega\), but there is a class of atoms which can be refuted effectively: those for which there is a finitely failed search tree, i.e., a finite search tree with no solution. This class is easily found using \(T_P\), by iterating from \(\text{At}_\Sigma\) instead of \(\emptyset\): \(T_P^{\emptyset} = \text{At}_\Sigma\), \(T_P^{1} = T_P(T_P^{0}), \ldots\). So, if \(A\) admits a search tree without solution of depth 1 (root with no child), then \(A\) is not an instance of a clause head of \(P\): \(A \notin T_P^{1}\).
More generally, if $A$ admits a tree without solution of depth $h$, then $A \notin T^h_P$. Hence, if $A$ has a finitely failed search tree, then $A \notin T^{1\omega}_P$. This result is easily proven in [4].

The set $T^{1\omega}_P$ has already been presented: if $A$ allows an infinite fair derivation, then $A \in T^{1\omega}_P$ (proposition 7.8). Hence, if $A \notin T^{1\omega}_P$, every fair search tree is necessarily finite and without solution, since $T^{1\omega}_P \subseteq T^1_P$. The following result has been proven.

**Proposition 7.16** The following properties are equivalent, for $A \in \text{At}_\Sigma$ and a definite program $P$:

1. $A \notin T^{1\omega}_P$.
2. $A$ has a finitely failed search tree.
3. Every fair search tree for $A$ is finitely failed.

From an operational view, one must (and can) choose a *fair* selection rule $S$. Given $A \in \text{At}_\Sigma$, the construction determined by $S$ of the resolution tree of $A$ yields one of the two following results, or yields nothing at all:

- **true**, as soon as a solution is found (be the tree finite or not), i.e., $A \in T^{1\omega}_P$;
- **fail**, if the tree is finitely failed, i.e., $A \notin T^{1\omega}_P$.

The true/fail pair of NU-Prolog is preferable to yes/no, which makes the two answers appear to be symmetrical. The intermediate case is $A \in T^{1\omega}_P \setminus T^{1\omega}_P$, which corresponds to an infinite tree without solution: no interpreter can give an answer in such a case. In practice, a response such as stack overflow can correspond to any of the three cases, but it is quite probable that an infinite tree is being constructed which would exclude $A \notin T^{1\omega}_P$.

Note that fail is not sensitive to the search strategy, since it can only be generated for finite trees. This is not the case for true. With an unfair rule, which is standard, there can also be non-termination when $A \notin T^{1\omega}_P$: stack overflow no longer gives any indication.

### 7.4.3 Completed program

The problem is to extend the program $P$ into a ‘recursively axiomatizable’ theory $\bar{P}$ such that $\bar{P} \vdash \neg A$ if $A$ is effectively refutable, i.e., $A \notin T^{1\omega}_P$. Clark introduced program completion to achieve this goal.

**Clause completion**

Let $C$ be a clause $p(t_1, \ldots, t_r) : \neg A_1, \ldots, A_p$ with variables $y_1, \ldots, y_m$. Its homogeneous form is clause $C^h$,

$$p(x_1, \ldots, x_r) : \neg x_1 = t_1, \ldots, x_r = t_r, A_1, \ldots, A_p,$$
written \( p(\bar{x}) :\neg C^h \), where \( x_1, \ldots, x_r \) are new variables. Variables \( y_1, \ldots, y_m \) all appear on the right-hand side, where they are quantified existentially. So this clause can be written as the formula
\[
p(x_1, \ldots, x_r) \leftarrow \exists y(y_1 = t_1 \land \ldots \land y_r = t_r \land A_1 \land \ldots \land A_y).
\]
A clause and its homogeneous form are equivalent modulo the equality axioms: \( Ax \vdash_C C \iff C^h \). Consider the homogeneous forms \( p(\bar{x}) :\neg C^h \) of all of the clauses defining \( p \). In the following discussion, variables \( \bar{x} \) are fixed and distinct from all of the variables in clauses. The homogeneous definition of \( p \) is the formula
\[
\hom(p) \equiv \forall \bar{x} p(\bar{x}) \leftarrow \bigvee C^h_i,
\]
which is equivalent to the conjunction of the clauses defining \( p \), since \( Ax \vdash_M (p(\bar{x}) \leftarrow \bigvee_i C^h_i) \iff \land_i (p(\bar{x}) \leftarrow C^h_i) \) and \( Ax \vdash_M C_i \iff C_i^h \) for each \( i \). Hence, for the whole program
\[
Ax \vdash_M P \iff \bigwedge p \hom(p).
\]
This is a program transformation, analogous to the transformation of call by pattern matching in functional programming into a case structure inside the body of the defined function. This homogeneous form makes the non-determinism of pure logic programming explicit through a disjunction: \( C_i^h \) is chosen ‘randomly’ to solve a literal of predicate \( p \). If these homogeneous forms are considered as programs, the SLD-resolution must be restricted so that it does not select any equations: \( 1 \cdot R_0 \not\vdash 1 \cdot E \) is obtained, where \( E \) is a sequence of equations, no variable of \( R_0 \) having yet been bound.

**Example** The homogeneous form of the predicate \( p \) defined by
\[
\begin{align*}
p(U, 0, U) . \\
p(U, s(V), s(W)) & :\neg p(U, V, W) .
\end{align*}
\]
is
\[
\begin{align*}
p(X, Y, Z) & :\neg X=U, Y=0, Z=U . \\
p(X, Y, Z) & :\neg X=U, Y=s(V), Z=s(W), p(U, V, W) .
\end{align*}
\]
or, more simply, by using the or ‘\( ; \)’ of Prolog:
\[
\begin{align*}
p(X, Y, Z) & :\neg (X=U, Y=0, Z=U) \\
& ; (X=U, Y=s(V), Z=s(W), p(U, V, W)) .
\end{align*}
\]

If symbol \( p \) appears in the program, without any clause defining it, the disjunction \( \bigvee_i C^h_i \) reduces the neutral element of \( \forall \) to \( \bot \). Since \( p \leftarrow \bot \) is an intuitionistic tautology, one can define \( \hom(p) \equiv \top \): for a definite program, an undefined fact, which is therefore not provable using SLD-resolution, is considered to be true. This is exactly the opposite of negation as failure! Instead of \( p \leftarrow \bot, \) or \( \top \), it is \( p \Rightarrow \bot, \) or \( \neg p \), which represents the absence of information about \( p \) under negation.
More generally, since all that is known about \( p \) is in one formula, the definition is 'closed' by replacing \( \iff(p \text{ if } \ldots) \) by \( \iff(p \text{ if and only if } \ldots) \). The completed definition of the predicate \( p \) is

\[
\text{comp}(p) \equiv \forall \overline{x}(p(\overline{x}) \iff \exists \overline{y}(C_1^{h-} \lor \ldots \lor C_n^{h-})).
\]

**Example (cont.)**

\[
\forall(x, y, z) \left( p(x, y, z) \iff \exists u(x = u \land y = 0 \land z = u) \lor \exists uv(x = u \land y = s(v) \land z = s(w) \land p(u, v, w)) \right).
\]

\( \square \)

In the case where \( p \) has no definition in the program, then \( n = 0 \), which leads to \( \forall \overline{x}(p(\overline{x}) \iff \bot) \), or

\[
\text{comp}(p) \equiv \forall \overline{x} \neg p(\overline{x}).
\]

which is equivalent. Since \( \text{comp}(p) \) is the conjunction of \( \text{hom}(p) \) and of the symmetric (in \( \iff \) / \( \rightarrow \)) formula, it is clear that \( \text{comp}(p) \Rightarrow \text{hom}(p) \), hence

\[
A_x \vdash_M \bigwedge_p \text{comp}(p) \Rightarrow P.
\]

The formula \( \forall \overline{x}(p(\overline{x}) \Rightarrow \exists \overline{y}(C_1^{h-} \lor \ldots \lor C_n^{h-})) \) is the homogeneous converse definition of \( p \).

**Fixpoints of \( T_P \)**

**Proposition 7.17** A Herbrand algebra \( H \) is a model of \( \bigwedge_p \text{comp}(p) \) if and only if \( H \) is a fixpoint of \( T_P \).

**Proof** Since \( H \models P \) if and only if \( T_P(H) \subseteq H \), it suffices to show that \( H \) satisfies the homogeneous converse definitions of \( P \) if and only if \( T_P(H) \supseteq H \).

\( T_P(H) \supseteq H \) means that \( H \) is an algebra supported by \( P \), i.e., for every \( A \in H \), there exists a closed instantiation \( \theta C \) of a clause \( C \) of \( P \) such that \( A \equiv \theta C^+ \) and \( H \models \theta C^- \).

\( H \models \forall \overline{x}(p(\overline{x}) \Rightarrow \forall i \exists \overline{y}(\overline{x} = \overline{t} \land C_i^-)) \) means that for every \( M_1, \ldots, M_r \in T_E \), if \( H \models p(M) \), then there exists an \( i \) such that \( H \models \exists \overline{y}(M = \overline{t} \land \theta C_i^-) \). Hence there exists an \( i \) and a closed substitution \( \theta \) such that \( H \models M = \theta \overline{t} \land \theta C_i^- \).

It is clear that the two properties are equivalent. \( \square \)

**Negation as omission and Clark axioms**

The only way to state a negative fact in a definite program is to leave it out. For example, the predecessor over the natural integers is defined by the unit clause:

\[
\text{pred}(s(X), X).
\]
which is unifiable with no literal of the form \( \text{pred}(0, y) \), and the interpreter correctly answers fail for such a query.

It is true that \( \text{comp}(\text{pred}) \equiv \forall x y (\text{pred}(x, y) \iff \exists u (x = s(u) \land y = u)) \), but \( A \models \text{comp}(\text{pred}) \Rightarrow \neg \text{pred}(0, t) \) no matter what \( t \) is, simply because there exist models with equality of \( \text{comp}(\text{pred}) \) where \( 0 \) receives the same interpretation as a successor term. For example, if \( (ss0)^A = 0^A \) in a model with equality \( A \) of \( \text{comp}(\text{pred}) \), then \( A \models \text{pred}(0, s0) \), which shows that \( \neg \text{pred}(0, s0) \) cannot be a consequence of \( \text{comp}(\text{pred}) \).

Since the completion of clauses is insufficient to express this negation as omission, we add a set of axioms, due to Clark, defining the theory of equality of terms:

- **decomposition** \( f(x_1, \ldots, x_r) = f(y_1, \ldots, y_r) \Rightarrow x_1 = y_1 \land \ldots \land x_r = y_r \), for each \( f \);
- **consistency** \( f(x_1, \ldots, x_r) \neq g(y_1, \ldots, y_s) \), if \( f \neq g \);
- **acyclicity** \( x \neq t \) for every term \( t \) different from \( x \) and containing \( x \).

The particular case of consistency axioms for constants \( c \neq c' \) if \( c \) and \( c' \) are distinct constants) is well known in the framework of databases.

The completion \( \text{comp}(P) \) of a program \( P \) is the union of the equality axioms, the Clark axioms and the completed definitions of the predicates.

**Corollary 7.18** If \( P \) is a definite program, theory \( \text{comp}(P) \) is consistent.

**Proof** \( T_P \) has at least one fixpoint (its least fixpoint \( \mu T_P = \mathcal{M}_P \)) which is a model of \( \Lambda_p \text{comp}(p) \) (proposition 7.17). Since every Herbrand algebra satisfies the Clark axioms, the fixpoint is also a model of \( \text{comp}(P) \). So \( \text{comp}(P) \) is consistent. \( \square \)

Every model with equality of the Clark axioms contains a subalgebra isomorphic to \( T_\Sigma \). It has already been shown (without the Clark axioms) that \( \Lambda_p \text{comp}(p) \models_M P \). Since every Herbrand algebra satisfies the Clark axioms, we have the following corollary.

**Corollary 7.19** \( \text{comp}(P) \models_M P \).

From proposition 7.17, we also have the following corollary.

**Corollary 7.20** The Herbrand models of \( \text{comp}(P) \) are all the Herbrand algebras between \( \mu T_P \) and \( \bar{\mu} T_P \). Hence, \( \text{comp}(P) \cup \{ A \} \) has a Herbrand model if and only if \( A \in \bar{\mu} T_P \).

**Positive consequences of \( \text{comp}(P) \)**

The completion of clauses is a conservative extension of the program, with respect to positive formulas.

**Proposition 7.21** Let \( \psi \) be a positive formula, i.e constructed over \( \land, \lor, \forall \) and \( \exists \). If \( \text{comp}(P) \models \psi \), then \( P \models \psi \).
Proof Suppose that \( \text{comp}(P) \vdash \psi \). It will be shown that \( P \Rightarrow \psi \) is a tautology. From the Herbrand–Skolem theorem, \( P \Rightarrow \psi \) is a tautology if and only if its existential skolemized function \((P \Rightarrow \psi)^e \equiv P \Rightarrow \psi^e\) is satisfied by all of the Herbrand algebras of domain \( T^\Sigma \), where \( \Sigma \) follows from \( \Sigma \) by adding Skolem symbols. Let \( \mathcal{H} \) be a Herbrand model of \( P \) of domain \( T^\Sigma \). By the definition of \( \mathcal{M}_P \), \( \mathcal{M}_P \subseteq \mathcal{H} \cap \text{At}_P \). Since \( \mathcal{M}_P \) is a fixpoint of \( T_P \), it is also a model of \( \text{comp}(P) \). By assumption, \( \text{comp}(P) \vdash \psi \), hence \( \mathcal{M}_P \models \psi \). Now \( \models \psi \Rightarrow \psi^e \), so \( \mathcal{M}_P \models \psi^e \). From exercise 10, since \( \psi^e \) is existential positive, deduce that \( \mathcal{H} \models \psi^e \) from \( \mathcal{M}_P \subseteq \mathcal{H} \) and \( \mathcal{M}_P \models \psi^e \). Formula \( P \Rightarrow \psi \) is therefore a tautology. \( \square \)

Corollary 7.22 Let \( R \) be a positive query and \( \sigma \) a substitution.

If \( \text{comp}(P) \vdash \forall(\sigma R) \), then \( P \vdash \forall(\sigma R) \).

Negative consequences of \( \text{comp}(P) \)

It is easy to show that if \( A \notin T_P^\omega \), then \( \text{comp}(A) \vdash_M \neg A \), by using the homogeneous converse definitions of \( P \) and the Clark axioms. Only the base case of the proof will be considered: \( A \notin T_P^1 \), i.e., \( A \equiv p\overline{M} \) is not an instance of a clause head. The converse definition of \( p \) is used: \( p(\overline{x}) \Rightarrow \ldots \). By instantiating \( \overline{x} \) in \( \overline{M} \), and by supposing \( A \), the disjunction \( \exists y(\overline{M} = \overline{t} \land \overline{C}^-) \lor \ldots \) follows. By assumption, the equations \( p\overline{M} = p\overline{t} \) between \( A \) and the clause heads defining \( p \) have no solution. The Clark axioms allow one to derive \( \bot \) from each equation \( \overline{M} = \overline{t} \). So \( \bot \) was derived by supposing \( A \), which proves \( \neg A \) in intuitionistic logic, and even in minimal logic, since \( \bot \)-elimination was not used.

The converse is more delicate, since a model of the theory \( \text{comp}(P) \cup \{A\} \) must be constructed when \( A \in T_P^\omega \). If one had the assumption \( A \in \mu T_P \), which is stronger since \( \mu T_P \subseteq T_P^\omega \), proposition 7.20 would show that there exists a Herbrand model of \( \text{comp}(P) \cup \{A\} \). But for \( A \in T_P^\omega \), a general model must be constructed, using the fact that there exists a successful or infinite derivation in a fair search tree of \( A \). This construction is given in [95, §16].

Theorem 7.23 If \( P \) is a definite program, and \( A \in \text{At}_P \), then

- \( \text{comp}(P) \vdash_M A \) is equivalent to \( A \in T_P^\omega \);
- \( \text{comp}(P) \vdash_M \neg A \) is equivalent to \( A \notin T_P^\omega \).

7.4.4 General programs

For a programmer, it is easier to write negation explicitly rather than through omission in definite programs. General programs and queries allow the bodies of clauses and queries to be formed of arbitrary literals.

However, the extension of the logical approach to general programs is problematic.
The intersection of Herbrand models is not always a model. There does not always exist a least Herbrand model, since the $T_P$ operator is no longer monotone. The definition of completion can be transposed to general programs, but $\text{comp}(P)$ can be inconsistent even if $P$ is consistent.

**Example** Consider the clause:

$$p : - \text{not } p.$$  

The definition of $p$ is classically equivalent to the assertion of $p$, but the theory $\text{comp}(P) \equiv p \iff \neg p$ is inconsistent. 

Normally, a query’s variables are quantified existentially. If non-closed negative atoms are allowed in queries, negation as failure can only work if universal quantifiers are introduced: the failure of a proof of $\exists x p(x)$ does not prove $\exists x (\neg p(x))$, rather $\neg (\exists x p(x))$, i.e., $\forall x (\neg p(x))$.

**SLDNF-resolution** The operational semantics of a general program is given by SLDNF-resolution (NF means negation as failure).

The same environment-query state is used, but both SLDNF-derivations and SLDNF-search trees must be defined. The nodes of a search tree are labeled by the search states. A search state is a pair $(\rho \cdot R, s)$, where $R = T$, or $R/s$ is a literal of $R$ which is an atom or the negation of a closed atom. The search tree for the solutions of an initial query $R_0 \neq T$ has as root the state $(1 \cdot R_0, s_0)$. Let $(\rho \cdot R, s)$ be a search state.

- If $R/s$ is an atom, let $C$ be a separate variant of a clause of $P$, and
  - if $R/s = C^+$ has no solution, then there is no transition tagged by $s, C$ out of $\rho \cdot R$;
  - if $R/s = C^+$ has a most general unifier $\theta$, then $\rho \cdot R \xrightarrow{s, C} \theta\rho \cdot \theta(R[s \leftarrow C^-])$.

The node $(\rho \cdot R, s)$ has a descendent with state $\theta\rho \cdot \theta(R[s \leftarrow C^-])$ for each such $C$. It is a failure node if it has no descendents.

- If $R/s$ is the negation $\neg B$ of a closed atom $B$, and
  - if $B$ has a solution, then $(\rho \cdot R, s)$ is a failure node, which has no descendents;
  - if $B$ has finitely failed, then $\rho \cdot R \xrightarrow{s, C} \rho \cdot R \setminus s$, and the node $(\rho \cdot R, s)$ has a unique descendent, with state $\rho \cdot R \setminus s$.

Clark gave a soundness theorem for SLDNF-resolution:

**Theorem 7.24** Let $P$ be a general program and $R$ a general query:

- if $R$ fails, then $\text{comp}(P) \vdash_M \neg \exists(R)$;
- if $\rho$ is an answer to $R$, then $\text{comp}(P) \vdash_M \forall(\rho R)$.
SLDNF-resolution proves theorems in minimal logic, as does SLD-resolution.

Prolog interpreters, such as C-Prolog, implement negation as failure, but rarely safely since they can select a non-closed negative literal: these interpreters may give incorrect answers. Suppose that the closure condition on negative literals were left out, and consider program \( p(a) \) and query not \( p(X) \). Since there is the solution \( 1 \cdot p(x) \rightarrow ^x_1 \cdot T \), not \( p(X) \) fails and an erroneous application of the soundness theorem leads to \( \text{comp}(P) \vdash \neg \exists (\neg p(x)) \), hence \( \text{comp}(P) \vdash \forall x p(x) \), which is incorrect.

NU-Prolog is a safe implementation of negation as failure, which suspends the selection of a negative literal if it is not closed, selects another literal, and awakens the suspended literal as soon as its variables are bound to closed terms. It is possible that all of the literals of a query are negative and not closed: in that case no literal may be selected and the search flounders. Hence \( \neg \) not \( p(X) \) flounders immediately. But whether a query flounders is an undecidable problem. Because of these problems, the extension of Prolog to general programs cannot any longer be considered as an automatic theorem prover.

**Examples**

- With the program \( p(X) :\neg q(X), \) query \( p(a) \) succeeds, but \( p(X) \) flounders: hence \( p(a) \) is provable, but not \( \exists x p(x) \).
- By adding \( r(a) \) to the preceding program, query \( r(X), p(X) \) succeeds but \( p(X) \) flounders: \( \exists x (p(x) \land q(x)) \) is provable, but not, \( \exists x p(x) \).
- With the program \( p :\neg q, p \) succeeds and \( q \) fails, while \( q \) succeeds and \( p \) fails with the equivalent program \( q :\neg p \).

As was stated previously, negation as failure requires that the free variables of negative sign be quantified universally: the literal not \( p(X) \) means \( \forall x (\neg p(x)) \).

NU-Prolog has explicit quantifiers some and all: by default not \( p(X) \) is equivalent to all \( X \) not \( p(X) \) and to not \( (\text{some} X p(X)) \), while the query \( \neg (\exists \bar{x} R_0) \) in the body of a rule, where \( R_0 \) is a query and \( \bar{x} \) are variables (not necessarily all those of \( R_0 \)).

**Example** The disjoint/2 predicate, which tests if two lists have no element in common, is defined by one of the following equivalent clauses:

\[
\text{disjoint}(X,Y) :\neg (\text{some} A (\text{member}(A,X), \text{member}(A,Y))).
\]

\[
\text{disjoint}(X,Y) :\text{all} A \neg (\text{member}(A,X), \text{member}(A,Y)).
\]

The soundness theorem shows that \( \neg (\exists \bar{x} R_0) \) is erasable by constructing a finitely failed tree for \( R_0 \) if \( \neg (\exists \bar{x} R_0) \) is closed, i.e., all of the variables in \( R_0 \) are bounded by the \( \exists \). A coroutine mechanism allows the suspension of the selection of \( \neg (\exists \bar{x} R_0) \) so long as it contains free variables. Even better, the search can begin even if there
are free variables, and it will be suspended before broadcasting a binding relating to these variables. If $R_0$ succeeds, then $\neg(\exists x R_0)$ fails. If $R_0$ has no solution, then $\neg(\exists x R_0)$ is deleted.

In addition to the logical weaknesses of negation as failure, its major computing defect is that it computes nothing: it is a test which checks the non-existence of a solution without producing any solution. This defect has already been mentioned with respect to evaluating predicates. With a program $p(a)$, it would be useful if a literal $\neg p(x)$ were deleted and replaced by the constraint $x \neq a$. This idea is the basis for constraint programming.

### 7.5 Equality and resolution

The two computing paradigms presented in this book, reduction and resolution, are based on quite distinct principles. Several proposals have been made to unify functional and logic programming. These two paradigms have one point in common: equations. SLD-resolution is based on the resolution of equations in $T[X]$, while functional programs can be seen as rewrite rules (oriented equations) and equality is axiomatized by equations (first-order in combinatory logic).

An interesting and useful extension would be to incorporate an equational theory $E$, for example to state a recursive function definition, properties of a data structure or a domain of computation, such as Booleans, rationals or multi-sets. Although the treatment of equality poses no theoretical difficulties in logic, the complexity of automatic proof methods constitutes a real obstacle to their transformation into effective computations. In a system based on the resolution principle, the equality axioms multiply the number of equations to be solved.

Although it is more natural to define functions in an equational manner (Prolog appeared fifteen years after Lisp), Prolog programmers have become used to writing non-equational definitions. This shows that equational definitions are not irreplaceable, and that it is possible to write efficient logic programs by renouncing the ability to express equality directly.

#### 7.5.1 Reduction by resolution

A first method consists of adding to the program axioms of $E$ which are simple facts $M = N$, and the equality axioms in definite clause form for each of the symbols used. Here is the equational definition of addition:

- $e(X+0, X)$.
- $e(X+s(Y), s(X+Y))$.
- $e(X, X)$.
- $e(X, Y) :- e(Y, X)$.
- $e(X, Z) :- e(X, Y), e(Y, Z)$. 
Resolution

\[ e(s(M), s(N)) :\text{\texttt{-}} e(M, N). \]
\[ e(M1+N1, M2+N2) :\text{\texttt{-}} e(M1, M2), e(N1, N2). \]

\(?- e(s(s(0))+s(s(0)), Y). \]

The interpreter supplies a great number of solutions of little interest, and in this case does not give the 'good' solution, which is \( Y = s(s(s(s(0)))) \):

\[ \sim Y = s(s(s(0)) + s(0)); \]
\[ Y = s(s(0)) + s(s(0)); \]
\[ Y = (s(s(s(0)) + s(s(0))) + 0; \]
\[ Y = s(s(0)) + s(s(0)); \quad \text{etc} \ldots \]

As can be checked, this program is very sensitive to the order of clauses: the search tree has infinite branches and far too many stupid and repeated solutions. It correctly defines an equality (congruence over terms, stable under substitutions), but ignores the existence of a canonical rewriting system for this equality. In particular the interpreter is incapable of recognizing the normal form of a term, which is the only interesting response. Yet this idea is easily defined:

\[ \text{normal}(0). \]
\[ \text{normal}(s(M)) :\text{\texttt{-}} \text{normal}(M). \]

It is then possible to state the rewriting rules with an execution order using the following logic program, where the predicates eq1 and eq2 both state the equality:

\[ \text{eq1}(M, M) :\text{\texttt{-}} \text{normal}(M). \]
\[ \text{eq1}(M, X) :\text{\texttt{-}} \text{not normal}(M), \text{eq2}(M, Y), \text{eq1}(Y, X). \]
\[ \text{eq2}(X+0, X). \]
\[ \text{eq2}(X+s(Y), s(X+Y)). \]
\[ \text{eq2}(s(M), s(N)) :\text{\texttt{-}} \text{eq1}(M, N). \]
\[ \text{eq2}(M1+N1, M2+N2) :\text{\texttt{-}} \text{eq1}(M1, M2), \text{eq1}(N1, N2). \]

\(?- \text{eq1}(s(s(0))+s(s(0)), X). \]

\[ \sim X = s(s(s(s(0)))) ; \]

The role of the transitivity rule is to introduce a new variable for each reduction step. Doing this avoids returning the results of intermediate steps. The first rule is only applied at the end, when \( M \) is normal. At each step of the reduction, the selected literal is of the form \( \text{eq2}(M, Y) \), where \( M \) is a non-normal term and \( Y \) is free. If \( M \) is of the form \( \theta P \) for a rule \( P \rightarrow Q \), then the literal is deleted, \( Y \) receiving as value \( \theta Q \). The following selected literal is \( \text{eq1}(\theta Q, X) \). Otherwise, \( M \) is decomposed by the other rules for \( \text{eq2} \). The first answer obtained is the required normal form.
7.5.2 Equational programs and unification

SLD-resolution eliminates equations from programs, but incorporates them into the computing mechanism. They clearly appear in the homogeneous form of programs which introduce equations in queries. These homogeneous forms can be written using the $==$ symbol instead of $=$, which will be kept for writing the systems of equations to be unified.

Resolution (in the ordinary sense) of equations by unification can be interpreted at the level of homogeneous programs by adding the unit clause $V == V$ to the program. The unification problem $(M == N) = (V == V)$ is transformed into the equivalent problem $M = V, N = V$, then to $M = N, V = M$. The natural idea is to extend this interpretation by adding a theory (over the symbol $==$) more specific than $V == V$ to the program.

Another idea consists of not introducing equations in the programs, but to solve the equations in domains other than the algebra of terms, for example, by unification modulo $\mathcal{E}$. The difficulties are handed to unification in the cases where there is no most general unifier: the don't know non-determinism is introduced even before the choice of clause: a search state should have as many children as there are 'minimal' solutions for each unification problem, and minimal solutions do not always exist.

7.5.3 Narrowing

Narrowing is a transformation which has aspects of rewriting and resolution. While resolution solves queries using a logic program, narrowing solves equations using a functional program, which is a rewriting system. The narrowing of a term $M$ consists of applying a smaller substitution to $M$, making $M$ reducible by a rule, and of applying that rule. The narrowing of a closed term corresponds to rewriting.

The definitions are formally analogous to SLD-resolution. A narrowing state is a pair $\rho \cdot M$, where $\rho$ is a logical environment, $M$ a term, and $\text{dom}(\rho) \cap \text{var}(M) = \emptyset$. Let $\mathcal{R}$ be a rewriting system. A narrowing step is defined as follows: let $\rho \cdot M$ be a state, and $P \rightarrow Q$ a rule such that $P$ is superposable on $M$ at non-variable occurrence $u$, with most general unifier $\theta$. Then

$$\rho \cdot M \xrightarrow{u,P\rightarrow Q} \theta\rho \cdot \theta(M[u \leftarrow Q]).$$

As in SLD-resolution, there are concepts of derivation, of successful derivation $1 \cdot M \xrightarrow{\ast} \rho \cdot \top$ and of result $\rho|\text{var} M$.

This mechanism will be used to solve equations. In this case the terms are always of the form $M == N$.

**Example** Given the rewriting system $x+0 \rightarrow x$, $x+s(y) \rightarrow s(x+y)$, let $0+x = s(0)$ be an equation to be solved. Here are two examples of narrowing, one failed, one
successful:

\[ 1 \cdot 0 + x \equiv s(0) \quad \Rightarrow \quad 1 \cdot 0 \equiv s(0), \]

\[ 1 \cdot 0 + x \equiv s(0) \quad \Rightarrow \quad \left[ \begin{array}{c} x \\ s(y_1) \end{array} \right] \cdot s(0 + y_1) \equiv s(0), \]

\[ \Rightarrow \quad \left[ \begin{array}{c} x \\ s(0) \end{array} \right] \cdot s(0) \equiv s(0). \]

The soundness and completeness properties have the following form:

- if \( 1 \cdot M \equiv N \xrightarrow{\rho} \rho \cdot T \), then \( \mathcal{E} \vdash \rho(M) = \rho(N) \);
- if \( \mathcal{E} \vdash \sigma(M) = \sigma(N) \), there exists a derivation \( 1 \cdot M \equiv N \xrightarrow{\rho} \rho \cdot T \) such that \( \sigma \geq \rho \).

These properties are satisfied if one has a canonical rewriting system for \( \mathcal{E} \).

### 7.5.4 Knuth–Bendix completion

Narrowing is a particular case of completion. In fact, the narrowing state \( \rho \cdot M \) can be interpreted as a rule \( M \rightarrow \text{res}(\rho(x_1), \ldots, \rho(x_n)) \), where \( \{x_1, \ldots, x_n\} \) are the variables of the initial query, and \( \text{res} \) is a new symbol.

The superposition of \( P \rightarrow Q \) on \( M \equiv N \rightarrow \text{res}(\rho(x_1), \ldots, \rho(x_n)) \) at occurrence \( u, \theta \) being the most general unifier of \( (M \equiv N)/u \) and \( \theta(P) \), gives the equation

\[ \theta((M \equiv N)/u \leftarrow Q) = \theta(\text{res}(V_1, \ldots, V_n)), \]

i.e., \( \text{res}(\theta(\rho(x_1), \ldots, \rho(x_n))) \). It follows that the narrowing of \( \rho \cdot M \) by the rule \( P \rightarrow Q \) is equivalent to the computation of a critical pair due to a superposition of \( P \rightarrow Q \) on \( M \equiv N \rightarrow \text{res}(\rho(x_1), \ldots, \rho(x_n)) \).

Critical pairs are computed by superposition of the 'programs' on a query' by considering the rules \( P \rightarrow Q \) of the system \( \mathcal{R} \) as the program, and the \( M \equiv N \rightarrow \text{res}(\bar{V}) \) as the queries. No superposition between rules of the program, nor between query and rule, nor between successive queries has been computed. The rewrite rule \( u \equiv u \rightarrow T \) is added in order to halt the computation with a result.

**Example**

\[ 0 + x \equiv s(0) \rightarrow \text{res}(x) \quad \text{superposed by 2} \]

\[ \rightarrow s(0 + y_1) \equiv s(0) \rightarrow \text{res}(s(y_1)) \quad \text{reduced} \]

\[ \rightarrow 0 + y_1 \equiv 0 \rightarrow \text{res}(s(y_1)) \quad \text{superposed by 1} \]

\[ \rightarrow 0 \equiv 0 \rightarrow \text{res}(s(0)) \quad \text{superposed by } u \equiv u \rightarrow T \]

\[ \rightarrow \text{res}(s(0)) \rightarrow T. \]

\[ \square \]
While Knuth–Bendix completion is, like Robinson’s resolution, a method of automatic theorem-proving, narrowing is, as is SLD-resolution, a method of computation.

Exercises

1. Define two unary predicates for the mutually recursive types of trees and forests.

2.* Write a quicksort program using D-lists.

3. Define a D-tree structure analogous to D-lists, along with a concatenation program.

4. Show that if \( \mathcal{H} \) and \( \mathcal{H}' \) are two Herbrand algebras, there exists a morphism of algebras from \( \mathcal{H} \) to \( \mathcal{H}' \) if and only if \( B(\mathcal{H}) \subseteq B(\mathcal{H}') \), in which case the morphism is necessarily the identity.

5. Let \( P \) be the program
   \[
   p1(f(X)) :- p1(X).
   \]
   \[
   p2(a) :- p1(X).
   \]
   \[
   p2(f(X)) :- p2(X).
   \]
   Compute \( T_P^{\omega} \), \( T_P^{\lambda+1} \), \( T_P^{\lambda} \), \( T_P^{\omega+n} \) and \( T_P^{\omega+k} \). Write a program \( P_k \), for \( k \in \mathbb{N} \), such that \( \mu T_{P_k} = T_P^{\omega+k} \) (the ordinal \( \omega k \) represents \( k \) copies of \( \mathbb{N} \) placed end to end).

6.* Let \( \Phi_E : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*) \) be the operator associated with a signature \( \Sigma \) (Chapter 1). Show that \( \Phi_{E}^{\omega} = \Phi_{E}^{\omega} \).

7.* Let \( P \) be a definite program such that \( \text{var}(C^-) \subseteq \text{var}(C^+) \) for every clause \( C \) of \( P \). Show that \( T_P^{\omega} = \mu T_P \).

8. Show that if the signature does not contain any functional symbols of arity > 1, then \( T_P^{\omega} = \mu T_P \).

9. Prove that the selection rule which selects at each step the leftmost of the residual radicals, or if there is none, one of the introduced literals, is fair.

10. Let \( \mathcal{H} \) and \( \mathcal{H}' \) be two Herbrand algebras such that \( B(\mathcal{H}) \subseteq B(\mathcal{H}') \), and \( \varphi \) be a positive existential formula, i.e., constructed with \( \land, \lor \) and \( \exists \) as the only logical symbols. Show that for every valuation \( \xi \), \( \varphi^{\mathcal{H}}(\xi) \leq \varphi^{\mathcal{H}'}(\xi) \). Is this inequality still true for an arbitrary \( \varphi \)?

---

*Hints to the exercises labeled with a ★ can be found at the end of the book.*
11. Write a logic program to evaluate rewriting systems with lazy constructors. Treat the example of §3.3.1.

12.* Consider the class of P-formulas defined by:
   - an atomic formula is a P-formula;
   - if B and C are P-formulas and if F is an arbitrary formula, then \( B \land C, \forall x B \land F \Rightarrow B \) are P-formulas.

Show, by induction over the derivations of cut-free LJ, that if \( \Pi \) is a list of P-formulas, and if \( A \) and \( B \) are arbitrary formulas, then

1. if the sequent \( \Pi : \exists x A(x) \) is derivable, then there exists a term \( t \) such that \( \Pi : A[x := t] \) is derivable;
2. if the sequent \( \Pi : A \lor B \) is derivable, then one of the two sequents, \( \Pi : A \) or \( \Pi : B \), is derivable.

Bibliographic notes

On the art of programming in Prolog, Sterling and Shapiro’s book [141] is recommended, as is O’Keefe [115]. C-Prolog is described in [118], NU-Prolog in [149], and Prolog-II in [50, 153]. The model of activation boxes is due to L. Byrd [12]. A study of Prolog implementation, in particular of suspension mechanisms, is made in [11].

The many refinements of the resolution principle, due to Robinson [128], are developed in the standard books of Chang and Lee [13] and of Loveland [96]. The logical interpretation of Prolog is due to Kowalski [88]. The links between SLD-resolution and the sequent calculus are the focus of [47, 54]. The foundations of logic programming are given in Lloyd’s book [95], which compiles the results of several essential articles [4, 17, 155].

The handling of negation, initiated by articles of [17, 127], is analyzed by Shepherdson in [134], where theorem 7.24 is given. The introduction of equality to logic programming is studied in [40, 74, 156, 157]. The writing of a rewriting system as a logic program is due to van Emde and Yukawa [157].

There are numerous extensions to logic programming: constraint programming (Jaffar and Lassez [73], van Hentenryck [61] and van Hentenryck and Deville van-Hentenryck91), Gallier and Raatz’s Horn clauses [48], the higher-order logic programming of Miller and Nadathur [108], the extension to languages more general that Horn clauses [109, 107]. Shapiro’s survey [133] deals with concurrent logic programming and Saraswat et al. [131] study constraint concurrent programming.
Chapter 8

The computable landscape

8.1 Computability

Functions have been approached from two opposing and important views. First, mathematicians made functions into banal, particular cases of binary relations, via set theory, opening the road, most remarkably, to functional analysis. Second, the logicians of the 1930s (Herbrand, Church, Kleene, Gödel, Turing, Post, etc.) succeeded in defining the concept of computable function, as well as the mechanisms of computation. Equal in expressive power since they characterize the same computable function, the different paradigms for computation, complementing each other, allow for an analysis of the different styles of programming.

8.1.1 A few encodings

Computation is a concrete activity which manipulates very real objects, typically words, which represent integers, programs, etc. It excludes everything which is not effectively given, in particular the concepts of function or set in their contemporary mathematical sense.

Since questions of computability and decidability are untouched by dimension, encodings are used to translate $\mathbb{N}^k$ into $\mathbb{N}$. Let $\langle \cdot, \cdot \rangle : \mathbb{N}^2 \to \mathbb{N}$ be a bijective function, with inverse $(\pi_2, \pi_2) : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$. For each $k \geq 1$, there is a bijection $\langle \cdot \rangle_k : \mathbb{N}^k \to \mathbb{N}$, defined by induction over $k$ by

$$\langle m \rangle_1 = m$$
$$\langle m_1, \ldots, m_{k-1}, m_k \rangle_k = \langle \langle m_1, \ldots, m_{k-1} \rangle_{k-1}, m_k \rangle, \quad k \geq 2,$$

and using this function, a bijection $\langle \cdot \rangle^* \colon \bigcup_{k \geq 0} \mathbb{N}^k \to \mathbb{N}$:

$$\langle \rangle_0^* = 0$$
$$\langle m_1, \ldots, m_k \rangle_k^* = \langle m_1, \ldots, m_{k-1}, k-1 \rangle_{k+1} + 1 \quad k \geq 1.$$
It therefore suffices to know how to code pairs of integers. The encoding that follows has been in use ever since Cantor showed that there are as many rationals as there are integers. Enumerate \( \mathbb{N} \times \mathbb{N} \) along the lines \( x + y = p \), for increasing \( p \), and on each line, with increasing \( x \) (i.e., \( (0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), (0, 3), \ldots \)).

\[
\begin{array}{ccc}
6 \ast & 3 \ast & 7 \\
1 \ast & 4 \ast & 8 \\
0 \ast & 2 \ast & 5 \ast & 9 \\
\end{array}
\]

\( x + y = 2 \)

It is useful to define some auxiliary functions: if \( x = \langle x_1, \ldots, x_k \rangle^* \), let

\[
\begin{align*}
\|x\| &= k, \\
\text{val}(x) &= \langle x_1, \ldots, x_k \rangle_k, \\
\text{pre}(x) &= \langle x_1, \ldots, x_{k-1} \rangle^*, \\
\text{last}(x) &= x_k, \\
\text{pre}(0, x) &= x, \\
\text{pre}(i + 1, x) &= \text{pre} \langle i, x \rangle, \\
\text{last}(i, x) &= \text{last} \langle i, x \rangle, \\
\text{cons}(\langle x_0, \ldots, x_{k-1} \rangle^*, x_k) &= \langle x_0, \ldots, x_k \rangle^*.
\end{align*}
\]

This encoding allows several useful constructions. First, associate with each function \( f : \mathbb{N} \to \mathbb{N} \) a history function

\[
f^*(x) = \langle f(0), f(1), \ldots, f(x - 1) \rangle^*.
\]

A history function for \( f : \mathbb{N}^k \to \mathbb{N} \) can also be defined by distinguishing one of the \( k \) arguments of \( f \). Second, there is a bijection between unary and \( k \)-ary functions, by associating with each unary function \( f \) the function \( f_k : \mathbb{N}^k \to \mathbb{N} \), defined by \( f_k = f \circ \langle \rangle_k \).

### 8.1.2 Computable functions

The development of the concept of computable function will be presented. This concept first came to light in the 1930s and has now become well known because of computer science. A function from \( \mathbb{N}^p \) to \( \mathbb{N} \), for \( p \geq 0 \), is called a numerical function; write \( f(\vec{n}) \) instead of \( f(n_1, \ldots, n_p) \). A subset \( A \) of \( \mathbb{N}^k \) is a \( k \)-ary relation over \( \mathbb{N} \). The notations \( m \in A, A(m), 1_A(m) = 1 \) and even \( A(m) = 1 \) are
interchangeable. When \( A \) is seen as a mapping over \( \mathbb{N}^k \) with Boolean values, \( A \) is called a predicate.

**Church, Kleene and \( \lambda \)-definable functions**
Recall that an integer \( n \) is coded in the \( \lambda \)-calculus as a *Church numeral*, a ‘function iterator’ \( n \equiv \lambda f \cdot f^n x \). For example, \( 3fx =_β f(f(fx)) \).

**Definition 8.1** Let \( f : \mathbb{N}^k \to \mathbb{N} \) be a numeric function. A \( \lambda \)-term \( M_f \) \( \lambda \)-defines \( f \) if for every \( (n_1, \ldots, n_k) \in \mathbb{N}^k \), with \( f(n_1, \ldots, n_k) = m \), \( M_f n_1 \ldots n_k \Rightarrow m \). The function \( f \) is \( \lambda \)-definable if there exists a \( \lambda \)-term \( M_f \) which defines \( f \).

The successor, addition, product and exponentiation functions are \( \lambda \)-definable (see Chapter 2).

A \( \lambda \)-definable function is computable: it suffices to compute the normal form of \( M_f n_1 \ldots n_k \) by a left reduction, which is effective and deterministic. In 1932 Kleene [79], Church’s student, checked that all the numeric functions which he knew how to compute were \( \lambda \)-definable. In 1933 this led Church [14] to consider \( \lambda \)-definability as the correct concept of ‘computability’. This conviction was not shared by everyone, in particular not yet by Gödel, who had just arrived at Princeton Institute.

**Herbrand, Gödel and primitive recursive functions**
Following Dedekind (1888 [36]) and Peano, it became known that recursion could be used to define numeric functions: the recursive definitions of addition and multiplication are a subset of the Peano axioms of arithmetic. In the form of the day, now called primitive recursive, this mechanism did not allow the computation of certain obviously ‘computable’ functions (Ackermann 1928 [3]). Inspiring himself by a suggestion made by Herbrand, in 1934 Gödel [57] proposed his definition of recursive functions. This definition used three operators over functions, composition, primitive recursion and minimization, as well as a closure condition.

A set \( \mathcal{E} \) of numeric functions is

1. **closed under composition** if for every \( h, g_1, \ldots, g_p \in \mathcal{E} \), if \( f \) is defined by

   \[
   f(\bar{n}) = h(g_1(\bar{n}), \ldots, g_p(\bar{n})),
   \]

   then \( f \in \mathcal{E} \);

2. **closed under primitive recursion** if for every \( h, g \in \mathcal{E} \), if \( f \) is defined by

   \[
   f(0, \bar{n}) = g(\bar{n}),
   f(m + 1, \bar{n}) = h(f(m, \bar{n}), m, \bar{n})
   \]

   then \( f \in \mathcal{E} \);
3. closed under (total) minimization if for every \( g \in E \) such that for each \( \bar{n} \), there exists \( p \) such that \( g(\bar{n}, p) = 0 \), if \( f \) is defined by

\[
  f(\bar{n}) = \min\{ p \in N; g(\bar{n}, p) = 0 \}
\]

then \( f \in E \). Write \( f = \mu p \cdot [g(\cdot, p) = 0] \).

The base functions are:
- the constant \( 0 : N \to N \);
- the successor \( s : N \to N \); and
- the projections \( pr_k^1 : N^k \to N \), for \( 1 \leq i \leq k \).

**Definition 8.2** The primitive recursive functions are the elements of the least set of numeric functions containing the base functions and closed under composition and primitive recursion. A subset \( A \) of \( N^k \) is primitive recursive if its characteristic function \( 1_A : N^k \to \{0, 1\} \) is primitive recursive.

All the usual functions are primitive recursive (see exercise 4).

**Example** Addition is primitive recursive:

\[
  \begin{align*}
  0 + y &= y = pr_1^1(y), \\
  s(x) + y &= s(x + y) = s(pr_3^1(x + y, x, y)).
  \end{align*}
\]

It was obtained by primitive recursion and composition starting from the base functions \( pr_1^1, pr_3^1 \) and \( s \).

The primitive recursive functions are easy to program. Here is a primitive recursive operator in ML:

```ml
fun rec_prim g h =
  let
    fun f(m,n) = if m=0 then g(n) else h(f(m-1,n),m-1,n)
  in
    f
  end;
```

The definition of primitive recursive functions is easily extended from \( N \sim T_{\{0,a\}} \) to any \( \Sigma \)-algebra of terms, without needing to code the terms by integers.

A primitive recursion definition is only a particular case of a (syntactically) recursive definition. From the sequential programming point of view, the primitive recursive programs are those in which the only iteration structure is the finite loop (Pascal's for). Here is the grammar of a sequential language using this structure:

```
term ::= 0
   | variable
   | variable + 1

program ::= variable ::= term
   | program; program
   | do variable times program end
```
In a loop do $x$ times $\pi$ end, the variable $x$ is first evaluated, and if $a$ is that value, the program $\pi$ is executed $a$ times.

The set of primitive recursive functions is stable under certain operations using bounded sets.

If $g$ is primitive recursive, the function $f$ defined by

$$f(n, \bar{m}) = \sum_{p \leq n} g(p, \bar{m})$$

is primitive recursive (see exercise 6). The same is true for the bounded quantifiers $\forall \leq$ and $\exists \leq$: if $A$ is a predicate, define

$$(\exists p \leq n) A(p, \bar{m}) = \max_{p \leq n \in \mathbb{N}} [(p, \bar{m}) \in A],$$

$$(\forall p \leq n) A(p, \bar{m}) = \min_{p \leq n \in \mathbb{N}} [(p, \bar{m}) \in A].$$

If $A$ is primitive recursive, then the predicates $(\exists p \leq n) A(p, \bar{m})$ and $(\forall p \leq n) A(p, \bar{m})$ are as well. The search for an element in a bounded set is defined by the operator $\mu \leq$, or bounded minimization $\mu$:

$$\mu y \leq m. A(y) = \begin{cases} \min \{y \leq m; y \in A\} & \text{if that minimum exists} \\ m + 1 & \text{otherwise.} \end{cases}$$

This operator is computable by primitive recursion:

$$\mu y \leq 0. A(y) = \begin{cases} 0 & \text{if } 0 \notin A \\ 1 & \text{otherwise,} \end{cases}$$

$$\mu y \leq n + 1. A(y) = h(\mu y \leq n. A(y), n),$$

where $h$ is

$$h(x, n) = \begin{cases} x & \text{if } x \leq n \\ n + 1 & \text{if } x = n + 1 \text{ and } n + 1 \in A \\ n + 2 & \text{otherwise.} \end{cases}$$

The class of primitive recursive functions is also stable under the bounded maximization operator $\mu'$ (see exercise 7).

By applying this operator to the integer quotient $m \mapsto m/2$, it can be shown that the encoding of pairs $\langle \rangle$ is primitive recursive, as are the inverse projections $\pi^1$ and $\pi^2$ (note that $m, n \leq \langle m, n \rangle$). It follows that the encoding functions $\langle \rangle_k : \mathbb{N}^k \rightarrow \mathbb{N}$ are recursive, as are their inverses $\pi^k_k$.

Recall that the history $f^*$ of a function $f$ is defined by

$$f^*(x) = \langle f(0), f(1), \ldots, f(x - 1) \rangle^*.$$  

By using the definition of the encoding $\langle \rangle^*$, it is easy to see that $f^*$ is obtained starting from $f$ by the following primitive recursion:

$$f^*(0) = 0,$$

$$f^*(m + 1) = cons(f^*(m), f(m)).$$
It follows that \( f^* \) is primitive recursive if \( f \) is, and conversely, since \( f(m) = \text{last}(f^*(m+1)) \). The history of \( f \) incorporates all of the memory of \( f \). It is therefore tempting to generalize primitive recursion (which defines \( f(m+1) \) from \( f(m) \)) using \( f^* \), to define \( f(m+1) \) from \( f(0), \ldots, f(m) \).

**Definition 8.3** Let \( h : \mathbb{N}^{k+2} \to \mathbb{N} \). The function \( f : \mathbb{N}^{k+1} \to \mathbb{N} \) is defined by complete primitive recursion from \( h \) if

\[
f(m, \overline{n}) = h(f^*(m, \overline{n}), m, \overline{n}).
\]

Given the primitive recursive function \( h \), define \( g \) by

\[
g(0, \overline{n}) = 0,
\]

\[
g(m + 1, \overline{n}) = \text{cons}(g(m, \overline{n}), h(g(m, \overline{n}), m, \overline{n})).
\]

It is a primitive recursive definition of \( g \), and it is clear that \( g = f^* \). All that is left to do is to define \( f \) by composition:

\[
f(m, \overline{n}) = \text{last}(g(m + 1, \overline{n})),
\]

which proves that \( f \) is primitive recursive.

The definition of primitive recursive functions is an inductive definition. Since each of the operators has only a finite number of arguments, this definition can be reformulated ‘from below’: for every primitive recursive function \( f \), there exists a sequence of functions \( f_0, \ldots, f_n \), with \( f_n = f \), each function \( f_i \) being a base function or a function defined from preceding functions by composition or primitive recursion. Inductive definition can also be seen as an inference system: the base functions correspond to the axioms and the composition operator to the rule:

\[
h, g_1, \ldots, g_p \quad \text{primitive recursive} \\
\hline
h(g_1, \ldots, g_p) \quad \text{primitive recursive}
\]

Thus it can be formally proven that a given function is primitive recursive, as a derivation (a term) in this inference system (more precision would be needed, since the preceding rule using ‘...’ is not very formal). Note that a derivation of ‘\( f \) primitive recursive’ is almost a method to compute \( f \). Furthermore, an arbitrary derivation (a term) of this inference system evidently defines a function, since its definition appears in the conclusion.

**Recursive functions**

It is easy to write programs which do not compute primitive recursive functions, ones which do not always terminate (they compute partial functions). It is difficult to write an interesting program which always terminates but which does not compute a primitive recursive function.
Example Ackermann defined a function $A$ in 1928 [3]; see Chapter 2. Here is a slight variant:

$$
A(0, y) = y + 1, \\
A(x + 1, 0) = A(x, 1), \\
A(x + 1, y + 1) = A(x, A(x + 1, y)).
$$

This function is easily computable by a recursive program—although proof of its termination is not easy—but it is not primitive recursive. It can be shown that $A(n, n)$ increases faster than any primitive recursive function.

Here is another example, which appears to be more artificial. It is possible to enumerate all of the derivations (or constructions) of primitive recursive functions effectively. To the $n$-th derivation, associate a primitive recursive function $\phi_n$, computable by following the derivation step by step. Define $\phi(n) = \phi_n(n) + 1$. This function is computable: to compute $\phi(5)$, write the 5-th derivation of the primitive recursive functions, which constitutes a procedure for $\phi_5$, compute $\phi_5(5)$ and add 1. However, if $\phi$ were primitive recursive, there would exist an integer $p$ such that $\phi = \phi_p$. But $\phi(p) = \phi_p(p) + 1 \neq \phi_p(p)$. Therefore there exist computable functions which are not primitive recursive. The third operation, minimization, must be introduced.

**Definition 8.4 (Herbrand, Gödel)** The recursive functions are the elements of the least set of numeric functions containing the base functions and closed under composition, primitive recursion and total minimization.

Recursive functions can be programmed in any language allowing arbitrary recursive calls or having a ‘while’ iteration structure. In ML, the minimization operator can be defined by:

```
fun mu g n = let fun h c = if g(n,c)=0 then c else h(c+1)
         in h 0 end ;
```

Making this closure condition explicit implies that, for every recursive function $f$, there exists a sequence of functions $f_0, \ldots, f_n$, with $f_n = f$, each function $f_i$ either being a base function or defined from preceding functions using one of the three operators. This sequence of definitions is called a Herbrand–Gödel system. However, it is not natural to consider this inductive definition as an inference system, because of the condition to be checked before ‘inferring’ that a function obtained through minimization is recursive. Rules containing validity conditions have already been shown. This is the case for the $\forall$-introduction rule in natural deduction: there, the condition (that a variable is not free in the assumptions) is easy to check; here, the existence of a zero is of another order of difficulty: it is an ‘undecidable’ problem.

The minimization operator prescribes a search for a value in an unbounded set. This characteristic of computing procedures is essential. It is missing in primitive
recursion, which limits the usable resources. The \( \lambda \)-calculus terms constructed using \( \beta \)-reduction can be of arbitrary length (one cannot predict ahead before starting the computation how much paper and ink are necessary to attain a normal form). And Turing machines, it will be shown, use an ‘infinite’ tape.

The minimization operator is more powerful than primitive recursion. It will be shown that a function defined by complete primitive recursion can be obtained by compositions and minimizations. This is particularly true for functions defined by ordinary primitive recursion. This result of recursion ‘elimination’ will be the subject of a much more general theorem in the case of partial recursive functions (theorem 8.18).

Let \( f \) satisfy \( f(m) = h(f^*(m), m) \) for every \( m \geq 0 \) (omit the parameters to simplify). Then \( f(m - i) = h(f^*(m - i), m - i) \) for every \( i, 0 \leq i \leq m \), which can be stated by

\[
\text{last}(i - 1, f^*(m)) = h(\text{pre}(m - i, f^*(m)), m - i).
\]

Hence, \( x = f^*(m) \) is a solution of the system of equations:

\[
\begin{align*}
\text{last}(i - 1, x) &= h(\text{pre}(m - i, x), m - i) & (1 \leq i \leq m), \\
\|x\| &= m.
\end{align*}
\]

Define \( \bar{f} \) by \( \bar{f}(m) = \mu x. h'(x, m) = 1 \), where \( h' \) is the primitive recursive function defined by composition and bounded minimization of primitive recursive functions:

\[
h'(x, m) = \|x\| = m \land \forall i (1 \leq i \leq m) \text{last}(i - 1, x) = h(\text{pre}(m - i, x), m - i).
\]

Then \( \bar{f} \) is primitive recursive, since it is defined explicitly by minimization of \( h' \) and satisfies \( \bar{f} = f^* \). And \( f \) is defined by composition of \( \bar{f} \) and \( \text{last} \):

\[
f(m) = \text{last}(\bar{f}(m + 1)).
\]

It is remarkable that the introduction of minimization allows the generation of all computable functions. The following theorem states the equivalence between Herbrand–Gödel and Church’s definitions.

**Theorem 8.1** For \( f : \mathbb{N}^p \rightarrow \mathbb{N} \) the following properties are equivalent:

1. \( f \) is \( \lambda \)-definable;
2. \( f \) is recursive.

**Proof** It will be shown that recursive functions are \( \lambda \)-definable. The proof is quite simple, but long. A ‘compiler’ must be created; it translates a Herbrand–Gödel system into \( \lambda \)-calculus terms. This process is realistic, since the source language is of sufficiently high-level, and almost immediately translatable into usual programming languages. The proof uses induction, with cases for base functions, composition, primitive recursion and minimization.
For the base functions, the following terms are used:

\[
\begin{align*}
0 & \equiv \lambda f . x, \\
\text{succ} & \equiv \lambda nf . f(nfx), \\
\text{pr}_n^k & \equiv \lambda x_1 \ldots x_n . x_k.
\end{align*}
\]

First consider composition. Let \( h : \mathbb{N}^p \rightarrow \mathbb{N} \) and \( g_i : \mathbb{N}^k \rightarrow \mathbb{N} \ (1 \leq i \leq p) \), functions whose respective \( \lambda \)-definable terms are \( H \) and \( G_i \). If

\[
f(n_1, \ldots, n_k) = h(g_1(n_1, \ldots, n_k), \ldots, g_p(n_1, \ldots, n_k)),
\]

let

\[
F \equiv \lambda x_1 \ldots x_k . H(G_1x_1 \ldots x_k) \ldots (G_p x_1 \ldots x_k).
\]

Then \( F \) defines \( f \).

For primitive recursion, suppose that \( g : \mathbb{N}^k \rightarrow \mathbb{N} \) and \( h : \mathbb{N}^{k+2} \rightarrow \mathbb{N} \) are \( \lambda \)-defined by terms \( G \) and \( H \) respectively and let

\[
\begin{align*}
f(0, \bar{\alpha}) & = g(\bar{\alpha}), \\
f(m+1, \bar{\alpha}) & = h(f(m, \bar{\alpha}), m, \bar{\alpha}).
\end{align*}
\]

This definition of \( f \) can be interpreted as a fixpoint equation, by writing it in the form

\[
f(m, \bar{\alpha}) = \begin{cases} 
  g(\bar{\alpha}) & \text{if } m = 0 \\
  h(f(m-1, \bar{\alpha}), m-1, \bar{\alpha}) & \text{otherwise}.
\end{cases}
\]

It suffices to know the \( \lambda \)-terms \( \text{pred} \) and \( \text{zero} \) which define the functions \( m \mapsto m-1 \) and \( m, u, v \mapsto \text{if } m = 0 \text{ then } u \text{ else } v \) to apply a fixpoint combinator \( Y \) and deduce from it that

\[
F = Y (\lambda f x \bar{y}. \text{zero}(G\bar{y})(H(f(p\text{pred } x)\bar{y})(\text{pred } x)\bar{y})).
\]

In this case, using a fixpoint combinator is a sledgehammer solution that a good compiler would not use. In fact, this is not an arbitrary syntactic recursive definition but a very particular computation scheme, for which it is known, for example, that it always terminates in \( m \) (the first argument) steps: it is preferable that \( F \) be normalizable. The solution is to construct a primitive recursion combinator \( R \), normalizable, so that \( F \) can be defined from \( R, G \) and \( H \).

The idea is to replace primitive recursion by iteration, a simpler operation, over pairs. In fact, for a given \( k \)-tuple \( \bar{n} \), \( f(m, \bar{n}) \) is computed by iterating the mapping

\[
(x, y) \mapsto (h(x, y, \bar{n}), y + 1)
\]
m times from the pair \((g(\bar{n}), 0)\). There is also a combinator, \( \text{cons} \equiv \lambda xy . m . x y \), which produces pairs. Write \([X, Y]\) instead of \( \text{cons } XY \). The iteration over pairs can be defined by the term

\[
\lambda p . [H(pT)(pF)\bar{n}, \text{succ}(pF)],
\]
where $p$ is a ‘pair’ variable, $T$ and $F$ are the Booleans true and false. Let

$$Q \equiv \lambda uv. [v(pT)(pF), \text{succ}(pF)].$$

The function $Qv$ is iterated $m$ times, starting from $[u, T]$ ($u$ will be $G\bar{n}$ and $v$ will be $\lambda xy. Hxy\bar{n}$):

$$m(Qv)[u, T].$$

It is a pair whose first component is applied to $T$:

$$m(Qv)[u, T]T.$$

Abstraction is made over $m$ and $v$ to make a function $f$:

$$\lambda vm. m(Qv)[u, T]T,$$

and then over $u$ to obtain the primitive recursion combinator $R$:

$$R \equiv \lambda uvvm. m(Qv)[u, T]T.$$

It remains to $\lambda$-define the function $f$ by the term:

$$F \equiv \lambda m\bar{n}. \lambda xym. m(G\bar{n})(\lambda xy. Hxy\bar{n})m.$$

For minimization, suppose that $g : N^{k+1} \rightarrow N$ satisfies the total minimization condition and is $\lambda$-defined by term $G$. Let $f(\bar{n}) = \mu m. [g(\bar{n}, m) = 0]$. Essentially, a while loop is to be translated into the $\lambda$-calculus. First a combinator $W$ is constructed such that

$$WUn = \begin{cases} n & \text{if } Un = 0 \\ WU(\text{succ } n) & \text{if } Un \text{ is a numeral } \neq 0. \end{cases}$$

How does one obtain $W$? $WUn$ must contain a term $V$ which takes as argument $Un$ and which tests if it is null. Use the combinator $D$,

$$D \equiv \lambda xym. m(Ky)x.$$

The pair $DXY$ is a functional object, which serves as conditional operator testing if its argument is null:

$$DXYP = \begin{cases} X & \text{if } p = 0 \\ Y & \text{if } p > 0. \end{cases}$$

Take for $V$ a pair $DAB$. If $Un = 0$, $A$ must return $n$ and forget everything else. Otherwise, $B$ must continue the computation where it was, with $V$ and $n$ as arguments, by incrementing $n$. It is therefore possible to have $WUn = V(Un)\bar{V}n$, with $V = DAB$ and to take $A \equiv \lambda vm. n$ so that $AVn = n$, and $B \equiv \lambda vm. v(U(\text{succ } n))v(\text{succ } n)$, so that $BVn = WU(\text{succ } n) = V(U(\text{succ } n))$
\( V(\text{succ } n) \). Since \( B \) depends on \( U \), define \( V = TU \), defining first \( T \), then \( W \) using \( T \):

\[
T \equiv \lambda u. D(\lambda vn. n)(\lambda vn. v(u(\text{succ } n))v(\text{succ } n)), \\
W \equiv \lambda un. Tu(un)(Tu)n.
\]

The function \( f \) is then \( \lambda \)-defined by the term

\[
F = \lambda n. W(Gn)0.
\]

The converse (every \( \lambda \)-definable function is recursive) will be deduced from general results on inference systems.

\( \blacksquare \)

Turing machines

Neither \( \lambda \)-definable functions nor recursive functions are particularly convincing models of computability. Turing undertook to analyze the (human, mental) computation process and proposed a new model in 1936 [151]. His machines are automata, such as finite state push-down automata, which preceded the von Neumann machine by ten years.

A fixed 'blank' symbol, \( \square \), will be used below.

**Definition 8.5** A Turing machine is specified by three finite sets \( \Gamma, Q \) and \( R \):

- \( \Gamma \) is the tape alphabet, where \( \square \not\in \Gamma \);
- \( Q \) is the set of states;
- \( R \) is the set of rules, which are 5-tuples \( (q, s, q', s', d) \) where \( q, q' \in Q, s, s' \in \Gamma \cup \{\square\} \) and \( d \in \{-, 0, +\} \).

It is useful to have a 'physical' description of a Turing machine. There are three components:

- an infinite tape divided into cells, each capable of holding a symbol from \( \Gamma \) or a blank symbol;
- a read/write head on the tape which moves at most one cell to the left or right at each step;
- a control unit which contains the 'program' \( R \) and which, at each step, can be found in one of the states of \( Q \).

As for any automaton, a Turing machine works by transitions between configurations, starting from an initial configuration and working towards a possible final configuration. It is supposed that in the initial configuration, the tape contains a word from \( \Gamma^* \), with blank symbols to the left and to the right. Each transition rule can trigger the writing of an element of \( \Gamma \cup \{\square\} \) on the tape. After a finite number of transitions, the tape can only contain a finite number of elements of \( \Gamma \), with, perhaps, inter-spaced blanks. The contents of the tape can therefore be represented by a word in \((\Gamma \cup \{\square\})^*\), by omitting all the prefix and suffix \( \square \)'s.
Example. The tape contains $bb \Box a \in (\Gamma \cup \{\Box\})^*$.\hfill $\Box$

To represent a machine other than by a picture, the concept of configuration must be formalized by incorporating the current state. A configuration is a word of the form $aq\beta$ where $q \in Q$, $\alpha, \beta \in (\Gamma \cup \{\Box\})^*$, with $\alpha$ (respectively $\beta$) not starting (respectively finishing) with $\Box$. The configuration $aq\beta$ states that the tape contains the word $\alpha\beta$, that the state is $q$ and that the head points to the beginning of $\beta$. In the preceding example, the configuration is $bbq \Box a$.

Here is the definition of the transitions of a Turing machine $(\Gamma, Q, R)$: if, in state $q$, the head points to the symbol $s \in \Gamma \cup \{\Box\}$ and if $\rho = (q, s, q', s', d) \in R$, the $\rho$ triggers the following transition:

- the state becomes $q'$;
- the head writes $s'$ to replace $s$ and then moves one cell to the left if $d = -$ or to the right if $d = +$.

Write this transition $c \xrightarrow{\rho} c'$ as a relation between configurations $c$ (before) and $c'$ (after). The transition rules can be interpreted as rewrite rules of words over configurations:

$$aqs \rightarrow \begin{cases} 
as'q' & \text{if } d = + \\ aq's' & \text{if } d = 0 \\ q'as' & \text{if } d = - \end{cases}.$$

A halting configuration is a configuration for which no rule is applicable (it is a word that is irreducible by rewriting). A computation is a sequence of transitions which is either infinite, or finite and terminating by a halting configuration:

$$C_0 \xrightarrow{\rho_1} C_1 \xrightarrow{\rho_2} \ldots \xrightarrow{\rho_p} C_p.$$

The sequence $\rho_1, \ldots, \rho_p$ is the trace of the computation.

A Turing machine has two uses: as an automaton recognizing a language, and as a transducer. This last use will allow the definition of Turing-computable functions. The definition of the computations of a Turing machine leaves, however, the choice of rule to apply to a given configuration, which does not correspond to the normal idea of a program (but does correspond to the activity of a human being\(^1\)). A deterministic Turing machine (DTM) is a machine $(\Gamma, Q, R)$ such that for every

\(^1\)Neither of us believes in predestination.
Let \( f : \Gamma^* \rightarrow \Gamma^* \) be a function. A DTM \((\Gamma, Q, R)\) computes \( f \) if for every \( \alpha \in \Gamma^* \) the computation starting from the initial configuration \( q_0 \alpha \) terminates in a configuration \( q \beta \) and if \( f(\alpha) = \beta \) (it is not necessary to impose that the head points to the first letter of \( \beta \)). The function \( f \) is \textit{Turing-computable} if there exists a DTM computing \( f \).

This concept of computability, applicable to words, can be transposed to the integers by encoding the integers and \( k \)-tuples of integers. Take as alphabet \( \Gamma = \{0, 1\} \), and encode the integer \( n \) by \( n + 1 \) letters ‘1’ and the \( k \)-tuple \( (n_1, \ldots, n_k) \) by separating the codes for \( n_{i, i+1} \) with a ‘0’: 1110111101 encodes the triple \( (2, 3, 0) \). A function \( f : \mathbb{N}^k \rightarrow \mathbb{N} \) therefore defines a function \( \tilde{f} : \Gamma^* \rightarrow \Gamma^* \) associating with the encoding of a \( k \)-tuple the encoding of its image by \( f \). Note that the image of \( \tilde{f} \) is contained in the language \( \{1\}^+ \).

A numeric function \( f : \mathbb{N}^k \rightarrow \mathbb{N} \) is \textit{Turing-computable} if, after encoding in \( \Gamma \), the function \( \tilde{f} : \Gamma^* \rightarrow \Gamma^* \) is Turing-computable by a DTM.

\textbf{Example} The successor function, \( s : \mathbb{N} \rightarrow \mathbb{N} \), is Turing-computable by the following DTM:

\[
\begin{align*}
\Gamma & = \{1\}, \\
Q & = \{q_0, q_1\}, \\
R & = \{(q_0, 1, q_0, 1, +), (q_0, \Box, q_1, 1, 0)\}.
\end{align*}
\]

Turing showed that all the recursive functions are Turing-computable. His proof, by induction over the construction of recursive functions, consists of compiling the Herbrand–Gödel system into a formalism resembling a macro-assembler and is not particularly interesting. Conversely, all the Turing-computable functions are recursive: the proof of this property is not immediate, but it will follow from more general properties.

If Turing machines give a convincing description of computation, their influence on the design of high-level languages is very limited. From the theoretical point of view, many constructions and proofs are done using Turing machines. Furthermore, they allow a precise definition of the number of ‘steps’ and ‘cells’ necessary for a computation, hence of the temporal and spatial complexities of an algorithm. Finally, the Turing machine model is often extended (several tapes, several heads, \ldots) to facilitate the handling of certain problems, without the expressive power being changed (this is proven for each extension): it is called a ‘robust’ model.
8.1.3 Church’s thesis

This ‘thesis’, formulated by Church in 1936 [15], is the result of theoretical results and practical experience. The theoretical results are the ‘equivalence theorems’ between recursive functions, \(\lambda\)-definable functions and Turing-computable functions. Each of these definitions offers a different vantage over the same class of functions. What is the interest of these diverse characterizations?

The usefulness of the \(\lambda\)-calculus for functional programming has already been indicated.

The Herbrand–Gödel definition of recursive functions is the closest to typical programming languages. It highlights the class of primitive recursive functions, which contains all of the usual arithmetic functions. In terms of sequential programming, the primitive recursive functions can be characterized as those functions which can be computed using finite (for) loops. Furthermore, their equational definition can be seen as a noetherian and confluent system.

Finally, the description of computation given by Turing seems to be fine enough to cover any imaginable computation.

In addition, it is a result of experience that all known computable functions have been shown to be recursive. And contemporary practice has only confirmed this experience.

*Church thesis* \[\text{‘Every effectively computable function is recursive.’}\]

In this formulation, only the word *recursive* has a rigorous definition, while *effectively computable* refers to the ‘physical’ concept of computation: effective, mechanical, automatic, etc. It is certainly not a theorem, but a postulate of experimental origin. The thesis has practical consequences:

- Do not try to write a program to compute a function which has been shown to be non-recursive.
- A visibly computable function can, during a proof, be considered to be recursive to avoid a tedious proof (but one must be sure!), and one can then deduce, for example, the possibility of a Turing machine computing it without having to construct one explicitly.

One normally refers to computable functions, for recursive, \(\lambda\)-definable or Turing-computable functions. The list of equivalent characterizations of computable functions can still be completed:

- canonical systems (Post 1943 [123]);
- theory of Markov algorithms (1951 [99]);
- elementary formal systems (Smullyan 1961 [138]);
The Church thesis, as given here, deals with the class of computable functions as extensional objects, inside the class of functions, and not as intensional computational processes. The equivalence theorems state that this class is well defined. It can be obtained by using several more or less useful definitions, and it is the multiplicity of approaches which makes this class particularly interesting and which motivates the Church thesis.

The proofs of equivalence theorems prove a lot more than the simple equality between sets of functions. They are constructive proofs which transform a Herbrand–Gödel definition into a $\lambda$-term, and into a Turing machine, and conversely. The same concept of algorithm underlies all these constructions. These remarks justify the following statement:

\begin{quote}
**Church (strong) thesis**

‘If a function $f$ is computable by an algorithm, then this algorithm can be effectively transformed into a Turing machine computing $f$.’
\end{quote}

One can then freely refer to an algorithm, in the same manner that the first version of the Church thesis allows one to refer to a computable function—without specifying the model used. In what follows, algorithms will be designated using the notations which define them: for example a Turing machine will be referred to as an algorithm, and when an encoding of these machines is introduced, an integer $n$ will represent an algorithm executed by the $n$-th Turing machine.

Logicians have never been able to define the concept of algorithm as generally as the concept of function. For Turing machines, one could state that an algorithm is defined by a Turing machine. For the $\lambda$-calculus, a reduction strategy must be associated with a term. For recursive functions, it is a system of Herbrand–Gödel equations. An algorithm always uses a notation (a triplet $(\Gamma, Q, R)$, a term or a system of equation), which is only usable as ‘instructions’ when more general constructions (a word rewriting relation, $\beta$-reduction or an inference system) indicate how these instructions are executed. These notations do not in themselves represent the computing procedures, but do suffice to define them.

For us, an algorithm is a computation procedure which is practised without really looking for a formal definition. A major contribution of logic is to have given a mathematical status to ‘computation’ by defining convincing models. Logic should then be seen as an experimental science, just like mechanics, which succeeded in defining the concept of force from that of interaction of bodies. It is this aspect, and not that of metamathematics or that of theory of theories, which is at the origin of its natural links with computer science.

### 8.1.4 Kleene and partial functions

The preceding processes sought the computable functions from $N^k$ to $N$ and how to compute them. By doing this, computing methods—leftmost $\beta$-reduction of a


\section*{The computable landscape}

\lambda\text{-term and rewriting of configurations in a Turing machine—and notations were introduced to specify these processes: \lambda\text{-terms, deterministic Turing machines or Herbrand–Gödel systems. If } f \text{ is computable, there exists a term } M_f, \text{ a Herbrand–Gödel system and a DTM which compute } f.\text{ Conversely, what should be done with an arbitrary term, system or DTM? The same computing processes can always be used, but in general, they will compute partial functions.}

A partial function from \( A \) to \( B \), written \( f: A \rightarrow B \), is a (total) function from a subset \( D(f) \subseteq A \), the domain of \( f \), to \( B \). The expression \( f(a) \downarrow \) means that the partial function \( f \) is defined in \( a \), i.e., \( a \in D(f) \). What is the meaning of a partial function and its domain? From a denotational point of view, \( a \notin D(f) \) states simply that \( f(a) \) is not defined. From an operational point of view, either the computation of \( f(a) \) never terminates or there is no instruction to compute it, in which case an error is detected and the computation cannot continue. It will be shown that it is not possible in general to decide in advance if a computation halts: the concept of domain of a partial function is not effective. The first undecidability problems appear with the general use of these computation processes.

Hence, if \( M \) is a \( \lambda \text{-term}, M_{n_1} \ldots n_k \) is not always normalizable and its normal form, if it exists, is not necessarily a Church numeral.

In the case of a deterministic Turing machine \(( \Gamma, Q, R \)) with initial state \( q_0 \in Q \), consider the initial configuration \( q_0 \alpha \), where \( \alpha \in \Gamma^* \). The unique computation starting in this configuration does not necessarily halt. If it halts in configuration \( \beta q \gamma \), let \( f_R(\alpha) = \beta \gamma \). The function \( f_R: \Gamma^* \rightarrow \Gamma^* \) is the partial function computed by \(( \Gamma, Q, R \)).

Consider numerical functions. The problem is no longer to find a DTM computing a given function, rather to start from an arbitrary DTM over the alphabet \( \Gamma = \{0, 1\} \) and to make it run. Let \( k \geq 1 \) be fixed, and consider the initial configurations which correspond to encodings of \( k \)-tuples of integers: they are of the form \( q_0 \alpha \), where \( \alpha = 1^{m_1}0 \ldots 01^{m_k} \). As in the case of \( \lambda \text{-terms, the computation from one of these initial configurations, when it halts, does not necessarily produce a configuration containing the encoding of an integer. When the final configuration is } \beta q \gamma, \text{ with } \beta \gamma = 1^{m+1}, \text{ let } f_R^k(m_1, \ldots, m_k) = m. \text{ The function } f_R^k \text{ is the } k \text{-ary partial function computed by } R. \text{ For a given } R \text{ over the alphabet } \{0, 1\}, \text{ the preceding constructions yield partial functions } f_R: \Gamma^* \rightarrow \Gamma^*, f_R^1: \mathbb{N} \rightarrow \mathbb{N}, f_R^2: \mathbb{N^2} \rightarrow \mathbb{N}, \text{ etc. The } f_R^k \text{ are, by definition, the Turing-computable partial functions.}

Consider also systems of recursive functions. Each of the three Herbrand–Gödel operations is ‘computable’ by a sequential program. Minimization is very different from the others: the existence of zeros over \( g \) must be satisfied before one can define \( f(x) = \mu y. [g(x, y) = 0] \). If minimization is applied without checking this condition, a program might not halt and will then only compute a partial function. If it halts, the number of iterations is not known in advance, while in primitive recursion exactly \( k \) iterations are used to compute \( f(k, \vec{n}) \). The existence of such zeros is not necessarily decidable: the only general way to check and obtain the minimum is to compute \( g(n, 0), g(n, 1), g(n, 2), \ldots \) successively. Furthermore, there is no
way to extend effectively a partial function $f$ obtained under minimization to a total function. A definition such as

$$f(\bar{n}) = \begin{cases} \min \{k \in \mathbb{N}; g(\bar{n}, k) = 0\} & \text{if this minimum exists} \\ 0 & \text{otherwise} \end{cases}$$

is not effective, even if $g$ is computable.

Therefore, the class of total recursive functions is not well adapted to programming. It must be extended to partial functions, which are the objects naturally computed by a computation process. Kleene studied them systematically starting in 1938 [80]. If partial functions are allowed, the total minimization condition is no longer necessary.

There is worse to come: the minimization operation, applied as a partial function, is itself not necessarily computable.

Example Let $h$ be a computable partial function such that the problem $m \in D(h)$? is undecidable. Let $g$ be the partial function defined by

$$g(m, n) = 0 \quad \text{if } n = 1, \text{ or if } n = 0 \text{ and } m \in D(h),$$

and $f$ by $f(m) = \mu n . [g(m, n) = 0]$. Then

$$f(m) = \begin{cases} 0 & \text{if } m \in D(h) \\ 1 & \text{otherwise} \end{cases}.$$

It is clear that $f$ is not computable since it is a definition by cases over an undecidable condition. However, if this definition is translated into a program, the latter will compute $f'$, which equals 0 over $D(h)$ and is not defined elsewhere, since if $m \notin D(h)$, the program computing $h(m)$ does not halt, so the decision as to whether $f(m) = 1$ is never made.

This example shows that the class of computable partial functions is not closed under minimization and that when there are partial functions the operational interpretation of a definition can differ from its (mathematical) denotational interpretation.

A precise definition for the class of partial recursive functions, the same as the classes of partial Turing-computable and $\lambda$-definable functions, will be given later on. Practice allows the statement of a version of the Church thesis for partial computable functions:

**Church–Kleene thesis**

"An effectively partial computable function is recursive."

The use of this version is sometimes dangerous, as was illustrated by the above example of non-effective minimization.

The fundamental results of recursive function theory, due to S. Kleene, are presented in the framework of computable partial functions. Recursive function theory
is an important branch of logic, and several applications to metaprogramming will be presented. Before defining the partial recursive functions rigorously, the Church thesis will be used to equate partial recursive Turing-computable functions.

It is time to distinguish the different uses of the word 'recursive': the logical, syntactical and algorithmic points of view.

In logic, 'recursive' refers to recursive functions and, by extension, to the study of computability; the quality of being recursive is recursiveness.

In computer science, there are several uses of 'recursive'. A definition of an object is called recursive when this object is defined in terms of itself. Hence an inductive definition is recursive: this use corresponds better to impredicativity in logic. Syntactic recursivity refers to when the same symbol appears in the two sides of a definition. For example, the rule in a context-free grammar \( Z \rightarrow aZ \) is recursive since the non-terminal \( Z \) appears to the left and to the right of the \( \rightarrow \). A definition of the form

\[
\text{datatype nat = zero | succ of nat ;}
\]

is recursive since \( \text{nat} \) appears on both sides of the \( = \). Similarly, one refers to recursive definitions of functions and procedures. These functions are obviously computable, hence recursive, in the logical sense. These definitions generally have a fixpoint equation as semantics. However a mathematician would never call the differential equation \( y' = y \) 'recursive'.

In programming, the resources required to execute a task distinguish recursive from iterative. A procedure is recursive if its execution requires the repeated allocation of memory, in practice, the growth of an unbounded stack. In general, a procedure whose definition is syntactically recursive is executed in a recursive manner. A procedure is iterative if its execution can be done in constant space. This is the case of a program which only uses iterative control structures. It it also the case for procedures using tail-recursion, which is syntactically recursive, if the compiler knows how to handle them correctly: it is unnecessary to preserve the stack when tail-recursion is being used.

When recursive functions of logic are to be computed by a program, a Herbrand–Gödel system can be translated into recursive function definitions—primitive recursion is already in this form, and minimization calls for computing \( g(0) \), where \( g(n) \) calls \( g(n + 1) \). An iterative procedure can also be constructed: a \texttt{for} loop for primitive recursion and a \texttt{while} loop for minimization.

These three uses are all distinct.

8.1.5 Computing in a logical system

First-order logic and arithmetic
Consider first-order logic with the NK system, and let \( T \) be a theory with equality. Since what are of interest are numeric functions, suppose that the language \( L_S \) of \( T \) contains a constant (or a closed term) \( n \), called \textit{numeral}, for each integer \( n \), and
that \( T \vdash \neg(n = m) \) as soon as \( n \) and \( m \) are distinct integers. This is the case for elementary arithmetic where \( n \) is \( s^n 0 \) (see exercise 26).

**Definition 8.6** Let \( f : \mathbb{N}^k \rightarrow \mathbb{N} \). The function \( f \) is **\( T \)-defined** by formula \( \varphi \in L_{\Sigma[x_1, \ldots, x_k, y]} \) if: for every \( (n_1, \ldots, n_k, m) \in \mathbb{N}^{k+1} \), \( f(n_1, \ldots, n_k) = m \) if and only if

\[
T \vdash \forall y (\varphi(n_1, \ldots, n_k, y) \iff y = m).
\]

\( f \) is then **\( T \)-definable**. Note that this condition implies in particular the theorem \( \varphi(n_1, \ldots, n_k, m) \), if \( f(n_1, \ldots, n_k) = m \).

The problem is to determine which theories define all the recursive functions. It is clear that if a theory \( T \) is suitable, every extension of \( T \) is suitable as well. However, a restriction \( T' \) of \( T \) (restriction of the language, axioms or inference rules) might not suffice. It is therefore interesting to know the minimal theories.

**Theorem 8.2 (Gödel definability)** A numeric function is recursive if and only if it is definable in Peano arithmetic.

The proof of this system consists precisely of a compilation of a Herbrand–Gödel system into a finite set of formulas of arithmetic. The proof can also be seen as the synthesis of a program from its specification if a definition of a recursive function is seen as a specification and the formulas of arithmetic as programs. Here are some details. The reasoning uses induction over the Herbrand–Gödel definitions of recursive functions; there are three cases. That of composition is easy (exercise 10).

For minimization, let \( f = \mu y \cdot [g(\cdot, y) = 0] \). If \( g \) is defined by the formula \( \varphi(x, y, z) \), it is easy to prove that \( f \) is defined by the formula

\[
\psi(x, u) \equiv \forall y (y < u \Rightarrow \neg \varphi(x, y, 0)) \wedge \varphi(x, u, 0).
\]

It is not as easy for primitive recursion. It was shown that if \( f \) is defined by complete primitive recursion from \( h \), i.e.,

\[
f(m, \bar{n}) = h(f^*(m, \bar{n}), m, \bar{n}),
\]

then the history \( f^* \) of \( f \) is obtained by minimization of a function \( h' \) constructed explicitly (i.e., by composition and bounded minimization) from \( h \) and a few functions linked to the encoding of \( n \)-tuples of integers. Hence \( f \) is obtained directly by composition: \( f(m) = \text{last}(f^*(m + 1)) \). Apart from the other two cases, it suffices to prove that the few encoding functions are definable, which will not be done here.

It would have been more astute to omit primitive recursion from the definition of recursive functions, by adding the predicate \( < \) over \( \mathbb{N} \) to the base functions. It would then suffice to show that the encoding functions can be obtained by (bounded) minimization and compositions from the base functions. But the primitive recursion scheme is quite natural in programming. The difficulty of this case
shows that definability in arithmetic is not well adapted to programming. It would lead to a programming style avoiding 'recursion' and favoring imperative structures, such as while, which translates directly into an arithmetic formula, and systematic encodings of data structures by integers. Peano arithmetic appears to be a very low-level language to describe typical programs. Other concepts of definability, with other inference systems, can be used more naturally.

This result has been extended to elementary arithmetic, which is finitely axiomatizable, and it will be used later on. However, Presburger arithmetic, i.e. with only addition, is insufficient.

Equational logic
Instead of working in a fixed theory, one can, for each recursive function \( f \), construct a theory which defines \( f \): the representation is in the inference system rather than in the theory.

Kleene (1952 [81]) showed that partial recursive functions are definable in equational logic (see Chapter 6, section 6.1.3). It is clear that the construction by composition and primitive recursion is equational in itself. The case for minimization is less simple. Introduce two new binary symbols \( a \) and \( b \), along with the equations:

\[
\begin{align*}
a(u, su) &= u, \\
b(u, 0) &= u.
\end{align*}
\]

Consider \( f \) to be defined so that \( f(x) = \mu y. [g(x, y) = 0] \), where \( g \) has already been defined by a set of equations. Introduce a new symbol \( \bar{g} \) with the same arity as \( g \), along with the equations,

\[
\begin{align*}
\bar{g}(x, 0) &= g(x, 0) \\
\bar{g}(x, sy) &= a(g(x, sy), \bar{g}(x, y)),
\end{align*}
\]

and then the equation defining \( f \):

\[
f(x) = b(u, \bar{g}(x, u)).
\]

Write \( E \) for the set of introduced equations (including those defining \( g, a \) and \( b \). Then \( E \vdash f(n) = m \) if and only if \( f(n) = m \).

Logic programs
In 1977 Tärnlund ([147]) proved that all partial recursive functions are computable by a definite logic program. This can be done by simulating Turing machines, or in the sense of §8.1.5. As in the equational case, only minimization is not obvious.

The integers are coded by terms over the signature \( \{0, s\} \). A \( k \)-ary partial function \( f \) is defined by a program \( P \) with a \( k + 1 \)-ary predicate \( f \) associated with \( f \) such that: \( f(\bar{n}) = m \) if and only if \( f(s^n(0), \ldots, s^{n_k}(0), s^n(0)) \in M_P \), the minimal model of \( P \).
To define \( f(x) = \mu y. (g(x, y) = 0) \), suppose \( g \) is defined by the program \( P_0 \) and the \( k+2 \)-ary predicate \( g \). Introduce the \((k+1)\)-ary symbols \( h \) and \( f \), and form the program \( P \) starting from \( P_0 \) and the clauses (here \( k = 1 \)):

\[
\begin{align*}
h(X, 0) & : h(X, 0). \\
h(X, s(Y)) & : h(X, Y), g(X, Y, s(Z)). \\
f(X, Y) & : h(X, Y), g(X, Y, 0).
\end{align*}
\]

Then the goal \( f(s^n(0), Y) \) has a solution \( Y = s^m(0) \) if and only if \( f(n) = m \).

The \( \lambda \)-calculus

The pure \( \lambda \)-calculus is formalized by an inference system whose judgments are the \( M \vdash N \), where \( M \) and \( N \) are terms. The very first definition which defined computable functions was that of functions definable in this inference system.

The typing rules for terms of a typed \( \lambda \)-calculus involve judgments of the form \( C \vdash M : \tau \) and restrict the formation of terms. Since there are fewer terms, there are fewer numeric functions definable by a typed \( \lambda \)-calculus. In the case of simple types, only ‘polynomials with conditional’ can be defined, which is insufficient for programming. The addition of constants, along with their contraction rules, or of a more powerful typing system, allows the definition of more numeric functions. For example, Gödel’s system \( T \) (a typed \( \lambda \)-calculus with atomic types for integers and Booleans, and for constants zero, successor, recursors and conditionals) allows the definition of all functions whose termination is provable in Peano arithmetic.

8.2 Decidability

This section studies problems from the decidability point of view. The logicians of the 1930s, dealing with the new methodological requirements of mathematicians, the foundation problem, discovered unthought of phenomena which would have profound implications. By creating a real ‘effective’ laboratory, computer science illustrates these phenomena concretely.

8.2.1 Problems and encodings

A problem is specified by the set of its instances and by a query.

Example

Given: an context-free grammar \((Z, A, P, z_0)\).

Problem: is the generated language \( L(z_0) \) non-empty?

The queries considered here allow yes or no as answers: a query determines the set of instances for which the answer is positive. From an abstract point of view, these are decidability problems of the form \( x \in A? \), for a subset \( A \) of the set of instances.
To define a decision problem to an algorithm, the statement of each instance must be made precise. It is natural to suppose that the instances are 'finite' objects which are uniformly and effectively representable by words over a finite alphabet or by integers.

Example (cont.) A context-free grammar can be given by

- the number $n$ of its non-terminal symbols, designated by $z_0, \ldots, z_{n-1}$;
- the number $t$ of its terminal symbols, designated by $a_0, \ldots, a_{t-1}$;
- a finite sequence of production rules, each being a pair formed of a non-terminal and a word over the alphabet $\{z_0, \ldots, z_{n-1}, a_0, \ldots, a_{t-1}\}$; this sequence can be represented by a word over the alphabet, extended by a symbol, e.g., 'l', separating the left- and right-hand sides of rules, along with the rules themselves.

This representation is not quite satisfactory, since it requires an infinite alphabet to allow for an arbitrary number of grammar symbols. These symbols must be coded in a finite alphabet using, for example, the three letters 'n' (non-terminal), 't' (terminal) and 'l': $z_0$ is represented by $n$, $z_1$ by $n1$, $z_2$ by $n11$, etc., and $a_0$ by $t$, $a_4$ by $t1111$, etc. A grammar is therefore represented by a word over the alphabet $\{n, t, 1, l\}$.

It is this kind of encoding which must be produced when the decidability of a problem is to be studied. What follows does not deal with problems which cannot be encoded explicitly. In particular, the results of this section will not be applied to any arbitrary problem. However, in many domains there are quite natural encodings: programming practice ensures that normal objects such as programs or data such as files, floating point numbers and graphs are obviously encoded: it will be rare that the encoding of a problem will be given. An encoding for $k$-tuples of integers has already been given in §8.1.1, but the construction of the Gödel encoding of logic will not be given.

So decision problems will be studied in the abstract form $x \in A?$, for $A \subseteq \mathbb{N}^k$ $k \geq 1$, or sometimes, $A \subseteq X^*$, with a finite alphabet $X$. A decision algorithm is equivalent to the computation of the characteristic function $1_A : \mathbb{N} \rightarrow \{0, 1\}$, of the set $A$ (of encodings) of the instances of the problem which have yes as answer. The semi-characteristic function of a subset $A$ of $\mathbb{N}^k$ will also be used: it is the partial function $(1/2)_A$ of domain $A$ where its value is 1.

8.2.2 Recursive and recursively enumerable sets

Recursive sets are defined from recursive functions.

Definition 8.7

- A subset $A$ of $\mathbb{N}^k$ is called recursive if its characteristic function $1_A : \mathbb{N}^k \rightarrow \{0, 1\}$ is recursive.
• Let $A \subseteq \mathbb{N}^k$. The problem $n \in A$? having as instances the $n \in \mathbb{N}^k$ is called decidable if $A$ is recursive.

Note a few properties of recursive sets which follow from the same properties of recursive functions.

Proposition 8.3 Let $A, B$ be recursive subsets of $\mathbb{N}^k$. Then $A \cap B, A \cup B, A \setminus B$ and $\mathbb{N}^k \setminus A$ are recursive.

Proof Since $1_{A \cap B} = \min(1_A, 1_B), 1_{A \cup B} = \max(1_A, 1_B)$ and $1_{\mathbb{N}^k \setminus A} = 1 - 1_A$, the proposition follows from the recursiveness of functions min, max and $t \mapsto 1 - t$. □

Conversely, recursive functions can be defined from recursive sets. This characterization cannot be extended to partial functions.

Proposition 8.4 A function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is recursive if and only if its graph

$$G_f = \{(n_1, \ldots, n_k, f(n_1, \ldots, n_k)); n_i \in \mathbb{N}\}$$

is a recursive subset of $\mathbb{N}^{k+1}$.

Proof This equivalence is proven by the two relations

$$1_{G_f}(n_1, \ldots, n_k, p) = 1_{\{0\}}(f(n_1, \ldots, n_k) \downarrow p),$$

$$f(n_1, \ldots, n_k) = \mu p [1_{G_f}(n_1, \ldots, n_k, p) = 1].$$

□

The class of recursive sets is closed under union, intersection and complement. However, it is clear that there exist non-recursive sets, using a simple cardinality argument, and they can be obtained by other operations. The simplest construction is projection: if $A \subseteq \mathbb{N}^2$ is recursive, a projection from $A$ to $\mathbb{N}$ is not necessarily recursive, just as the projection of a measurable set of $\mathbb{R}^2$ is not necessarily measurable. The projections of recursive sets can also be obtained by operations significant to computer science, such as the set of results of a program or the set of inputs for which a program halts.

Definition 8.8

• A subset $A$ of $\mathbb{N}^k$ is semi-recursive if it is the projection over $\mathbb{N}^k$ of a recursive subset of $\mathbb{N}^{k+1}$.
• A subset $A$ of $\mathbb{N}$ is recursively enumerable (r.e.) if it is empty or is the image of a recursive function.

It will be shown using the Church thesis that these two classes of sets are the same.
Let \( A \subseteq \mathbb{N}^k \) be a semi-recursive set, i.e. a projection over \( \mathbb{N} \) of a recursive set \( B \subseteq \mathbb{N}^{k+1} \), \( l \geq 1 \). Then the semi-characteristic function of \( A \),

\[
(1/2)_A(\vec{m}) = \exists n . \ [1_B(\vec{m}, n) = 1],
\]
is computable since it is obtained by minimization of a recursive function, and \( A \) is the domain of \( (1/2)_A \). Hence a semi-recursive set is the domain of a partial computable function.

Let \( f : \mathbb{N}^k \rightarrow \mathbb{N} \) be a partial computable function and \( M \) an algorithm (a \( \lambda \)-term or a Turing machine) computing \( f \). Associate with \( M \) a recursive set \( G_M \) which is an ‘effective hull’ of the graph of \( f \). The set \( G_M \) is formed of \((k + 2)\)-tuples \((\vec{m}, n, p) \in \mathbb{N}^k \times \mathbb{N} \times \mathbb{N} \) such that the computation of \( f(\vec{m}) \) halts in less than \( p \) steps and produces \( n \). It is computable: the computation of \( 1_{G_M}(\vec{m}, n, p) \) consists of making \( M \) run at most the \( p \) first steps of the computation of \( f(\vec{m}) \). If it halts at a step \( p' \leq p \), then \( 1_{G_M}(\vec{m}, n, p) = 1 \), otherwise \( 1_{G_M}(\vec{m}, n, p) = 0 \). This proof can be formalized to prove rigorously that \( G_M \) is recursive. It is a typical example where one can invoke the Church thesis. Note that the image of \( f \) is the projection from \( G_M \) to the \((k + 2)\)-th component, that the domain of \( f \) is the projection from \( G_M \) to the \( k \) first components, and that the graph of \( f \) is the projection from \( G_M \) to the \( k + 1 \) first components. Hence, if \( f \) is partial computable, its image, its domain and its graph are semi-recursive sets.

It remains to be shown that the non-empty domain of a partial computable function can be enumerated by a recursive function. Since all of the considered functions take values in \( \mathbb{N} \), suppose that \( A = D(f) \neq \emptyset \), with \( f : \mathbb{N} \rightarrow \mathbb{N} \). An algorithm \( M \) computing \( f \) will be used, along with the encoding \( (\ ) \) of \( \mathbb{N} \times \mathbb{N} \) to produce a sequence of integers, i.e., a function from \( \mathbb{N} \) to \( \mathbb{N} \), enumerating \( A \) by interleaving the computations of \( f(0) \), \( f(1) \), \ldots . \) For \( n \in \mathbb{N} \), the \((\pi^2(n))\)-th step of the execution of \( f(\pi^1(n)) \) is executed, if it has not halted previously, i.e., the first step of \( f(0) \) is executed, then the second step, then the first of \( f(1) \), the third of \( f(0) \), the second of \( f(1) \), the first of \( f(2) \), etc. If it terminates at instant \( n \), \( \pi^1(n) \) is produced as the new element of the list. This function must be shown in total, i.e., this enumeration must not halt. Since \( A \neq \emptyset \), there exists at least one integer \( m \) in the domain of \( f \), hence an integer \( p \) such that the computation of \( f(m) \) halts in \( p \) steps. Then \( m \) will be produced at instant \( n \), with \( n = (m, p) \), and an infinite number of times subsequently.

To refer to a recursively enumerable set of dimension \( k \geq 2 \), the recursive bijection \( n \mapsto (\pi^1(n), \ldots , \pi^k(n)) \) between \( \mathbb{N} \) and \( \mathbb{N}^k \) is used. It is standard to use ‘recursively enumerable’ (r.e.) rather than ‘semi-recursive’ no matter what the dimension or the mode of definition (projection, domain, graph or image) is. The following proposition has been shown:

**Proposition 8.5** Let \( A \subseteq \mathbb{N}^k \). The following properties are equivalent:

1. \( A \) is the projection of a recursive subset of \( \mathbb{N}^{k+1} \).
2. \( A \) is the image of a recursive function.
3. $A$ is the image of a partial computable function.
4. $A$ is the domain of a partial computable function.
5. $A$ is the graph of a partial computable function.
6. The semi-characteristic function $(1/2)_A$ is partial computable.

In the preceding discussion, the Church thesis was used to handle partial computable functions. It is time to give a precise and simple definition of partial recursive functions.

**Definition 8.9**

- A *partial recursive* function is a partial function whose graph is r.e.
- Let $A \subseteq \mathbb{N}^k$. The problem $n \in A$? having the $n \in \mathbb{N}^k$ as instances is called *semi-decidable* if $A$ is r.e.

Every recursive set is r.e. The construction of non-recursive r.e. sets is not easy. This is the technical version of undecidability results: the best-known examples are the set of provable formulas of arithmetic and the set of normalizable terms of the $\lambda$-calculus. The following proposition explains the term 'semi-decidable', which comes from 'semi-recursive':

**Proposition 8.6**

- $A \subseteq \mathbb{N}^k$ is recursive if and only if $A$ and $\mathbb{N}^k \setminus A$ are semi-recursive.
- The problem $n \in A$? is decidable if and only if the problems $n \in A$? and $n \notin A$? are semi-decidable.

**Proof**

1. If $A$ is recursive, then $A$ and $\mathbb{N}^k \setminus A$ are recursive, hence semi-recursive.
2. Conversely, if $A$ and $\mathbb{N}^k \setminus A$ are semi-recursive, let $B, C \subseteq \mathbb{N}^{k+1}$ be recursive projections $A, \mathbb{N}^k \setminus A$, respectively, over $\mathbb{N}^k$. Let $f(x) = \inf\{n \in \mathbb{N}; (x, n) \in B \cup C\}$, $x \in \mathbb{N}^k$. Under minimization, $f$ is a recursive function from $\mathbb{N}^k$ to $\mathbb{N}$ and $A = \{x \in \mathbb{N}^k; (x, f(x)) \in B\}$. Hence, $A$ is recursive. \(\square\)

The equivalence between some characterizations of r.e. sets, hence semi-decidable problems, was given. The justification of this equivalence is constructive, where, starting from an algorithm computing a partial function of domain $A$, an algorithm effectively enumerating $A$ is constructed. The 'declarative' definition must be distinguished from its operational interpretations: how is a semi-decision procedure for the problem $x \in A$ to be obtained from a definition of $A$?

If $A$ is the projection of a recursive set $B$, the procedure is to compute $1_B(0)$, $1_B(1)$, $1_B(2)$, \ldots, until, possibly, $1_B(x) = 1$.

If $A$ is the image of a recursive function $f$, the sequence $f(0)$, $f(1)$, $f(2)$, \ldots, is computed, until, possibly, $x$ is in the sequence. Note that the number of $f(p)$ to be computed is not known ahead of time, while the resolution of the decidable problem $n \in A$? requires a single computation, that of $1_A(n)$. Thus a problem is
semi-decidable is there exists a program which, given an instance of the problem, halts with ‘yes’ as result if the answer is ‘yes’.

This procedure is no longer valid if \( A \) is the image of a partial recursive function \( f \). The naïve operational interpretation, which consists of successively computing \( f(0) \), \( f(1) \), \ldots, need not enumerate \( A \) completely. It suffices that the computation of \( f(0) \) does not terminate for this procedure to only enumerate the empty set—even if \( f(N) = A \). An analogous situation is that of a tree traversal having infinite branches, as is the case in logic programming: a breadth-first search is ‘complete’, since all the leaves are reached, while a left-to-right depth-first search is ‘incomplete’: if the left branch is infinite, no leaf will be enumerated. The algorithm presented here does not enumerate the results of the computations of the \( f(p) \), but enumerates the computations themselves by ‘interleaving’ them: hence the non-termination of the computation of \( f(p) \) will not block the enumeration of the other results. Programming this interleaving requires coroutine mechanisms and the interruption of computations. This is an uncommon programming structure, similar to the engines of Scheme. Among the programming mechanisms which present themselves naturally in this chapter, such as recursion and compilation, it is the only one which is not yet implemented in a satisfying manner in languages. These techniques are now being introduced in logic and functional programming.

**Summary**

- The recursive functions are the functions obtained from the base function using composition, primitive recursion and total minimization.
- The recursive sets are the sets whose characteristic function is recursive.
- The recursively enumerable (r.e.) sets are the projections of recursive sets.
- The partial recursive functions are the partial functions whose graph is r.e.
- The problem \( x \in A \) is decidable (respectively semi-decidable) if \( A \) is recursive (respectively r.e.).

### 8.2.3 Definition of recursive sets in arithmetic

In first-order logic, a subset \( A \) of \( \mathbb{N}^k \) is defined in theory \( T \) by a formula \( \varphi \in \text{L}_\Sigma[x_1, \ldots, x_k] \) if:

1. \((n_1, \ldots, n_k) \in A \) implies \( T \vdash \varphi(n_1, \ldots, n_k) \);
2. \((n_1, \ldots, n_k) \notin A \) implies \( T \vdash \neg \varphi(n_1, \ldots, n_k) \).

If \( T \) is consistent, ‘implies’ can be replaced by ‘if and only if’. If the function 1\(_A\) is \( T \)-definable, then \( A \) is also \( T \)-definable (see exercise 13). Gödel's theorem (8.2) allows the statement of the following proposition, which will only be extended to r.e. sets in a weaker form, by only requiring part (1) of the preceding definition.

**Proposition 8.7** Every recursive set is definable in elementary arithmetic.
8.2.4 The decidability problem in logic

Let $\Sigma$ be a signature, $\Gamma \subseteq L_{\Sigma}$ a set of axioms and $T$ the theory generated by $\Gamma$. The decidability problem in $T$ is: $\varphi \in T$? i.e., $\Gamma \vdash \varphi$? To study the recursiveness of $T$, an encoding must be constructed. This was done by Gödel in 1931 [55] and it will not be described in detail here. In the following discussion, it is supposed that the signature $\Sigma$ and the set $X$ of variables are countable. All syntactic objects, i.e., elements of $\Sigma$, $X$, $T_\Sigma[X]$, $L_\Sigma[X]$ and $D_\Sigma[X]$ (derivations) are encoded by integers. The symbol $\#k$ will denote the integer encoding object $k$, be it a term, a formula or something else. What is important is that the sets of encodings of each of the sets be recursive and that the usual operations over these objects be themselves encoded by recursive functions: inductive constructions of terms, formulas, derivations and substitutions of a variable by a term in a formula. The classical construction gives sets of encodings and operations which are primitive recursive: this fact will be needed. In particular, the substitution functions $s^k$ satisfying

$$s^k(m, n_1, \ldots, n_k) = \begin{cases} p & \text{if } m = \#\varphi, \text{ for } \varphi \in L_\Sigma[x_1, \ldots, x_k] \\ \text{and } p = \#\varphi[x_1 := n_1, \ldots, x_k := n_k] & m \\ \text{otherwise} \end{cases}$$

will be used. The symbol $^k n$ will stand for term $n$ if $\#k = n$. It is important to distinguish the integer $n$ from the term $n$, and the syntactic object $k$ from its numeric encoding $\#k$ and its syntactic encoding $^k n$.

$T$ is recursively axiomatized if (the set of encodings of) $\Gamma$ is a recursive subset of $\mathbb{N}$. Intuitively, this means that one can effectively decide if a proposition is an axiom. This assumption is reasonable and is satisfied in particular if $\Gamma$ is finite.

Peano arithmetic, which contains an infinity of axioms, because of the recursion axiom scheme, is recursively axiomatized. This is also the case of the theory of algebraically closed fields, of fields of characteristic zero and of ZF set theory. The completion of a logic program is also a recursively axiomatized theory. However, consider complete arithmetic, a theory formed of all the true formulas in $\mathbb{N}$. All those formulas could be taken as axioms, but it will be shown that this set is not recursive, which makes those ‘axioms’ rather limited in their utility since they would be unrecognizable.

If $\Gamma$ is recursive, it can be proven that the set $\text{der}_\Gamma$ of pairs $(\#d, \#\varphi)$ where $d$ is a derivation of $\varphi$ with assumptions in $\Gamma$ is recursive: this means that one can recognize a derivation, that the formulas in the leaves are axioms and that the conclusion is $\varphi$. It follows that the set $\text{th}_\Gamma$ of $\#\varphi$ such that $\Gamma \vdash \varphi$, which is the second projection of $\text{der}_\Gamma$, is recursively axiomatizable. Hence:

**Proposition 8.8** The decision problem of a recursively axiomatizable theory is semi-decidable.

In the case where $\Gamma = \emptyset$, we obtain: the decision problem for the predicate calculus is semi-decidable.
Definition 8.10 A theory is decidable if the set (of encodings) of its theorems is recursive. Otherwise, the theory is called undecidable. A theory $T$ is called essentially undecidable if every extension $T'$ of $T$, over the same signature, is undecidable.

Examples

- Group theory was proven undecidable by Tarski in 1946 [148]. It is easy to generate decidable extensions: the theory of commutative groups, with the additional axiom $\forall xy(xy = yx)$, is decidable (Wanda Szmielew 1948 [146]). There are also decidable fragments of group theory: this is the case for the equational theory since, from the axioms, a noetherian and confluent rewriting system can be generated (6.3.3).
- Ring theory and the theory of commutative rings are undecidable. The theory of algebraically closed fields of characteristic zero is decidable.
- Elementary arithmetic is essentially undecidable. Presburger arithmetic is decidable (1929 [126]).
- If a signature $\Sigma$ contains a binary symbol, then the predicate calculus over $\Sigma$ is undecidable (Church 1936 [15]). If $\Sigma$ is formed only of unary relational symbols, then the predicate calculus over $\Sigma$ is decidable.

Proposition 8.9 Let $T$ be a complete theory. Then $T$ is decidable if and only if $T$ is recursively axiomatizable.

Proof If $T$ is complete, for every proposition $\varphi$, $T \vdash \varphi$ or $T \vdash \neg \varphi$. If $T$ is recursively axiomatizable, there is a semi-decision procedure. It is applied to $\varphi$ and $\neg \varphi$ to solve the problems, $T \vdash \varphi$? and $T \vdash \neg \varphi$?. By running the two procedures in parallel and by halting when one of them halts, which is guaranteed since $T$ is complete, a decision procedure is produced for $T \vdash \varphi$? This proof can be written more formally by using properties of encodings of formulas and the characterization of a recursive set by the semi-recursiveness of itself and its complement. Conversely, if $T$ is decidable, the recursive set of theorems can be taken as the set of axioms. □

Proposition 8.10 Let $T$ be a consistent theory, recursively axiomatizable and satisfying the conditions of §8.1.5. Then every numeric function (respectively every subset of $\mathbb{N}^k$) which is $T$-definable is recursive.

Proof Take $k = 1$ to simplify and consider the case of a function $f : \mathbb{N} \to \mathbb{N}$. Suppose that $f$ is defined in $T$ by the formula $\varphi \in L_2[x,y]$. Since $T$ is recursively axiomatizable, its theorems can be obtained by an enumeration procedure. If $n \in \mathbb{N}$, $m = f(n)$ must be computed. The enumeration procedure is initiated, and halted as soon as a formula $\varphi(n,p)$ is obtained. It will stop since by assumption on $f$ and $\varphi$, $T \vdash \varphi(n,m)$. So let $p$ be the integer upon which the procedure is halted. Since $T$ is consistent and $T \vdash \neg \varphi(n,p)$ if $p \neq m$, $p = f(n)$ must hold. □
From this proof, it should be remembered that under the assumptions of the proposition, an enumeration procedure of the theorems can be transformed into a computation procedure. This (too) general idea makes the design of logic(al) programming theoretically possible. It will be shown in a similar manner that the λ-definable and Turing-computable functions are recursive.

8.2.5 Undecidability of arithmetic

The discovery of the phenomenon of undecidability was one of the major discoveries of this century, particularly in the case of arithmetic, one of the best-known subjects. The undecidability of this theory is a result of its power of computation in the sense of §8.1.5. Here is a technical lemma allowing a syntactic manipulation necessary for the proof of the theorem. Recall that \( \forall \psi \) is the term \( n \) if \( \# \psi = n \).

**Lemma 8.11** Let \( T \) be a theory such that the recursive functions are definable in \( T \). Let \( x \) be a variable and \( \varphi \in L_\Sigma[x] \). There exists a proposition \( \psi \) such that

\[
T \vdash \psi \iff \varphi(\forall \psi^n).
\]

**Proof** Consider the diagonal function \( d : \mathbb{N} \to \mathbb{N} \) defined by:

\[
d(n) = \begin{cases} 
n' & \text{if } n = \# \chi \text{ and } n' = \#(\chi[x := n]) \text{ for } \chi \in L_\Sigma[X] \\
n & \text{otherwise.}
\end{cases}
\]

\( d \) is recursive, hence is defined in \( T \) by a formula \( \delta \in L_\Sigma[x, y] \). In fact, \( d \) is the only recursive function for which one supposes \( T \)-definability. Hence

\[
T \vdash \forall y(\delta(n, y) \iff y = d(n))
\]

for each \( n \in \mathbb{N} \). Let

\[
m = \#(\forall y(\delta(x, y) \Rightarrow \varphi(y))),
\]

\[
m' = d(m),
\]

\[
\psi = \forall y(\delta(m, y) \Rightarrow \varphi(y)).
\]

By the definition of \( d \), \( \# \psi = d(m) \). From the derivation of \( \delta \), we obtain the two following derivations:

\[
\begin{aligned}
(E_{\varphi}) & : \quad \forall y(\delta(m, y) \Rightarrow \varphi(y)) & (E_{\varphi}) & : \quad \forall y(\delta(m, y) \iff y = m') \\
(\varphi(m')) & : \quad \delta(m, m') \Rightarrow \varphi(m') & (\varphi(m')) & : \quad \delta(m, m') \iff m' = m' \quad m' = m'
\end{aligned}
\]

\[
(=) : \quad y = m' \Rightarrow \varphi(y)
\]

\[
(E_{=}) : \quad \forall y \delta(m, y) \iff y = m'
\]

\[
(I_{\varphi}) : \quad \forall y(\delta(m, y) \Rightarrow \varphi(y))
\]
which show that $T \vdash \psi \iff \varphi(m')$. Since $m' = \#\psi$, it follows that $T \vdash \psi \iff \varphi(\psi \psi \psi)$.

\[\Box\]

Theorem 8.12 If $T$ is a consistent theory such that all of the recursive functions are $T$-definable, then $T$ is essentially undecidable.

\textbf{Proof} Suppose that $T$ is decidable and let $\vartheta \in L_T[x]$ be the formula defining in $T$ the set $\text{th}_T$ (of encodings) of its theorems. Then:

- $n \in \text{th}_T$ if and only if $T \vdash \vartheta[x := n]$;
- $n \notin \text{th}_T$ if and only if $T \vdash \neg \vartheta[x := n]$.

The ‘if and only if’ comes from the consistency of $T$. Apply the lemma to $\neg \vartheta$ instead of $\varphi$. A proposition $\psi$ satisfying $T \vdash \psi \iff \neg \vartheta(\psi \psi \psi)$ is obtained, and hence the following contradiction: $T \vdash \psi$ if and only if $T \vdash \neg \vartheta(\psi \psi \psi)$ by the construction of $\psi$, if and only if $T \vdash \psi$ by the definition of $\vartheta$.

Once this general theorem is proven, the following important and well-known results easily follow.

\textbf{Theorem 8.13 (Church undecidability)} Elementary arithmetic is essentially undecidable.

\textbf{Proof} Follows from theorems 8.2 and 8.12. Hence Peano arithmetic is undecidable.

\[\Box\]

\textbf{Theorem 8.14 (Gödel incompleteness)} Every consistent and recursively axiomatizable extension of elementary arithmetic is incomplete.

\textbf{Proof} Follows from theorem 8.13 and proposition 8.9.

There is no hope of completing such a theory, e.g., Peano arithmetic, by adding a recursive set of axioms, i.e., by adding to $T$ one of $\varphi$ or $\neg \varphi$ if $T \nvdash \varphi$ and $T \nvdash \neg \varphi$, since the extended theory would remain recursively axiomatizable and consistent. Theorem 8.14 means that there exist propositions $\varphi$, necessarily true or false in $\text{N}$, but for which $\text{AP} \nvdash \varphi$ and $\text{AP} \nvdash \neg \varphi$. Rosser constructed such formulas, but artificially, as for lemma 8.11. In 1976 Paris and Harrington [116] gave a statement in the area of combinatorics, true in $\text{N}$ but unprovable in Peano arithmetic. There are now several ‘significant’ statements which fall in the same case: results on Goodstein sequences (Kirkby–Paris 1982 [78], Cichon 1983 [16]) and a finite form of Kruskal’s theorem (Friedman 1981, unpublished), among others. See also the remarks on the second incompleteness theorem (next page). The following theorem states the failure of ‘axiomatism’.

\textbf{Theorem 8.15} Complete arithmetic is not recursively axiomatizable.
8.2.6 The second incompleteness theorem

Gödel's second incompleteness theorem is even more famous than his first because of its epistemological interpretations. It deals with the consistency of arithmetic. Everyone knows that it is consistent, since N is a model. The proof of this fact is not done in arithmetic, but in general mathematics—it is obvious that all the theorems in this book were proven using mathematics—and to be formalist, in set theory. However, the consistency of arithmetic can be stated by an arithmetic formula, for example \( \neg \exists x \text{der}(x, "0 = 1") \). If this formula were provable in arithmetic, it would guarantee it an autonomous position, independent of the rest of mathematics. For the rest of mathematics, this problem is even more important. This question was crucial at the beginning of the century, to ensure a solid foundation for mathematics, which studies objects well beyond our intuition of concrete objects. The objective of Hilbert's program was a consistency proof of formalized mathematics—by finite means and mechanically verifiable. This stress on finite means recalls the old debate on the purity of methods and restricted means: in geometry, restricting constructions to the use of a ruler and a compass or banning coordinates from proofs, and in number theory, not using analytical means, among others.

Gödel's second incompleteness theorem states that a formula stating the consistency of Peano arithmetic is not derivable from its axioms. Its proof consists of formalizing the derivation of the first incompleteness theorem inside arithmetic. For computer science, the interest of this statement is not direct. It resides in an analysis of 'minimal' consistency proofs and particularly in proofs of other significant properties such as the normalization of the typed \( \lambda \)-calculus. What are the necessary supplementary means?

For example, the consistency of elementary arithmetic is provable in Peano arithmetic, thanks to induction over the integers. It is not necessary to use all of set theory to prove the consistency of Peano arithmetic. Gentzen’s proof uses induction over the ordinals up to \( \varepsilon_0 \). Induction up to \( \omega \) is one of the axioms of Peano arithmetic. It is sufficiently powerful to prove some transfinite induction principles \( TI(\alpha) \), i.e., up to an infinite ordinal \( \alpha \). Precisely, it allows the proof of \( TI(\alpha) \) for every ordinal \( \alpha < \varepsilon_0 \), but not \( TI(\varepsilon_0) \). Yet Gentzen used \( TI(\varepsilon_0) \) to prove the consistency of arithmetic.

8.3 Metaprogramming

The general results of the study of recursive functions can be interpreted naturally in programming terms, and even in metaprogramming, the area of computer science
dealing with general algorithmic constructions such as interpretation, compilation, verification and tests.

### 8.3.1 Universal predicates $T, U$

First consider the case of Turing machines. For each partial Turing-computable function, there exists a Turing machine $(\Gamma, Q, R)$ computing $f$, i.e. computing $f(n_1, \ldots, n_k)$ if $n_1, \ldots, n_k$ is placed on the tape. In computer science, since von Neumann it has been known that there 'exists' a unique machine, programmed (in the Turing machine sense) once and for all to which one passes as data both the instructions of the computation and the initial values of the variables $n_1, \ldots, n_k$: every compiler is an implementation for it. The same is true for Turing machines (Turing 1936 [151]): there exist universal Turing machines. What follows will proceed differently in order to remain inside logic's inference systems.

Recall that, by definition, a partial function $f : \mathbb{N}^k \to \mathbb{N}$ is recursive if and only if its graph $G(f)$ is r.e., hence the projection of a recursive subset $A$ of $\mathbb{N}^{k+2}$. By Gödel's theorem, $A$ is definable by a formula $\varphi \in \mathcal{L}_2[x_1, \ldots, x_{k+2}]$ in Peano arithmetic. Let $e = \# \varphi$ its numeric encoding.

By the definition of the graph $G(f)$, of $A$, of $\varphi$, of the relation $\text{der}$ and of the encoding $e$ of $\varphi$: $f(\overline{m}) = n$ if and only if there exists $c$ (a 'computation') such that $(\overline{m}, n, c) \in A$; if and only if there exists $c$ such that $AP \vdash \varphi(\overline{m}, n, c)$; if and only if there exist $c$ and $d$ (a 'derivation') such that $\text{der}(\#(\varphi(\overline{m}, n, c)), d)$; and if and only if there exist $c$ and $d$ such that $\text{der}(s^{k+2}(e, \overline{m}, n, c), d)$. A substitution function $s^{k+2}$ is used.

By encoding the pair $(c, d)$ by an integer $p$, it follows that there exists $p$ such that $\text{der}(s^{k+2}(e, \overline{m}, n, \pi^1(p)), \pi^2(p))$. Recall that $\text{der}$, $s^{k+2}$, $\pi^1$ and $\pi^2$ are recursive primitive. So, to compute $f(\overline{m})$ it suffices to find the smallest $n$—there is only one since $f$ is a partial function—and the smallest $p$ satisfying the preceding relation. All that has to be done is to encode the pair $(n, p)$ by an integer $q$, to minimize in $q$ and deduce $n = \pi^1(q)$.

\[
T^k(e, \overline{m}, q) = \text{der}(s^{k+2}(e, \overline{m}, n, \pi^1(q)), \pi^2(\pi^2(q)))
\]

\[
U(q) = \pi^1(q).
\]

Functions $T^k$ and $U$ are primitive recursive. The operator $T^k$ depends only on the encoding used for the syntactic operations of arithmetic. Only $f$ remains to be defined, using $T^k$, $U$ and the integer $e$: it has been shown that for every $k$-ary partial recursive function $f$, there exists an integer $e$ such that

\[
f(\overline{m}) = U(\mu q. [T^k(e, \overline{m}, q) = 1]).
\]

This is Kleene's normal form of partial recursive functions: the integer $e$ is called an index of $f$. 
From Kleene, write $\{e\}^k$ for the partial function $\vec{m} \mapsto U(\mu q \cdot [T^k(e, \vec{m}, q) = 1]$. Its domain is written $W^k_e$. It is the set of $\vec{m}$ such that there exists $q$ (a derivation or a computation) satisfying $T^k(e, \vec{m}, q)$. In the definition of $\{e\}^k(\vec{m})$, the integers $e$ and $\vec{m}$ are all arguments of $T^k$: the $(k + 1)$-ary partial function $u : (e, \vec{m}) \mapsto \{e\}^k(\vec{m})$ is therefore recursive. It is a universal partial recursive function in the sense that, for every partial recursive function $f : \mathbb{N}^k \rightarrow \mathbb{N}$, there exists $e$ such that $f(\vec{m}) = u(e, \vec{m})$. Note that for a given integer $e$, there is a partial function $\{e\}^k$ for each $k$. The $k$ indicating the arity is often omitted in practice, in particular when $k = 1$. The following theorem has been proven:

Theorem 8.16 (Kleene universality) There exists a primitive recursive function $U : \mathbb{N} \rightarrow \mathbb{N}$, and for each $k \geq 1$, a primitive recursive set $T^k : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ such that if one defines

$$\{e\}^k(\vec{m}) = U(\mu q \cdot [T^k(e, \vec{m}, q) = 1]),$$

then, for every partial recursive function $f : \mathbb{N}^k \rightarrow \mathbb{N}$, there exists an integer $e$ such that $f = \{e\}^k$.

The recursive functions were first defined using the composition, primitive recursion and (total) minimization operations. Then those operations were tentatively used to obtain—without success—the partial computable functions, since minimization is not effective over partial functions. The existence of a Kleene normal form shows that every partial recursive function is obtained from a primitive recursive function by a single application of minimization, then by composition with a primitive recursive function. In particular, it is not necessary to minimize partial functions.

This proof could have been written using Turing machines. First a universal Turing machine would have been constructed, hardly an interesting exercise. What plays the role of universal machine here is Peano arithmetic, already constructed, or any other theory of equal expressiveness. However, an encoding of arithmetic and of the substitution functions was used, and it was not presented explicitly. The approach chosen shows the links between logic and computation better.

One must understand the intuitive content of $T$ and $U$. In the logical approach $T^k(e, \vec{m}, q)$ states that $e$ is the encoding of a formula $\varphi$, $\vec{m}$ a $k$-tuple of values, and that $q$ encodes a derivation in Peano arithmetic, whose conclusion is of the form $\varphi(\vec{n}, \vec{c})$: $n$ is the value of $f(\vec{m})$. In the Turing machine approach, $T^k(e, \vec{m}, q)$ states that $e$ is the code of a Turing machine, $\vec{m}$ a $k$-tuple of values, and that $q$ encodes a computation executed on a universal machine with $e$ and $\vec{m}$ on the tape in the initial position: at the end of the computation the tape contains the value $f(\vec{m})$.

The mapping $n \mapsto \{n\}$ associating with an algorithm a function which computes it is obviously not injective. This corresponds to the idea that a function can be computed by an infinite number of different algorithms. The integer $n$ should be seen as encoding an algorithm as a real, finite object, while the partial function $\{n\}$ is an ideal, mathematical object. The equality $n = m$ of two
algorithms corresponds to the \textit{intensional} equality of functions \{\textit{n}\} and \{\textit{m}\}. The equality \{\textit{n}\} = \{\textit{m}\} of computed functions corresponds to the \textit{extensional} equality of algorithms: they compute the same function ‘in extension’.

Everything here can immediately be transcribed to r.e. sets: if \(A\) is an r.e. set, an \textit{index} of \(A\) is an integer \(e\) such that \(A = W_e\). In particular, an index of the semi-characteristic partial function \((1/2)_A\) is an index of \(A\). For each \(k\), the recursive set defined by \(T^k\) is \textit{universal}: for every r.e. set \(A \subseteq \mathbb{N}^k\), there exists an integer \(e\) such that

\[
A = W^k_e = \{\bar{m} \in \mathbb{N}^k ; (\exists q \in \mathbb{N})T^k(e, \bar{m}, q)\}.
\]

This result allows the enumeration of the partial recursive functions, and, consequently, of the r.e. sets: \(W_0, W_1, W_2, \ldots\).

Indices allow the statement of some problems. Let \(f\) be a partial recursive function, defined by a given algorithm \(e\). The formula \(\forall m \exists n T^k(e, m, n)\) states that it is total. It will be shown that the set of \(e\) for which this formula is true is not recursively enumerable. The problem here is: for a given \(e\), for example 6867, to know if the formula is provable. If this is the case \(\{e\}\) is called \textit{total provable} in Peano arithmetic.

There are recursive functions which are not total provable—since the proof of their termination must use an induction scheme stronger than the induction over integers. These functions are rarely encountered in programming, and a language which computed only the provably total functions would be sufficient: this is the case for certain typed \(\lambda\)-calculi such as Gödel’s system \(T\). The simplest example of a function which is not provably total is obtained by induction up to \(\varepsilon_0\), by iterating the construction which leads to Ackermann’s function, itself already complex enough.

\section*{8.3.2 Semantics of languages—interpreters}

Let \(L\) be a programming language computing all of the partial recursive functions, which is the case for usual languages. A way to give a semantics to \(L\) is to associate with each program \(\pi\) its interpretation \(\pi^N\) in \(\mathbb{N}\). Since every program contains only a finite set of variables, \(\pi^N : \mathbb{N}^* \rightarrow \mathbb{N}\) is the partial function computed by \(\pi\).

This idea applies to an arbitrary \(L\). The universality theorem transposes itself to \(L\): there is an encoding of programs of \(L\) by integers and the semantics \(\pi \mapsto \pi^N\) is a partial function \(S_L\) (or simply \(S\)) : \(\mathbb{N} \rightarrow (\mathbb{N}^* \rightarrow \mathbb{N})\), or \(\mathbb{N} \times \mathbb{N}^* \rightarrow \mathbb{N}\)

\[
(p, (m_1, \ldots, m_k)) \mapsto S_L(p, (m_1, \ldots, m_k)).
\]

This is the result of the \(p\)-th program with \(m_1, \ldots, m_k\) as initial values for the variables. This function \(S_L\) gives the execution of a program of \(L\) in a ‘machine-language \(L\)’. Actually, programs must be executed in a ‘universal machine’ \(U\), for which Kleene’s notation \(\{\}\) will be reserved, instead of \(S_U\). From the universality theorem, since \((p, m_1, \ldots, m_k) \mapsto S_L(p, (m_1, \ldots, m_k))\) is computable (partial
recursive), there exists an integer \( i \) such that

\[
S_L(p, (m_1, \ldots, m_k)) = \{i\}^{k+1}(p, m_1, \ldots, m_k).
\]

This integer \( i \) encodes a program in machine-language \( U \) which, executed on this machine with \( p, m_1, \ldots, m_k \) as input, yields the same result as the program \( p \) of \( L \) executed on \( L \) with \( m_1, \ldots, m_k \) as input.

In programming, \( i \) is interpreted from \( L \) to \( U \). The universality theorem therefore states the existence of interpreters.

### 8.3.3 Partial evaluation and compilation

A natural problem is that of partial evaluation of a computable function. Let \( p \) be an integer and consider the function \( \{p\}^{k+1} \). Fix the first \( k \) arguments \( m_1, \ldots, m_k \).

It is clear that the function

\[
(m_{k+1}, \ldots, m_{k+l}) \mapsto \{p\}^{k+l}(m_1, \ldots, m_k, m_{k+1}, \ldots, m_{k+l})
\]

is computable. The universality states the existence of \( p' \) such that this function is \( \{p\}' \). Program \( p' \) is computable as a function of \( p \): a text editor such as Emacs computes \( p' \) from \( p \) by substituting values to the identifiers. But a better partial evaluation of if \( m>0 \) then \( A \) else \( B \) with \( m = 4 \) gives \( A \) instead of if \( 4>0 \) then \( A \) else \( B \): this is the sort of thing done by a good optimizing compiler.

**Theorem 8.17 (s–m–n)** Let \( k \) and \( l \) be two integers \( \geq 1 \). There exists a primitive recursive function \( \rho_k : N^{l+1} \rightarrow N \) such that

\[
\{s\}^{k+l}(m_1, \ldots, m_{k+l}) = \{\rho_k(s, m_1, \ldots, m_k)\}^l(m_{k+1}, \ldots, m_{k+l}).
\]

**Proof** The notation used for the universality theorem will be used here:

\[
T^k(s, \bar{m}, q) = \text{der}(s, \bar{q}(q), \pi^1(q), \pi^2(q)), \pi^2(\pi^2(q)).
\]

The proof relies on a general property of Gödel's encoding concerning substitutions: for every substitution \( \sigma \), the function \( \#\varphi \mapsto \#(\sigma(\varphi)) \) is recursive primitive. To each formula \( \varphi \in L_\Sigma[x_1, \ldots, x_{k+l+2}] \) is associated \( \psi \in L_\Sigma[x_1, \ldots, x_{k+2}] \) by substitution,

\[
\psi \equiv \varphi[x_1 := m_1, \ldots, x_k := m_k, x_{k+1} := x_1, \ldots, x_{k+l+2} := x_{l+2}],
\]

and let \( \rho^{k,l} \) be the function associating \( \#\psi \) with \( (\#\varphi, m_1, \ldots, m_k) \). Then

\[
\varphi(m_1, \ldots, m_k) \equiv \psi(m_{k+1}, \ldots, m_{k+l}),
\]

from which

\[
\text{der}(s^{k+1+2}(s, m_1, \ldots, m_{k+l+1}, \pi^1(q), \pi^1(\pi^2(p)), \pi^2(\pi^2(p))).
\]

\[
\text{der}(s^{l+2}(\rho^{k,l}(s, m_1, \ldots, m_k), \pi^1(\pi^2(p)), \pi^1(q), m_{k+1}, \ldots, m_{k+l+1}, \pi^2(\pi^2(p))).
\]
It follows that

\[
T^{k+1}(s, m_1, \ldots, m_{k+1}, q) = T^k(\rho^{k,l}(s, m_1, \ldots, m_k, m_{k+1}, \ldots, m_{k+l}, q),
\{s\}^{k+l}(m_1, \ldots, m_{k+l}) = \{\rho^{k,l}(s, m_1, \ldots, m_k)\}^k(m_{k+1}, \ldots, m_{k+l}). \]

Apply the universality theorem again. Let \( e \) be an index of \( \rho \):

\[
\rho(s, m_1, \ldots, m_k) = \{e\}(s, m_1, \ldots, m_k).
\]

Program \( e \) is called a partial evaluator.

This result, important in recursive function theory, forms the basis for powerful methods of metaprogramming. Let \( e \) be a partial evaluator, i.e.,

\[
\{\{e\}(p, m)\}(m') = \{p\}(m, m').
\]

Define \( c \) by partially evaluating \( e \) with the interpreter \( i \), of \( L \) over \( U \), as input:

\[
c = \{e\}(e, i).
\]

Let \( s \) be a source program in \( L \). Compute

\[
o = \{c\}(s) = \{\{e\}(e, i)\}(s) = \{e\}(i, s),
\]

\[
\{o\}(m) = \{\{e\}(i, s)\}(m) = \{i\}(s, m) = S_L(s, m).
\]

It follows that \( c \) is a compiler from \( L \) to \( U \) and that \( o \) is the object form of \( s \). Now \( c \) can be obtained from \( i \) by letting

\[
g = \{e\}(e, e),
\]

\[
c = \{g\}(i),
\]

since

\[
c = \{\{e\}(e, e)\}(i) = \{e\}(e, i).
\]

So \( g \) is therefore a compiler generator, which produces a compiler from an interpreter. This method of compiler generation is used in several software engineering projects, in particular Danish and Japanese. The difficult aspects lie in the construction of a good partial evaluator.
8.3.4 Recursivity and reflexivity

The following theorem is one of the best known in recursive function theory. In programming, a syntactically recursive definition is of the form

\[ f(x) = \ldots f \ldots \]

How does one formalize this syntactic recursivity when it is interpreted effectively? It is in fact an equation that can be written

\[ f(x) = \Phi(f, x), \]

where \( \Phi \) is a 'functional', which must be computable. But the computable functionals have not been defined. The idea is to replace \( \Phi \) by a partial recursive function \( g \) and the argument \( f \) of \( \Phi \) by an index \( n \) of \( f \), so that this equation is transformed into

\[ \{n\}(x) = g(n, x). \]

This equation is in \( n \), but once a solution \( n \) has been found, the function \( f = \{n\} \) is defined explicitly by \( f(x) = g(n, x) \): there is no more 'recursivity'. In particular, \( g \) can be defined using a partial recursive function \( (n, x) \mapsto \{n\}(x) \), yielding the standard case of recursion where \( g \) contains 'recursive calls' \( f(\ldots) \).

This formulation has larger scope: \( g \) can call \( f \) not only extensionally, but also intensionally by using the algorithm computing \( f \) globally, represented by its index \( n \), i.e., the code of the program. For example, a solution \( n \) of the equation \( \{n\}(x) = n \) is a program which gives as output its own text, whatever the input.

Its proof uses the s-m-n theorem.

**Theorem 8.18 (Kleene recursivity)** Let \( g \) be a partial recursive function from \( \mathbb{N}^k \) to \( \mathbb{N} \). There exists \( n \in \mathbb{N} \) such that \( \{n\}^k(x_1, \ldots, x_k) = g(n, x_1, \ldots, x_k) \) for every \( \vec{x} \).

**Proof** Let \( \rho : \mathbb{N}^2 \to \mathbb{N} \) be the recursive function given by theorem 8.17. Consider the partial recursive function:

\[ (t, x_1, \ldots, x_k) \mapsto g(\rho(t, t), x_1, \ldots, x_k). \]

From the universality theorem, there exists \( m \in \mathbb{N} \) such that

\[ g(\rho(t, t), x_1, \ldots, x_k) = \{m\}^{k+1}(t, x_1, \ldots, x_k). \]

By the definition of \( \rho \),

\[ \{m\}^{k+1}(t, x_1, \ldots, x_k) = \rho(m, t)^k(x_1, \ldots, x_k). \]

Let \( n = \rho(m, m) \). Then \( g(n, x_1, \ldots, x_k) = \{n\}^k(x_1, \ldots, x_k) \). \( \square \)

This theorem evidently applies to recursive definitions of functions, but also to reflexive constructions, common in functional and object-oriented programming, where data structures belonging to the interpreter are accessible at the program level.
8.3.5 Undecidability of $\beta$-conversion

This is the first undecidability result, proven in 1936 by Church [15], from which he proved the undecidability of the predicate calculus. A more general version due to D. Scott (1963, unpublished) will be presented here. Its proof closely resembles those of the Rice and Church theorems, which also use a fixpoint lemma and an encoding.

Suppose there is an encoding $M \mapsto \#M$ of $\lambda$-terms, with recursive functions $\text{app} : \mathbb{N}^2 \to \mathbb{N}$ and $\text{num} : \mathbb{N} \to \mathbb{N}$ such that $\#(MN) = \text{app}(\#M, \#N)$ and $\#n = \text{num}(n)$. Let $rM = m$ if $m = \#M$.

Lemma 8.19 For every combinator $G$, there exists $X$ such that $G^rX^r = X$, i.e., if $\#X = p$, then $G^p = X$.

Example For $G \equiv I$, there exists $X$ such that $rX = X$. □

Proof The functions $\text{app}$ and $\text{num}$ are $\lambda$-definable by terms $\text{app}$ and $\text{num}$. Hence $rM = r^rM$ and $\text{app}r^rM^rN^r = r^rMN^r$.

Let $\omega \equiv \lambda x. G(\text{app}x(\text{num}x))$ and $X \equiv r^r\omega^r$. Then

\[
X \equiv \omega^r \omega^r \rightarrow G(\text{app}r^r\omega^r(\text{num}r^r\omega^r)) = G(\text{app}r^r\omega^r r^r\omega^r) = G(r^r\omega^r r^r\omega^r) = X.
\]

□

Definition 8.11 Two subsets $A_0$, $A_1$ of $\mathbb{N}$ are recursively separable if there exists a recursive function $f : \mathbb{N} \to \{0, 1\}$ separating $A_0$ and $A_1$, i.e., such that $f(n) = i$ if $n \in A_i$, for $i = 0, 1$.

Two sets of terms are recursively separable if their sets of encodings are recursively separable.

Theorem 8.20 Two sets of terms, non-empty and stable under $\beta$-conversion, are not recursively separable.

Proof Suppose $A_0$ and $A_1$ are separated by the recursive function $f$, and let $F$ be a term defining $f$. If $A_0$ and $A_1$ are non-empty, let $M_0 \in A_0$ and $M_1 \in A_1$. Let $G \equiv \lambda x. DM_1 M_0(Fx)$: by the definition of the combinator $D$, if $M \in A_0$, $G^rM^r = M_1$ and if $M \in A_1$, then $G^rM^r = M_0$. By lemma 8.19, there exists $X$ such that $G^rX^r = X$. If $X \in A_0$, then $X = G^rX^r \in A_1$, by the stability of $A_1$. If $X \in A_1$, then $X = G^rX^r \in A_0$, by the stability of $A_0$. Contradiction! □

Corollary 8.21
- For fixed $A$, the set $\{M; M = A\}$ is not recursive.
- The relation $M = N$ is not recursive.
- The set of normalizable terms is not recursive.
8.3.6 Decidability problems in programming

It is still assumed that the programming language is sufficiently powerful to compute all the partial computable functions.

The halting problem
Given: a pair \((p, m)\) formed of a program \(p\) and data \(m\).
Problem: does program \(p\), with \(m\) as input, halt?

Intuitively, it should be clear that the problem is semi-decidable: it suffices to run the program, and give a positive answer when it halts. The set of its solutions is

\[
\{(p, m); \{p\}(m) \downarrow \} = \{(p, m); m \in W_p\}.
\]

A more particular problem is obtained through diagonalization. The set of its solutions is

\[
K = \{n; \{n\}(n) \downarrow \} = \{n; n \in W_n\}.
\]

Since the partial function \(n \mapsto \{n\}(n)\) is recursive, the set \(K\) is r.e. It will be shown that it is not recursive, using a standard diagonalization method. Let

\[
f(n) = \begin{cases} 
\{n\}(n) + 1 & \text{if } n \in K \\
0 & \text{otherwise.}
\end{cases}
\]

This function is total and is not recursive if \(K\) is. Furthermore it is distinct from all the partial recursive functions since \(f(n) \neq \{n\}(n)\) for every \(n\). Hence \(K\) cannot be recursive.

Rice’s theorem follows easily from the Kleene recursion theorem.

Theorem 8.22 (Rice) Let \(I_E = \{x \in \mathbb{N}; \{x\} \in E\}\), where \(E\) is a set of partial recursive functions of one variable. The set \(I_E\) is recursive if and only if \(E\) is empty or equal to the set of partial recursive functions.

Proof Suppose \(I_E\) is recursive. Suppose that \(E\) is non-trivial. There exist partial recursive functions \(f, g : \mathbb{N} \rightarrow \mathbb{N}\) such that \(f \in E, g \notin E\). Let

\[
h(x, y) = \begin{cases} 
g(y) & \text{if } x \in I_E \\
f(y) & \text{if } x \notin I_E.
\end{cases}
\]

\(g, f\) and \(I_E\) being recursive, \(h\) is partial recursive. From theorem 8.18, there exists \(n \in \mathbb{N}\) such that \(\{n\}(y) = h(n, y)\) for every \(y\).

If \(n \in I_E\), then \(\{n\}(y) = g(y)\), so \(n \notin I_E\).

If \(n \notin I_E\), then \(\{n\}(y) = f(y)\), so \(n \in I_E\).

Contradiction. \(\square\)
Examples This theorem proves the non-existence of decision procedures for the following problems involving programs $p, q$ of one variable:

- Does $p$ halt for all input?
- Does $p$ not halt for any input?
- Does $p$ compute a primitive recursive function?
- Does $p$ compute a function of finite domain?
- Do $p$ and $q$ compute the same function?

The problems of partial (respectively total) correctness of a program with respect to a condition $(\varphi, \psi)$ are also undecidable since certain particular cases appear among the above problems.

Rice's theorem can also be stated for sets instead of functions, with $E$ a set of r.e. subsets of $\mathbb{N}$, and $I_E = \{n; W_n \in E\}$.

The characterization of r.e. sets is more delicate. It uses an effective enumeration of finite sets: for example, to the finite set $\{a_1, \ldots, a_n\}$ is associated the integer $2^a_1 + \ldots + 2^a_n$. It is a bijection from the set of finite subsets of $\mathbb{N}$ to $\mathbb{N}$. Write $D_n$ for the finite set encoded by the integer $n$ and $U(D_n)$ for the set of r.e. sets containing $D_n$. It is an open neighborhood of $D_n$ in an appropriate topology. The following theorem is stated without proof.

**Theorem 8.23 (Rice–Shapiro)** Let $E$ be a set of r.e. subsets of $\mathbb{N}$. The set $I_E$ of its indices is r.e. if and only if $E$ is empty or there exists a recursive function $f$ such that $E = \bigcup_{n \in \mathbb{N}} U(D_{f(n)})$. The set $E$ is an effective union of open neighborhoods.

In other words, $A \in E$ if and only if there exists $n$ such that $A \supseteq D_{f(n)}$. In particular, it is necessary that if $A \in E$ and $A \subseteq B$, then $B \in E$. It is easy not to satisfy this condition.

**Example** Let $E = \{\emptyset\}$; then $I_E = \{n; W_n = \emptyset\}$ is not r.e. and the problem $A = \emptyset$ is not semi-decidable.

### 8.3.7 Equation resolution

If $P(X_1, \ldots, X_k; Y_1, \ldots, Y_l)$ is a polynomial with coefficients in $\mathbb{Z}$, the set

$$R = \{(x_1, \ldots, x_k) \in \mathbb{N}^k; \exists (y_1, \ldots, y_l) \in \mathbb{N}^l, P(x_1, \ldots, x_k, y_1, \ldots, y_l) = 0\}$$

is called a diophantine set. As a projection of the recursive set $P = 0$, $R$ is recursively enumerable.

Among the famous problems stated by Hilbert in 1900 [63], the tenth was called the 'Determining the solvability of a diophantine equation’. It proposed to 'find a process under which one can determine in a finite number of operations if the
equation is solved for rational integers’. That was in 1900, and before Gödel and Turing, the first explicit statement of a decidability problem. It was solved in 1970 by Matijasevič [105] by a surprising theorem:

**Theorem 8.24 (Matijasević)** Each r.e. set is diophantine.

Its proof is difficult, the key step consisting of proving that the relation $x = y^x$ is diophantine. Since there exist r.e. sets, there exist non-recursive diophantine sets, hence undecidable diophantine equations.

**Unification** The unification problem for first-order terms is decidable. This is an important result which shows the importance of the first-order to computation. In 1981 Goldfarb [58] showed that the unification problem for second-order terms is undecidable, where second-order terms are formed of function variables which can be substituted for terms.

Unification modulo an equational theory $E$ was mentioned in Chapter 5: find $\sigma$ such that $\sigma(M_1) \equiv E \sigma(M_2)$. Here are examples of different behaviors which also show the difficulty of equational unification:

- The associative–commutative case is decidable and leads to a complete, finite set of unifiers (Stickel 1975 [142]).
- The associative case is decidable, but there does not exist a complete, finite set of unifiers.
- The distributive–associative case is undecidable (Szabo 1982 [145]).

### 8.3.8 Computable reals

Recursive function theory, condemned to handle only finite objects such as integers and words, seems inadequate for one of the most useful applications of computer science, numerical computation. Of course, the ‘floating point numbers’ of programming are only decimals, almost integers, finitely representable by a machine. However, the mathematical methods using numerical computations are those of analysis, not of arithmetic! So what is the effective content of the approximations of the continuous by the discrete?

To characterize the computable reals, the basic concrete objects are the rational numbers, which are easily coded by integers. To the integer $n$ is associated the rational $(\pi_3(n) - \pi_0(n))/\pi_3(n) + 1$. The rational $p/q + 1$ (with $p \in \mathbb{Z}$, $q \in \mathbb{N}$) is coded by the integer $\langle p, 0, q \rangle$ if $p \geq 0$, or by $\langle 0, -p, q \rangle$ otherwise. If such an encoding were adopted, one could refer to recursive functions from $\mathbb{N}$ to $\mathbb{Q}$ or to $\mathbb{Q} \times \mathbb{Q}$.

**Definition 8.12** The recursive functions $r, s : \mathbb{N} \to \mathbb{Q}$ define a **recursive real** if the sequence of intervals $[r(n), s(n)] \subseteq \mathbb{Q}$ is decreasing and $\lim_{n \to \infty} |s(n) - r(n)| = 0$. 


Even though it is not necessary to construct $R$ beforehand, the recursive reals that these functions define are the intersections of the intervals $[r(n), s(n)]$ in $R$.

A recursive real is an abstract object if it is compared to all reals. It is much more concrete than the latter from an operational point of view, as is the program-computing the interval $[r(n), s(n)]$. It is easy to show that the sum and the product of two recursive reals and the inverse of a non-null recursive real are recursive. It follows that the set $R_{\text{rec}}$ of recursive reals is a subfield of $R$. It contains $Q$ and is countable, like the set of recursive functions. It also contains the algebraic reals (real roots of polynomial equations with coefficients in $Q$) and even certain transcendental numbers such as $e$ and $\pi$ for which approximation methods are known. By working a little harder, one can show that every positive recursive real has a square root in $R_{\text{rec}}$ and that every polynomial of odd degree with coefficients in $R_{\text{rec}}$ has a root in $R_{\text{rec}}$. These properties mean that $R_{\text{rec}}$ is a closed real field, just as $R$ and the field of algebraic reals are. One of Tarski's famous theorems states that the theory of closed real fields is complete. It follows that $R$ and $R_{\text{rec}}$ are elementarily equivalent, i.e., satisfy exactly the same propositions. The recursive reals are therefore indistinguishable from the reals from the point of view of global algebraic properties. Unfortunately, these reals are not well suited to effective analysis; see exercise 21 on page 361, which shows that $R_{\text{rec}}$ is not complete.

*Example* The equality of two recursive reals is not decidable, which is not surprising, since equality here is an extensional concept. Suppose that the equality were decidable; this would imply the decidability of the halting problem. A 'counter' $c$ is used, initialized to 1 and capable of holding a rational. At each step of the program, $c/2$ is added to $c$, which produces a Cauchy sequence defining a recursive real $c^*$. The program does not halt if and only if $c^* = \sum_{n=0}^{+\infty} 1/2^n = 2$, which reduces the halting problem to the equality problem in $R_{\text{rec}}$. The method of problem reduction will be studied more below.

### 8.4 A classification of problems

How does one rate the difficulty of a problem? Decidability is a problem, but it is not the only one. Among decidable problems, the last section of this chapter will study the analysis of the complexity of decision algorithms. Among undecidable problems, it is also useful to classify them. It is useful to know not only the level of difficulty of a problem, but also to have general ways of measuring this level.

#### 8.4.1 The arithmetic hierarchy

The decidable and semi-decidable problems have already been distinguished. More generally, the arithmetic hierarchy allows one to classify problems according to
global criteria. Recall that the set of recursive sets is closed under intersection, union and complement, but not under projection. By definition, the r.e. are projections of recursives. The class of r.e. is closed under intersection, union and projection, but not under complement. The complements of r.e. sets are called co-r.e.:

The two projection and complement operations can be iterated alternately to generate two sequences of classes of sets, $\Sigma_n$ and $\Pi_n$:

\[
\begin{align*}
\Sigma_0 &= \Pi_0 = \text{class of recursive sets}, \\
\Sigma_{n+1} &= \text{class of projections of elements of } \Pi_n, \\
\Pi_n &= \text{class of complements of elements of } \Sigma_n \text{ (i.e. of the co-Sigma)}, \\
\Delta_n &= \Sigma_n \cap \Pi_n.
\end{align*}
\]

In particular, $\Sigma_1$ is the class of r.e., $\Pi_1$ of the co-r.e. A set is arithmetic if it belongs to one of these classes.

Recall that the recursive sets are definable in Peano arithmetic. It follows that they are definable in $\mathbb{N}$. A subset $A$ of $\mathbb{N}^k$ is defined in the interpretation $\mathbb{N}$ by $\varphi$ if

\[
A = \{ (m_1, \ldots, m_k) \in \mathbb{N}^k; \varphi^\mathbb{N}(m_1, \ldots, m_k) = 1 \}.
\]

This semantic definability of the recursives extends to the r.e. and beyond. It is easily proven that every arithmetic set is definable in $\mathbb{N}$ by a first-order formula: each arithmetic set can be defined in the form

\[
\{ (x_1, \ldots, x_k) \in \mathbb{N}^k; Q_0 y_0 \ldots Q_p y_p R(x_1, \ldots, x_k, y_0, \ldots, y_p) \},
\]

where $R$ is recursive, $Q_1 \ldots Q_p$ is a sequence of alternating quantifiers, the first being $\exists$ if the set is $\Sigma_p$, and $\forall$ if it is $\Pi_p$. For example, the sets $\Pi_2$ are of the form $\forall \exists \forall R$. Using standard coding techniques, a sequence of $n$ identical quantifiers can be reduced to a single quantifier over an $n$-tuple. For example, $\exists y_0 \exists y_1 R(y_0, y_1)$ becomes $\exists y R'(y)$, where $R(y_0, y_1) = R'(y_0, y_1)$. Furthermore, the quantifiers are naturally introduced by projection and complement. Let $A \subseteq \mathbb{N}^k$, $\varphi$ be a formula which $\mathbb{N}$-defines the projection of a set $B \subseteq \mathbb{N}^{k+1}$, and $\bar{A} = \mathbb{N}^k \setminus A$: from

\[
B = \{ (\bar{n}, p) \in \mathbb{N}^{k+1}; \varphi^\mathbb{N}(\bar{n}, p) = 1 \},
\]

deduce

\[
A = \{ \bar{n} \in \mathbb{N}^k; \exists p \varphi^\mathbb{N}(\bar{n}, p) = 1 \}.
\]

Hence $A$ is $\mathbb{N}$-defined by $\exists x_{k+1} \varphi$, and

\[
\bar{A} = \{ \bar{n} \in \mathbb{N}^k; \forall p \varphi^\mathbb{N}(\bar{n}, p) = 0 \},
\]

hence $\bar{A}$ is $\mathbb{N}$-defined by $\forall x_{k+1} (\neg \varphi)$.

By the Matijasevič theorem, every r.e. set is the set of integer solutions to a diophantine equation, so a finer result is obtained: a set is $\Sigma_n$ if and only if it is
definable by a prenex formula, starting with $\exists$, and containing $n$ alternations of quantifiers.

The important result is Kleene’s hierarchy theorem, which follows by diagonalization from the universality result of exercise 24:

**Theorem 8.25** (Kleene hierarchy) Beyond $\Delta_1$ all inclusions are strict.

**Proof** Let $U_n \subseteq \mathbb{N}^2$ be a universal set $\Sigma_n$, $n \geq 1$. Define $A = \{m \in \mathbb{N}; (m, m) \in U_n\}$. Then $A$ is $\Sigma_n$, hence $\bar{A}$ is $\Pi_n$. If $\bar{A}$ were also $\Sigma_n$, then by universality there would exist $e$ such that $\bar{A} = \{m \in \mathbb{N}; (e, m) \in U_n\}$. Then $e \notin A$, a contradiction. Hence $\Sigma_n \neq \Pi_n$ and $\Delta_n \neq \Sigma_n$. $\square$

There is therefore a hierarchy of arithmetic sets which transposes to a hierarchy of problems. The problem $n \in A$? is decidable, semi-decidable or co-semi-decidable if $A$ is recursive, r.e. (or $\Sigma_1$), or co-r.e. (or $\Pi_1$). Beyond, the problem $n \in A$? cannot be touched by machines. Concrete examples of these problems will be given in §8.4.4. Are there non-arithmetic sets? A simple example is given by the set (of encodings) of true arithmetic formulas in $\mathbb{N}$: this is Tarski’s theorem (see exercise 25).

To place a set in the arithmetic hierarchy is generally quite easy.

**Example** Let $R = \{p; W_p = \mathbb{N}\}$ be the set of programs computing recursive functions. It suffices to write that

$$R = \{p; \forall m (m \in W_p)\} = \{p; \forall m \exists q T(p, m, q)\}.$$

Since $T$ is recursive and its prefix is $\forall \exists$, deduce that $R$ is $\Pi_2$. It is more difficult to show that $R$ is not $\Sigma_2$. $\square$

Natural problems are found among the first levels of the hierarchy, for example those mentioned around Rice’s theorem. However, even though the hierarchy is strict, practically no ‘natural’ problems are found above level four. This seems to be linked to the difficulty of formulating (and intuitively understanding) properties containing a high number of alternating quantifiers. Mathematicians generally introduce definitions to designate important properties, e.g., continuity for $\forall \epsilon \exists \delta$, or even new objects to ‘reset to 0’ the number of alternations to facilitate future developments.

While the $\mathcal{M}_p$ model of a definite logic program is r.e., after an encoding of $A_{\Sigma}$, Apt and Blair have shown that the standard model of a general program ‘with $n$ strata’ is $\Sigma_n$: this result shows the gap between a definite and even a restricted class of general programs.
8.4.2 Reductions

There are other ways to measure the difficulty between problems, by studying the relative difficulty of two problems. Science often progresses when a solution to a difficult problem is discovered. What more is known when a problem is solved? One can measure the importance of a solution by the class of other problems which become solved. The relations between problems are formalized by the concepts of reduction. There are two kinds of reduction. One allows the transformation of the solution of a problem into the solution of other (easier) problems. The other allows an extension of the available means, such as the use of a solution of instances of a solved problem as a supplementary means.

To reduce a problem to another is standard practice: a geometry problem is often reduced to an algebra problem. Each concept of reduction is defined as a preorder \( \leq \) over a set of subsets of \( \mathbb{N} \), \( A \leq B \) meaning that the problem \( x \in A \) reduces to \( x \in B \): one can consider that \( x \in B \) is more difficult, or more general than \( x \in A \). Associate with the preorder \( \leq \) the equivalence relation \( \sim \) that it generates, i.e. \( A \sim B \) if and only if \( A \leq B \) and \( B \leq A \): \( A \sim B \) states that the problems \( x \in A \) and \( x \in B \) each reduce to the other, and hence are equivalent problems and of the same difficulty. Furthermore, the maximal elements of this preorder are useful reference points. More precisely, let \( \leq \) be a reduction relation and \( \mathcal{E} \) a set of problems. A problem is called \( \mathcal{E} \)-difficult for \( \leq \) if it is an upper bound for all the elements of \( \mathcal{E} \), and \( \mathcal{E} \)-complete for \( \leq \) (or \( \leq \)-complete in \( \mathcal{E} \)) if it belongs to \( \mathcal{E} \) and is \( \mathcal{E} \)-difficult. This last concept is due to Post.

**Proposition 8.26**

1. If \( A \) and \( B \) are \( \mathcal{E} \)-complete, then \( A \sim B \).
2. If \( A \) is \( \mathcal{E} \)-complete and if \( B \in \mathcal{E} \), then \( B \) is \( \mathcal{E} \)-complete if and only if \( A \leq B \).

These reductions are useful, since one can easily prove that a problem is undecidable by showing that an undecidable problem can be reduced to it. It is therefore useful to have a catalogue of already classified problems which serve as a reference point.

8.4.3 Turing reduction (or relative reducibility)

Since the beginning of the chapter it has been assumed that the objects being handled in a computation are all given effectively: integers, words, etc. The addition of two integers is an effective operation: if \( a \) and \( b \) are two given integers (for example 36 and 5), then \( a + b \) can be computed effectively. If \( f \) and \( g \) are two given functions from \( \mathbb{N} \) to \( \mathbb{N} \), consider the function \( h = f + g : n \mapsto f(n) + g(n) \). To compute \( h(5) \), the values of \( f(5) \) and \( g(5) \) must first be obtained, then they must be added, this last operation being effective.
In a certain sense, the addition of two integers and the addition of two functions are equally effective operations, even though the first handles finite objects, integers, and the second handles infinite objects, functions. However, note that the way in which elements are ‘given’ must be made precise. There are not many ways to give an integer. There are, however, many ways, standard in programming, to give a function $f$: as the text of a program computing it, or as the call to an ‘external routine’ producing the value $f(n)$ on demand, when $n$ is supplied. In one case, the (source) text of the program is a finite object and it is known how $f$ is computed. In the other, only a finite number of useful values is known for $f$, and these are supplied by the external routine: a library, or in an interactive manner by a user. In the latter case, it is not even necessary that $f$ be effectively computable. So effective operations can be done on non-effective objects: computation is a transformation, and the effectiveness is relative to what is given. Turing was the first to approach computability in a relative manner by allowing the use of an ‘oracle’ during a computation.

The relation $A \leq_T B$ states that $A$ is recursive relative to $B$: during the computation of the function $1_A$, the use of an oracle is allowed: it gives a value of $1_B(x_0)$ when the latter is needed for an arbitrary value of $x_0$. No effectiveness assumption is made on this oracle, hence the name, but it can only be called a finite number of times, at certain steps of a computation. Turing reducibility can be stated in terms of elementary instructions of a machine. If the problem $x \in A$? can be solved, and if one adds to a Turing machine the test $x \in A$? as elementary instruction, then a new class of problems $x \in B$? can be solved: the class of $B \leq_T A$.

Example A set is always recursive relative to itself and to its complement. □

If $B$ is recursive, this oracle can be replaced by a DTM, in which case $A$ is recursive. The converse is used for undecidability proofs.

The concept of oracle is intuitive but may not appear rigorous. The precise definition of relative reducibility uses functionals rather than functions. Let $N^{k,1} = N^k \times (N^N)!$. A partial function from $N^{k,1}$ to $N$ is called a partial functional. To simplify, suppose $l = 1$ and write $N^k$ instead of $N^{k,1}$:

- A subset $A$ of $N$ is semi-recursive if there exists a recursive subset (in the ordinary sense) $B$ of $N^{k+2}$, such that $(m_1,...,m_k,f) \in A$ if there exists $p \in N$ such that $(m_1,...,m_k,p,f^*(p)) \in B$, where $f^*$ is the history of $f$.
- A partial functional $F : N \to N$ is recursive if its graph, which is a subset of $N^{k+1,1}$, is semi-recursive.
- A subset of $N$ is recursive if its characteristic function is recursive.
- Finally a function $f : N^k \to N$ is recursive relative to a function $g \in N^k \to N$, written $f \leq_T g$, if $f(\bar{m}) = F(\bar{m}, g)$ for a partial recursive functional $F$. If $A$ and $B$ are two subsets of $N^k$, $A$ is called recursive relative to $B$, written $A \leq_T B$, if $1_A \leq_T 1_B$.

Now that these definitions have been given, it has to be shown that they cover the intuitive approach to computations using oracles. Consider the problem $(\bar{m}, f) \in$
A?, where \( A \subseteq \mathcal{N} \) is a semi-recursive subset. To solve it, use \( B \subseteq \mathcal{N}^{k+2} \), recursive. For \( p \geq 0 \), the \( p \)-th step consists of obtaining the value of \( f^*(p) \) from an ‘oracle’, then to compute \( 1_B(m_1, \ldots, m_k, p, f^*(p)) \). If the computed value is 1, return 1, otherwise start over with \( p + 1 \).

This reduction is very important in logic. The associated equivalence classes are called *degrees of unsolvability* (Kleene and Post 1954 [83]). The ordered set of these degrees has surprising properties, which have been the object of original methods (Post, Friedberg, Muchnik, Sacks, etc.). The set of degrees has a least element 0 formed of all the recursive sets (it is in particular the degree of \( \emptyset \) and of \( \mathbb{N} \), these being \( \leq_T \) to any set). Another degree is well known: the degree 0' of the r.e. set \( K \) of the halting problem. It was thought for a long time that every r.e. set was of degree 0 (if it was recursive) or 0'. In fact all the undecidability proofs consisted of showing \( K \leq_T A \) (or even, which is stronger, \( K \leq_m A \)): in 1943 this led Post [123] to try to construct intermediate degrees between 0 and 0'. This problem was solved in 1956 by Muchnik [112] and Friedberg [44] independently by original techniques, thereby inaugurating a specialized new branch of logic. There exists a non-recursive r.e. set \( E \) such that \( K \not\leq_T E \), hence whose degree is strictly between 0 and 0'. There even exists an uncountable number of r.e. degrees, incomparable under \( \leq_T \). In 1964 Sacks [130] showed that the order over r.e. degrees between 0 and 0' is dense. From these theorems, there exists an infinite number of ‘distinct’ semi-decidable problems in the Turing reduction sense, while all the usual problems can be found in the degrees 0 and 0'. The moral of the story is that it is very difficult to imagine complicated problems.

### 8.4.4 Strong reduction

In practice, to reduce a problem to another one is to transform its statement. The idea is to let \( A \leq B \) if there is a sufficiently simple transformation \( f \) such that \( x \in A \) if and only if \( f(x) \in B \). Since this general definition depends on the class of accepted transformations, there are several concepts of strong reduction: recursive transformation, injective recursive transformation and polynomial transformation.

*Definition 8.13* The preorder \( \preceq_m \) is defined by: \( A \preceq_m B \) if there exists a recursive function \( f \) such that \( x \in A \) if and only if \( f(x) \in B \).

This reduction is stronger than that of Turing, since \( A \preceq_m B \) implies \( A \leq_T B \): the computation of \( 1_A(x) \) is reduced to that of \( f(x) \) by a DTM and a call to an oracle for \( 1_B(f(x)) \). If \( B \) is r.e., then \( A \) is as well: this property is not true if one only has \( A \leq_T B \).

Consider once again the set

\[
K = \{n; \{n\}(n) \downarrow\} = \{n; n \in W_n\}
\]

of the halting problem.
Theorem 8.27  1. \( K \) is r.e.
  2. \( K \) is not recursive.
  3. \( K \) is \( \leq_m \)-complete for the class of r.e.

Proof  (1) \( n \mapsto \{n\}(n) \) is partial recursive, hence by proposition 8.5, its domain is r.e.

(2) If \( K \) were recursive, \( \mathbb{N}\backslash K \) would be r.e. (proposition 8.6) from which \( \mathbb{N}\backslash K = \{x \in \mathbb{N}; \{p\}(x) \downarrow\} \) for a \( p \in \mathbb{N} \) (universality theorem). Then \( p \in K \) if and only if \( p \notin K \). Contradiction!

(3) Let \( A \) be an r.e. set. Then \( A = \{x \in \mathbb{N}; f(x) \downarrow\} \), where \( f \) is partial recursive. Introduce the partial recursive function \( g(x, y) = f(x) \).

There exists \( m \in \mathbb{N} \) such that \( g(x, y) = \{m\}(x, y) \) (universality). From the s–m–n theorem: \( \{m\}(x, y) = (\rho(m, x))(y) \). The function \( x \mapsto \rho(m, x) \) is recursive and:

\[
A = \{x \in \mathbb{N}; \rho(m, x) \in K\},
\]

from which \( A \leq_m K \). \( \Box \)

It can be shown that

- \( \{n \in \mathbb{N}; W_n = \mathbb{N}\} \) is \( \Pi_2 \)-complete;
- \( \{n \in \mathbb{N}; W_n \text{ finite}\} \) is \( \Sigma_2 \)-complete;
- \( \{n \in \mathbb{N}; W_n \text{ infinite}\} \) is \( \Pi_2 \)-complete;
- \( \{n \in \mathbb{N}; W_n \text{ cofinite}\} \) is \( \Sigma_3 \)-complete;
- \( \{n \in \mathbb{N}; W_n \text{ recursive}\} \) is \( \Sigma_3 \)-complete.

A set is strictly \( \Pi_n \) (respectively \( \Sigma_n \)) if it belongs to the class \( \Pi_n \) (respectively \( \Sigma_n \)) and does not belong to the class \( \Sigma_n \) (respectively \( \Pi_n \)). The hierarchy theorem implies that each of the preceding sets constitutes an example of a set strictly in its class.

Example  It has already been proven that \( R = \{n \in \mathbb{N}; W_n = \mathbb{N}\} \), the set of indices of recursive functions, is \( \Pi_2 \). We show that it is \( \Pi_2 \)-complete for \( \leq_m \). Let \( A \) be an arbitrary \( \Pi_2 \) set, hence of the form \( \{x; \forall n((x, n) \in B)\} \), where \( B \) is a r.e. Let \( e \) be an index of \( B\): \( (x, n) \in B \) if and only if \( \{e\}(x, n) \downarrow\). By the s–m–n theorem, \( \{e\}(x, n) = (\rho(e, x))(n) \). Let \( f(x) = \rho(e, x) \). Then \( (x, n) \in B \) is equivalent to \( n \in W_{f(x)} \). Deduce that \( x \in A \) is equivalent to \( \forall n(n \in W_{f(x)}) \), i.e., \( W_{f(x)} = \mathbb{N} \), or else \( f(x) \in R \). This shows that \( A \leq_m R \), using \( f \). \( \Box \)

The proof of the other properties is not immediate. Hence, these sets are neither r.e. nor co-r.e. There does not even exist a semi-decision procedure to determine whether a program terminates (or not). Such problems, crucial in programming, are the subject of human activity, as is most of mathematics.

This is also the case for programming, where one generally defines an ‘algorithm’ as a procedure which always terminates. If one accepts this definition, then one must accept the fact that this set of algorithms is \( \Pi_2 \), hence the simple problem
of determining if something is an algorithm has no effective solution. Algorithm analysis is not a theory of algorithms in the same way as arithmetic is a theory of numbers. This analysis studies interesting algorithms by developing general methods such as construction methods and complexity analysis, and restricted classes of algorithms of which none attain the heights of complexity that this hierarchy allows. In fact, the ceiling is more that of efficiency, which will be studied in the final section.

The situation is analogous to that in mathematics, where one looks for interesting theorems, and not for everything which can be deduced from the axioms. Besides, the axioms never have an extraordinary syntactic complexity: they are simply there because they serve to model useful objects.

These concepts of reduction often allow the ‘transfer’ of undecidability results. For example, in 1955 Novikov proved (in an article [114] of 153 pages!) that the word problem for groups is undecidable. In 1958 Markov [98] showed how to associate with each finite presentation of a group \( G \) a topological manifold \( V(G) \) of dimension 4 whose homotopy group is \( G \), in such a way that \( G_1 \) is a group isomorphic to \( G_2 \) if and only if the manifolds \( V(G_1) \) and \( V(G_2) \) are homeomorphic. It follows that the problem of homeomorphy of manifolds of dimension 4 is not decidable: here is a serve from algebra to topology.

It is via algebra that mathematicians look for substantial aid from computer science, e.g., in the use of computer algebra for algebraic geometry or algebraic topology. Yet, it is also via algebra that many undecidable problems branched out into other domains of mathematics. The use of algebra allows a transfer of problems from topology towards algebra, but also a transfer, in the reverse direction, of undecidability results.

8.4.5 Creative problems

It has been seen that the complement \( \overline{A} \) of a non-recursive r.e. set \( A \) cannot be r.e. Yet \( \overline{A} \) contains many r.e. sets, for example all of its finite subsets. But every r.e. \( W_n \) is distinct from \( \overline{A} \): there exists an element \( a_n \) in \( \overline{A} \setminus W_n \) or in \( W_n \setminus \overline{A} \). In particular, there does not exist an enumeration procedure which is both ‘sound’ and ‘complete’ for \( \overline{A} \): if a procedure enumerates only elements of \( \overline{A} \) (it is sound), it will necessarily forget some, and if it does not forget any (it is complete), it will also produce elements of \( A \). If the counterexample \( a_n \) can be computed effectively from \( n \), the following definition is reached:

**Definition 8.14** A recursive function \( f \) is a **productive** function for a set \( P \) if for every \( n \), \( f(n) \in P \setminus W_n \) or \( f(n) \in W_n \setminus P \). A set \( P \) is **productive** if there exists a productive function for \( P \). A set is **creative** if it is r.e. and if its complement is productive.

The productive problems are not semi-decidable, and **unfoolable**. Each time that someone pretends to have a decision procedure for \( P \), \( P \) productive, by supplying
an algorithm of index \( n \) such that \( P = W_n \), only \( f(n) \) need be computed to put it to rest.

**Example**  The set \( K \) of the halting problem is creative, since it is r.e. and its complement \( \bar{K} \) is productive. In fact, \( \bar{K} \) has the identity as production function. For every \( n, \ell \in \bar{K} \setminus W_n \), by the definition of \( K \).

It turns out that a large number of known r.e. are creative, for example the set \( K \), and the set of formulas provable in arithmetic: it is this last example which gives these sets the qualifier of ‘creative’, due to Post, since they illustrate the creative character of mathematical activity. The incompleteness of arithmetic is explained in this framework. The set of theorems is creative, hence the set \( P \) of non-provable formulas is productive. The set of negations of theorems is r.e., like the set of theorems, hence it is a certain \( W_n \), and a subset of \( P \). If \( f \) is a production function for \( P \), then \( f(n) \in P \setminus W_n \), i.e., \( f(n) \) is the encoding of a non-provable formula \( \varphi \), such that \( \neg \varphi \) is equally non-provable.

The creative problems are exactly the \( \Sigma_1 \)-complete problems for the reduction \( \leq_m \) (Myhill theorem).

### 8.5 Complexity

Even though computability was developed well before computer science, it is the former which largely motivated the study of the complexity of algorithms.

There is an ‘abstract theory’ of complexity, due to M. Blum (1967 [10]) who, starting from minimal assumptions about what a complexity measure should be, proves remarkable general phenomena (gap and speedup properties). It is in the same tradition of the fundamental results of Kleene, such as universality and recursivity. Complexity will not be studied here in this manner, but, rather, algorithmically.

Computability does not deal with the resources needed for a computation. It just supposes a Turing machine with an infinite tape. Algorithm analysis studies algorithms to determine the resources needed for them to run. There are several kinds of resources:

- Two are theoretically unlimited, though in practice finite and expensive:
  - time and
  - space (main memory).
- Others are available in a fixed, limited number, and the optimal allocation problem can be very difficult:
  - registers, stacks, \( \ldots \),
  - processors, and
— peripherals (secondary memory, random or sequential, input-output, etc.).

Only the space and time problems will be studied here. Their study is necessary to produce efficient algorithms.

### 8.5.1 Complexity measures

Before looking at the general context of partial recursive functions and Turing machines, here is a simple example of a class of functions and the programs computing them.

**Example** It is easy to see that the primitive recursive functions are the functions computed by a restricted class of iterative programs, the finite loop programs defined in §8.1.2.

There are two ways to measure the complexity of such a program. First consider the depth \(d(\pi)\) of a program \(\pi\) defined by the nesting depth of \(\text{do} \ldots \text{end}\) loops. For example, the parenthesization \((())()\) has depth 3. Second, let \(x_1, \ldots, x_q\) be variables of \(\pi\). Introduce the function \(T_\pi : \mathbb{N}^q \to \mathbb{N}\) measuring for \(a_1, \ldots, a_q \in \mathbb{N}\) the number of assignments executed by \(\pi\) with \(a_1, \ldots, a_q\) as initial values for variables \(x_1, \ldots, x_q\). The function \(T_\pi\) is a temporal complexity of dynamic character since it is computable by simulating the execution of the program, while \(d(\pi)\) is a static complexity, i.e., computable upon reading the program.

What is interesting about this class of programs is that \(T_\pi\) and \(d(\pi)\) are related. Introduce the functions \(f_n : \mathbb{N} \to \mathbb{N}\) defined by:

\[
\begin{align*}
f_0(m) &= \begin{cases} m + 1 & \text{if } m = 0 \text{ or } 1 \\ m + 2 & \text{otherwise,} \end{cases} \\
f_{n+1}(m) &= (f_n^m(1)).
\end{align*}
\]

These functions grow very quickly and can set bounds on \(T_\pi\). It can be proved that \(d(\pi) = p\) if and only if there exists \(k\) such that

\[
T_\pi(a_1, \ldots, a_k)^k \leq f_p(\max a_i).
\]

Hence the complexity analysis of finite loop programs has a satisfactory solution. But this language is not powerful enough, since it only computes the primitive recursive functions, and particularly since its programs halt.

In the case of a programming language computing all of the partial recursive function, there is no analogous result. There are programs whose execution time, compared to their textual length, are as long as one wishes, just as in logic, there are trivial theorems with gigantic derivations. These phenomena are due to the undecidability of the corresponding problems.
There are, however, static measures of complexity, i.e., depending only on the text of the program, much used in software engineering: number of lines or identifiers, or diverse characteristics of the calling graph for procedures. In a given class of applications for which standard values are known, these measures enable a superficial understanding of the readability of programs, but not of their complexity, be it of the logic of the design or of the efficiency of execution. These typical values are mainly a good source of information to determine the most useful compiler optimizations.

In the absence of a static measure of complexity, a dynamic measure must be given, which will take into account all the possible executions over all the possible data.

Among the models of computation described in this chapter, those of deterministic Turing machines (DTM) are the best adapted. The length of a computation \( c_0 \rightarrow c_1 \rightarrow \ldots \rightarrow c_n \) is \( n \).

**Definition 8.15**

1. Let \( (\Gamma, Q, R) \) be a DTM and \( S \) and \( T \) be functions from \( N \) to \( N \). The DTM \( (\Gamma, Q, R) \) is of **temporal complexity** (less than) \( T \) if the length of the computation of every word of length \( n \) is at most \( T(n) \). It is of **spatial complexity** (less than) \( S \) if, during the computation of every word of length \( n \), it traverses at most \( S(n) \) cells.

2. Let \( L \) be a subset of \( \Gamma^* \). \( L \), or the problem \( x \in L \) is of complexity \( T \) if there exists a DTM of complexity \( T \) accepting \( L \), i.e. such that every computation from a word of \( L \) halts.

3. Let \( f : \Gamma^* \rightarrow \Gamma^* \). The function \( f \) is of complexity \( T \) if there exists a DTM of complexity \( T \) computing \( f \).

In practice, the evaluation of the complexity of an algorithm solving a problem \( U \) is done as follows:

- The set \( I_U \) of of \( U \)'s instances is provided with a 'size' function \( | \cdot | : I_U \rightarrow N \), for example the length of a word or the dimension of a problem.

- A maximal temporal complexity \( T : N \rightarrow N \) is defined such that the execution of the algorithm with \( x \in I_U \) as data takes at most \( T(|x|) \) time units, this time being computed by associating with each instruction of the algorithm an execution time and by simulating its execution.

For example, computing the maximum of \( n \) integers requires \( (n-1) \) comparisons. It runs in \( O(n) \), \( n \) being the size of the problem. Recall the usual asymptotic notations: let \( f \) be a function from \( N \) to \( R^* \).

\[
O(f) = \{ g : N \rightarrow R^* ; \exists c \in R^+ \exists n_0 \in N \forall n \geq n_0 \ (g(n) \leq cf(n)) \}.
\]

Write \( g(n) = O(f(n)) \) instead of \( g \in O(f) \).
To justify such a practice in the framework of DTM, the instances of a problem must first be encoded in an alphabet. In the case of numeric data it will always be supposed that the encoding of integers is binary (or decimal, but never unary) and that the encoding of sets of integers can be done by listing their separate elements by a supplementary alphabet. This allows one to refer directly to the complexity of \( A \subseteq \mathbb{N}^k \) (or of the problem \( x \in A ? \)) and of a function \( f : \mathbb{N}^k \rightarrow \mathbb{N} \). The algorithm then has to be translated into a DTM, which is far from a normal translation.

This is why algorithm analysis uses the random access machine (RAM) model. RAMs realistically simulate the behavior of a computer with an assembler-like language. The logarithmic cost assumption is used, i.e., the time to execute instruction ‘INST \( n \)’ is proportional to the logarithm of \( n \).

The complexity computations in different models, DTM or RAM, obviously give different results. The model and the assumptions about the cost of instructions must therefore be made precise. The DTM model, more primitive, is well adapted to the theoretical study of complexity. The RAM model, more realistic, applies to concrete algorithms of computer science. There are nevertheless simple relations between the computed complexities in each of the models.

### 8.5.2 Polynomial problems

In the second section of the chapter, problems were divided into two classes: decidable and undecidable. The decidable problems are not all equally tractable. Many standard algorithms, such as searches, sorts and polynomial evaluations have asymptotic complexities of \( O(\log n) \), \( O(n) \), \( O(n \log n) \), \( O(n^2) \), \( O(n^3) \), \ldots, hence are bounded by a polynomial. As they are standard, they are of course tractable and increasing the computation speed of machines significantly increases the size of allowable data: a machine 100 times faster than the old one allows one to handle data ten times larger for an algorithm in \( O(n^2) \). There are also algorithms which theoretically solve problems, by exhaustive search in a finite set which can be a powerset, for example, but which take lots of time \( (2^n, n^n, n!, \ldots) \). They are executable for only very small values of \( n \) (for example \( 50! \approx 3.10^{20} \) and increasing the power of a machine has little effect on the size of possible data.

**Definition 8.16** An algorithm is polynomial if there exists \( k \in \mathbb{N} \) such that its complexity is \( O(n^k) \). The problem \( x \in A ? \), or the set \( A \), is polynomial (or in class \( \text{P} \)) if there exists a polynomial algorithm solving it. A function \( f \) is a polynomial transformation (or is in class \( \text{P} \)) if there exists a polynomial algorithm computing it (\( A \) is in \( \text{P} \) if and only if \( 1_A \) is in \( \text{P} \)).

**Example** The unification of two terms is decidable. Although the usual algorithm is exponential, there are polynomial algorithms, and even linear ones. The unification problem is therefore in \( \text{P} \).
The definition of class \( \mathbf{P} \) is robust. For example, it is not affected if DTMs with several tapes are used. The same class is achieved with RAMs of logarithmic cost. A RAM program of complexity \( O(n) \) will have a complexity \( O(n^5) \) once simulated by a DTM but will remain polynomial. There are therefore good reasons to state Cook’s thesis:

Cook thesis: The tractable problems are those in class \( \mathbf{P} \).

Like Church’s thesis, Cook’s thesis is of experimental origin, sustained by convincing theoretical results, and provides a definition of the word tractable.

Remark: The tractable/non-tractable delimitation is less drastic than that of decidability. It is sometimes more convenient to use a non-polynomial algorithm for a problem in \( \mathbf{P} \), because it is easy to write and is of reasonable execution speed for the normal instances of the problem. This is the case for linear programming in which the non-polynomial simplex algorithm (Dantzig 1951 [32]) is used rather than the complicated and inefficient polynomial algorithm of Khachiyan (1979 [77]); there is a new efficient polynomial algorithm, proposed by Karmarkar in 1984 [75]. The role of constants is ignored when tractability is studied; it is possible to prefer an algorithm in \( 2^n \) rather than one in \( 10^{100}n \). So efficiency involves more criteria than just tractability.

There are also several important problems for which no polynomial algorithm is known. In order to solve them by preserving the polynomial computation time, the concept of algorithm will be extended by introducing faster machines than DTM. They cannot solve all problems in polynomial time but only those whose solution, once given, can be checked in polynomial time.

Example: The existence problem of a Hamiltonian cycle in a graph is of this kind: it is immediate—hence polynomial!—to check if a given cycle is Hamiltonian.

Definition 8.17 \( A \subseteq \Gamma^* \) is in class \( \mathbf{NP} \) (non-deterministic polynomial) if there exists \( B \subseteq \Gamma^* \times \Delta^* \) in \( \mathbf{P} \) and a polynomial \( p \) such that

\[ A = \{ x \in \Gamma^* ; \exists y \in \Delta^*, |y| \leq p(|x|) \text{ and } (x, y) \in B \}. \]

It is the existential quantifier \( \exists y \) which introduced non-determinism. To solve \( x \in A \), one must

- generate \( y \) such that \( |y| \leq p(|x|) \);
- compute \( 1_B(x, y) \).

The complexity of the first operation is at most \( p(|x|) \). The second is polynomial in \( |x| + |y| \) by assumption on \( B \), hence from the condition on \( y \) it is polynomial in \( x \). This procedure can be executed by a (non-deterministic) Turing machine:
Definition 8.18 Let \( L \subseteq \Gamma^* \). A Turing machine \((\Gamma, Q, R)\) accepts \( L \) in time \( T \) over a subset \( Q_0 \) of \( Q \), if for every \( x \in L \) there exists a computation from \( x \), ending in a state in \( Q_0 \) and of length less than \( T(|x|) \).

Proposition 8.23

(1) \( L \) is in \( \text{NP} \) if and only if there exists a Turing machine accepting \( L \) in polynomial time.

(2) If \( L \) is accepted by a Turing machine in time \( T(n) = O(n^k) \), then there exists a constant \( c \) and a DTM of complexity \( O(c^n) \) solving the problem \( x \in L \).

Part (1) is often taken as the definition of \( \text{NP} \). Part (2) states that the problems in \( \text{NP} \) have at most exponential complexity. The essential idea of the proof (of the converse of (1) and (2)) is as follows. Let \( d \) be the degree of non-determinism of a Turing machine, i.e., the maximum number of rules \((q, a, \ldots)\) starting with the same pair \((q, a)\); it is the choice between these rules which creates non-determinism. For a DTM, \( d = 1 \). Then for \( x \in \Gamma^* \), there are at most \( d^{T(|x|)} \) different computations of length \( \leq T|x| \) from \( x \). By exploring the computations, a DTM accepting \( L \) can be constructed and one proves that \( L \) is in \( \text{NP} \).

The class \( \text{NP} \) does not abbreviate Not Polynomial. For example, the decidability problem in Presburger arithmetic is decidable in a non-deterministic time of at least \( 2^{k^n} \), where \( k \) is constant: this problem is not even in \( P \).

It is clear that \( P \subseteq \text{NP} \). But is it clear that the power of computation has been increased by allowing non-determinism? No, since, for the moment, no example of a problem in \( \text{NP} \) has been shown to not be in \( P \). However, there are good reasons to believe that \( P \neq \text{NP} \).

8.5.3 Strong polynomial reduction

The definition of this reduction is directly inspired from the strong reduction under a recursive transformation: \( A \preceq_P B \) if there exists a polynomial transformation \( f \) such that \( x \in A \) if and only if \( f(x) \in B \). If one can solve \( x \in B \), then, given extra polynomial time, \( x \in A \) can be solved. This reduction is particularly interesting outside of \( P \). Here is a first example of polynomial reduction.

**Hamiltonian cycle (HC)**

Given: an undirected graph \((S, A)\).

Problem: does \((S, A)\) have a Hamiltonian cycle?

**Traveling Salesperson (TS)**

Given: a distance \( d : C \times C \rightarrow \mathbb{N} \) and an integer \( M \).

Problem: is there a tour of \( C \) of length at most \( M \) ?
(HC) \( \leq_P \) (TS) is proven by constructing an instance \((C, d, M)\) of (TS) from an instance \((S, A)\) of (HC). Let:

\[
C = S, \\
M = |S|, \\
\hat{d}(x, y) = \begin{cases} 
1 & \text{if } (x, y) \in A \\
2 & \text{otherwise.}
\end{cases}
\]

This transformation is polynomial, after a suitable encoding in an alphabet describing the problems and \((S, A)\) has a Hamiltonian cycle if and only if there is a tour of \(C\) of length at most \(M\).

**Definition 8.19** An NP-complete problem is a problem in NP, complete under polynomial reduction \(\leq_P\).

The following properties are immediate (\(\sim_P\) is the equivalence generated by \(\leq_P\)):

**Proposition 8.29**

1. If \(A\) and \(B\) are NP-complete, then \(A \sim_P B\).
2. If \(A\) is NP-complete and in P, then \(P = NP\).
3. If \(A\) is NP-complete and if \(B\) is in NP, then \(B\) is NP-complete if \(A \leq_P B\).

**Remarks**

1. states that all NP-complete problems are as difficult as each other;
2. explains why the study of these problems is crucial if one wishes to show that \(P = NP\) or \(P \neq NP\);
3. gives the standard method for proving that a problem \((B)\) is NP-complete if this is already known for \((A)\).

\(\square\)

Several hundred NP-complete problems have been catalogued in many domains: operations research, graph theory, robotics, number theory, etc. For each of these problems, no polynomial algorithm has been found, despite intensive effort. This is why, because of property (2), \(P \neq NP\) is likely, even though it has never been proven that it is impossible to obtain polynomial solutions to these problems.

**Conjecture** \(P \neq NP\)
**SAT (satisfiability in propositional logic)** Let $X$ be a finite countable set. Given: a conjunctive formula $\varphi \in \text{Prop}[X]$. Problem: is $\varphi$ satisfiable?

**Theorem 8.30 (Cook) (SAT) is NP-complete.**

This was the first problem shown to be NP-complete, in Cook’s initial article defining polynomial reduction, NP and NP-complete problems, in 1971 [24]. Let $A \subseteq \Gamma^*$ be accepted in polynomial time by a Turing machine $(\Gamma, Q, R)$. The proof, long and tedious, explicitly constructs for each $x \in \Gamma^*$ a propositional formula $\varphi_x$ which is satisfiable if and only if $x \in A$. From (SAT), one can prove the NP-completeness of other problems (proposition (3)). Karp gave twenty or so in 1972 [76], and Garey and Johnson more than 300 in 1979 [49].

Before presenting a few examples, note the following parallel:

<table>
<thead>
<tr>
<th>Recursive</th>
<th>Class $\mathbb{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r.e.</td>
<td>Class NP</td>
</tr>
<tr>
<td>$\leq_m$</td>
<td>$\leq_P$</td>
</tr>
<tr>
<td>Church thesis</td>
<td>Cook thesis</td>
</tr>
</tbody>
</table>

In particular, the definition of NP by existential quantification from $\mathbb{P}$ is similar to the definition of the r.e. from the recursive, with the added restriction that there be a polynomial bound on the quantifier. However, many results in recursive function theory do not have equivalents in complexity theory.

### 8.5.4 Examples of NP-complete problems

The first problem (3-SAT) essentially plays a technical role, as it is easier to use than (SAT) to initiate the list of NP-complete problems. Recall that if $(S, A)$ is a unoriented graph, a subset $R$ of $S$ is a cover of $S$ if every arc has at least one end in $R$.

**3-SAT**

Given: a conjunctive formula $\varphi$ of $\text{Prop}[X]$, in which each clause contains exactly three literals.

Problem: is $\varphi$ satisfiable?

Here is the transformation allowing the reduction of (SAT) to (3-SAT). Let $C = \{l_1, \ldots, l_k\}$ be a clause containing $k \geq 4$ literals. Introduce $k - 3$ new propositional variables $y_1, \ldots, y_{k-3}$ and replace clause $C$ by the $k - 2$ clauses:
\begin{align*}
\{l_1, l_2, y_1\}, \\
\{l_3, \neg y_1, y_2\}, \\
\{l_4, \neg y_2, y_3\}, \\
\ldots \ldots \\
\{l_{k-2}, \neg y_{k-4}, y_{k-3}\}, \\
\{l_{k-1}, l_k, \neg y_{k-3}\}.
\end{align*}

A two-literal clause \(\{l_1, l_2\}\) is replaced by the two clauses \(\{l_1, l_2, y\}\), \(\{l_1, l_2, \neg y\}\), and a one-literal clause \(\{l_i\}\) by the four clauses \(\{l_i, y_1, y_2\}\), \(\{l_i, \neg y_1, y_2\}\), \(\{l_i, y_1, \neg y_2\}\), and \(\{l_i, \neg y_1, \neg y_2\}\).

**Vertex cover (VC)**

Given: a graph \((S, A)\) and an integer \(k\).

**Problem**: does \((S, A)\) allow a cover \(R\) of \(S\) with \(|R| \leq k\)?

\((3\text{-SAT}) \leq_P (VC)\) is proven by associating with each instance \(\varphi = C_1 \land \ldots \land C_n\) of \(3\text{-SAT}\) a graph having \(3n\) vertices, one for each literal of each clause, by linking the three vertices associated with a clause, and between clauses, the vertices corresponding to opposite literals \((l, \neg l)\). Let \(k = 2n\). For example, for

\[\varphi = (x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor x_2 \lor x_4),\]

the graph obtained is:

![Graph](image)

Some other \(\text{NP}\)-complete problems of graph theory: \(k\)-coloring, clique and Hamiltonian cycle—but the edge cover and Euler cycle problems are in \(\text{P}\). For network problems: traveling salesperson, maximal path between two vertices and various flow problems are \(\text{NP}\)-complete—but the minimal path problem is in \(\text{P}\). In computer science, many optimal code generation, data compression or dynamic memory allocation problems are \(\text{NP}\)-complete. Scheduling problems, for one or more processors, with priority relations between tasks, with time or resource constraints, etc., are easily \(\text{NP}\)-complete. Here are some examples:

**Partition**

Given: a finite set \(A\) and a size function \(s : A \rightarrow \mathbb{N}^*\).

**Problem**: does there exist a subset \(A' \subseteq A\) such that \(\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)\)?
Conjunctive queries
Given: a finite set \( D \), a family \( R_1, \ldots, R_p \) of \( r_i \)-ary relations over \( D \), and a (closed formula) query \( \rho : \exists x_1 \ldots x_n (\alpha_1 \land \ldots \land \alpha_m) \), where each \( \alpha_i \) is an atom with parameters in \( D \).
Problem: \( D \models \rho \)?

Multiprocessor ordering (MO)
Given: a finite set \( T \) of tasks with a length function \( l : T \to \mathbb{N}^* \), a number \( m > 0 \) of processors and a deadline \( d > 0 \).
Problem: is there a scheduling compatible with the data?

\((\text{Partition}) \preceq_p \text{(MO)}\) easily, by restriction, since \((\text{Partition})\) is the particular case of \((\text{MO})\) when \( m = 2 \) and \( d = 1/2 \sum_{a \in T} l(a) \).

8.5.5 Solving NP-complete problems

For a given NP-complete problem, it is possible that only a restricted class of problems is presented and that this class is in P. For example, (2-SAT), analogous to 3-SAT, and the 2-coloring of graphs are in P. The priority scheduling problem is in P if the priority graph is a tree or if there are only two processors. (SAT) restricted to Horn clauses is polynomial.

A non-polynomial algorithm can have an average polynomial temporal complexity, and quite satisfactory behavior in practice, if the worst cases are 'rare'. The probabilistic study of algorithms is not far advanced.

It is also possible that an algorithm may not be polynomial in the size of its data but polynomial, say, in the largest integer appearing in the data; obviously, \( n \) is linear in \( n \), but not in \( \log n \). This is the case when an exhaustive search is being simplified, for example by dynamic programming. For the \((\text{Partition})\) problem, this method leads to an algorithm in \( O(nk) \) where \( n = |A|, k = \sum_{a \in A} s(a) \). This algorithm is not linear in the size of its data, which is \( O(n \log k) \). However, in typical cases, \( k \) is in \( O(n) \), so a polynomial algorithm is obtained. Problems of this kind are called pseudo-polynomial and are typically solved by dynamic programming or 'branch and bound' methods. But not all NP-complete problems are pseudo-polynomial. This is the case for the (TS), for which the complexity does not come from possibly large distances, since the reduction of (HC) to (TS) only uses distances of 1 or 2, and (HC) itself is NP-complete.

If the known algorithms are not really polynomial, even with the probabilistic or pseudo variants, a polynomial algorithm can be sought to approximate a solution to an optimization problem. There are many good approximating algorithms. For (TS), one can find a tour of at most twice the optimal length in polynomial time. For some problems, it is possible to obtain a solution \( \epsilon \) away from optimal for an arbitrary positive \( \epsilon \). For others, such as (TS), this is impossible unless \( P = NP \).
Exercises

1. Show that \( (\cdot)_k \) is a bijection from \( \mathbb{N}^k \) to \( \mathbb{N} \), and find its inverse \( (\pi_1^k, \ldots, \pi_k^k) : \mathbb{N} \to \mathbb{N}^k \).

2. Show that \( \mathbb{N}^2 \) is enumerated by the formula:
   \[
   (x, y) = 1/2((x + y)^2 + 3x + y).
   \]

3. Define \( \|x\| \), \( \text{val}(x) \), \( \text{pre}(i, x) \), \( \text{last}(i, x) \) and \( \text{cons} \) using \( (\cdot) \) and the projections \( \pi^1 \) and \( \pi^2 \).

4. Show that the following functions are primitive recursive:
   1. the predecessor \( p \) \( (p(0) = 0, p(x + 1) = x) \);
   2. (total) subtractions \( \cdot \) (defined by \( x \cdot y = x - y \) if \( x \geq y \), 0 otherwise);
   3. the relations \( x \leq y, x = y \);
   4. the distance \( |x - y| \).

5.* Show that if \( g \), \( h \) and \( P \) are primitive recursive, then \( f \) defined by
   \[
   f(n) = \begin{cases} 
   g(n) & \text{if } n \in P \\
   h(n) & \text{otherwise}
   \end{cases}
   \]
   is primitive recursive.

6.* Show that if \( g \) is primitive recursive, then \( f \) defined by
   \[
   f(m, \bar{n}) = \sum_{p \leq m} g(p, \bar{n})
   \]
   is also primitive recursive. Similar result for \( \Pi \), max and min instead of \( \Sigma \).

7. Show that the class of primitive recursive functions is stable under the bounded maximization operator \( \mu' \) (to be defined).

8. Show that addition is Turing-computable.

9. Show that the successor and addition are respectively defined in Peano arithmetic by the formulas \( y = sx \) and \( z = x + y \).

10. Show that if \( f(n) = h(g_1(n), g_2(n)) \) and if \( h \), \( g_1 \) and \( g_2 \) are definable in arithmetic, then \( f \) is as well.

11.* Show that finite sets are recursive.

---

*Hints to the exercises labeled with a * can be found at the end of the book.*
12. Show that if $A$ is the domain of a computable partial function and is infinite, then it is the image of an injective recursive function. Show that if $f : \mathbb{N} \to \mathbb{N}$ is recursive increasing, then $f(\mathbb{N})$ is recursive.

13. Let $A \subseteq \mathbb{N}^k$. Show that if the function $1_A$ is $T$-definable, then the set $A$ is also $T$-definable.

14. Deduce from the s–m–n theorem that for every r.e. set $A \subseteq \mathbb{N}^2$, there exists a recursive function $f$ such that $A = \{(x, y) : x \in W_{f(y)}\}$.

15.* Solve for $X$ in the equations: (1) $X = AX^\perp$; (2) $X = XA^\perp$.

16. Show that each of the problems on page 340 is undecidable.

17. Show that the mapping associating the integer $2^{a_1} + \ldots + 2^{a_n}$ with the finite set $\{a_1, \ldots, a_n\}$ defines a bijection from the set of finite subsets of $\mathbb{N}$ to $\mathbb{N}$.

18.* Apply the Rice–Shapiro theorem to show that the following properties of r.e. sets are not semi-decidable:
   1. $A$ is recursive;
   2. $A$ is not recursive;
   3. $A = \mathbb{N}$.

19.* Apply the Rice–Shapiro theorem to show that the following properties are semi-decidable:
   1. $33 \in A$;
   2. $A \neq \emptyset$.

20.* Deduce from Matijasevič’s theorem and from the universality theorem the existence of a polynomial $P$ such that for every r.e. subset $A \subseteq \mathbb{N}$, there exists an integer $i$ such that

$$A = \{x \in \mathbb{N} : \exists (y_1, \ldots, y_k) \in \mathbb{N}^k P(i, x, y_1, \ldots, y_k) = 0\}.$$ 

21.* Let $f$ be an injective recursive function from $\mathbb{N}$ to $\mathbb{N}$, and $s$ be the limit of the sequence $s_n = \sum_{k=0}^{n} 1/2^f(k)$. Show that $s$ is a recursive real if and only if $f(\mathbb{N})$ is a recursive set.

22. Show the following inclusions: $\Sigma_n \cup \Pi_n \subseteq \Delta_{n+1}$ for every $n \geq 0$.

23. Prove that the interpretation in $\mathbb{N}$ of a formula of arithmetic is an arithmetic set (start with the interpretation of terms, then reason by induction over formulas).
24.* By using the universality theorem for r.e. sets, show that for every $n \geq 1$, there exists a subset $U_n \subseteq \mathbb{N}^2$, which is $\Sigma_n$ and universal for the subsets $\Sigma_n$ of $\mathbb{N}$.

25.* Show, using contradiction and lemma 8.11 that the set of true formulas in $\mathbb{N}$ is not definable in $\mathbb{N}$ by a formula of arithmetic (it is a theorem of Tarski's).

26.* Show that $A = \{p; W_p \text{ is recursive}\}$ is $\Sigma_3$.


28. Show that $K$ and its complement $\overline{K}$ are incomparable for $\leq_m$.

29. Write a program with finite loops $\pi$ to compute the product of two integers, and compute $T_\pi(a, b)$ and $d(\pi)$.

30. Using the transformations given on page 358, prove that $(\text{SAT}) \leq_f (3\text{-SAT})$.

**Bibliographic notes**

Rogers' book [129] is a standard reference for recursive function theory. [139] is a recent book treating these questions from a programming point of view. Programs with finite loops are introduced by Meyer and Ritchie [106]. See also [34, 139]. The encoding of logic is made explicit in several books on logic, including [135]. On proofs of soundness and the strength of theories, see [53]. The notion of reflexivity in programming is analyzed in, among others, [45]. Shoenfield's little book [136] is an introduction to the study of degrees of unsolvability. Garey and Johnson's book [49] is the reference for NP-complete problems.
Hints to selected exercises

Chapter 2

2. For the lexical order over $A^*$, if $i < j$ in $A$, every word $u_jv$ has as lower
bounds the $u_i^n$, for $n \geq 0$.

6. These properties are proved by induction. Consider (3) for a term $M$ by
supposing it is true for its immediate subterms. If $M = c$, ar$(c) = 0$ and $u \in
\mathcal{O}(M)$, then $u = e$, then $u \not\in \mathcal{O}(M)$. If $M = fM_1 \ldots M_p$ and $u \in \mathcal{O}(M)$,
then $u = i \cdot v$ with $1 \leq i \leq p$ and $v \in \mathcal{O}(M_i)$, hence $M(u) = M_i(v)$; by
applying the inductive hypothesis to $M_i$, if ar$(M_i(v)) = n$, then $v \cdot 1, \ldots, v \cdot n \in \mathcal{O}(M_i)$ and $u \cdot (n + 1) \not\in \mathcal{O}(M_i)$; hence $i \cdot v \cdot 1, \ldots, i \cdot v \cdot n \in \mathcal{O}(M)$
and $i \cdot u \cdot (n + 1) \not\in \mathcal{O}(M)$.

11. Proof by induction over $\lambda$-terms. For $M \equiv PQ$, distinguish depending on
$P$'s form.

12. The set of free variables of a de Bruijn term is given by

\[
\begin{align*}
\text{var}(i) &= \{v_i\}, \\
\text{var}(MN) &= \text{var}(M) \cup \text{var}(N), \\
\text{var}(\lambda(M)) &= \{v_{i-1}; i > 0 \text{ and } v_i \in \text{var}(M)\}.
\end{align*}
\]

Chapter 3

2. The graph is a parallelepiped of $2 \times 3 \times 3$ vertices.

3. If $p = m + 1$, then $Mpq = M'qm$, where $M' \equiv \lambda pq . pF(Kb)(qF(Ka))$, and
$M0q = a$. Hence $Mpq = a$ if $p \leq q$, and $= b$ otherwise. The length of the
reduction is proportional to $\min(p, q)$.
6. The types of the combinators S, B and C are

\[
\sim \quad \text{val } S = \text{fn} : (\text{'}a \rightarrow \text{'}b \rightarrow \text{'}c) \rightarrow (\text{'}a \rightarrow \text{'}b) \rightarrow \text{'}a \rightarrow \text{'}c
\]

\[
\text{val } B = \text{fn} : (\text{'}a \rightarrow \text{'}b) \rightarrow (\text{'}c \rightarrow \text{'}a) \rightarrow \text{'}c \rightarrow \text{'}b
\]

\[
\text{val } C = \text{fn} : (\text{'}a \rightarrow \text{'}b \rightarrow \text{'}c) \rightarrow \text{'}b \rightarrow \text{'}a \rightarrow \text{'}c
\]

11. Show that \( M = \pi^1(M \land N) \) and \( N = \pi^2(M \land N) \) by induction over \( M \land N \).

14. Iterators over trees and forests:

\[
\text{fun } Tree\_it\ u\ v\ c\ (\text{Union}(x,f)) = u(x, \text{Forest\_it } u\ v\ c\ f)
\]
and

\[
\text{Forest\_it } u\ v\ c =
\]

\[
\begin{aligned}
\text{let fun } F \text{ Empty } = v \\
F \text{ (Cons}(a,f)) &= c(\text{Tree\_it } u\ v\ c\ a, F f)
\end{aligned}
\]

\[
\text{in } F \text{ end;}
\]

\[
\sim \quad \text{val } Tree\_it = \text{fn} : (\text{'}a \rightarrow \text{'}b \rightarrow \text{'}c) \rightarrow
\]

\[
\begin{aligned}
\text{'}b \rightarrow \\
(\text{'}c \rightarrow \text{'}b \rightarrow \text{'}b) \rightarrow \\
\text{'}a \text{ Tree} \rightarrow \text{'}c
\end{aligned}
\]

\[
\text{val } Forest\_it = \text{fn} : (\text{'}a \rightarrow \text{'}b \rightarrow \text{'}c) \rightarrow
\]

\[
\begin{aligned}
\text{'}b \rightarrow \\
(\text{'}c \rightarrow \text{'}b \rightarrow \text{'}b) \rightarrow \\
\text{'}a \text{ Forest} \rightarrow \text{'}b
\end{aligned}
\]

Chapter 4

1. \( \Sigma_0 = \{ \bot \} \cup \text{At}_B, \Sigma_1 = \{ \forall x; x \in X \} \cup \{ \exists x; x \in X \}, \Sigma_2 = \{ \land, \lor, \Rightarrow \} \).

3. \( \lambda\varphi\psi.\lambda\varphi\Rightarrow\psi.\text{case}(z,[x\varphi].f\varphi,[y\psi].y) : \\
(\varphi \lor \psi) \Rightarrow ((\varphi \Rightarrow \psi) \Rightarrow \psi)
\]

\( \lambda\varphi\Rightarrow\psi(\psi\Rightarrow\chi)\lambda\varphi\land\psi.\varphi(\pi^1\varphi)(\pi^2\varphi) : \\
(\varphi \Rightarrow (\psi \Rightarrow \chi)) \Rightarrow (\varphi \land \psi) \Rightarrow \chi
\]

\( \lambda\varphi(\varphi\Rightarrow(\psi\Rightarrow\chi))\lambda\varphi\Rightarrow\chi.\text{case}(z,[f\varphi\Rightarrow\psi].\varphi(f\varphi),[g\varphi\Rightarrow\chi].\varphi(g\varphi)) : \\
((\varphi \Rightarrow \psi) \lor (\varphi \Rightarrow \chi)) \Rightarrow \varphi \Rightarrow (\psi \lor \chi)
\]

In Standard ML, the types and the terms would be:

\[
\begin{aligned}
\text{fun } A (\text{one } x) f &= f x \\
\text{and uncurry } f (y,z) &= f y z \\
\text{and disjunct (one } f) x &= \text{one } (f x) \\
\text{or disjunct (two } g) x &= \text{two } (g x)
\end{aligned}
\]

\[
\sim \quad \text{val } A = \text{fn} : (\text{'}a,\text{'}b) \rightarrow (\text{'}a \rightarrow \text{'}b) \rightarrow \text{'}b
\]
\[
\text{val uncurry} = \text{fn} : (\lambda a \to \lambda b \to \lambda c \to \lambda a * \lambda b \to \lambda c)
\]
\[
\text{val disjunct} = \text{fn} : (((\lambda a \to \lambda b),(\lambda a \to \lambda c)) \cup \lambda a \to (\lambda b, \lambda c)) \cup \lambda a
\]

4. Rules of introduction and elimination for equivalence:

\[
\begin{align*}
(I_{\equiv}) : & \quad \frac{\psi}{\varphi \equiv \psi} \\
(E_{\equiv 1}) : & \quad \frac{\varphi \equiv \psi}{\varphi} \\
(E_{\equiv 2}) : & \quad \frac{\varphi \equiv \psi}{\psi}
\end{align*}
\]

11. The \(\eta\) rule for conjunction is obtained by combining an elimination, then an introduction of \(\land\),

\[
\begin{array}{c}
\frac{\varphi \land \psi}{\varphi}, \\
\frac{\varphi \land \psi}{\psi}
\end{array}
\]

and by considering that if \(x\) is a derivation of \(\varphi \land \psi\), the previous combination, with \(x\) above each assumption, can be simplified to \(x\):

\[
(\pi^1(x), \pi^2(x)) \rightarrow_\eta x.
\]

For implication, the rule \(\lambda x . M x \rightarrow_\eta M\) is obtained, if \(x \notin \text{var}(M)\).

14. For contradiction, suppose \(\neg \exists x (\neg \varphi)\). Then, (by \(\vdash_M \forall x (\neg \varphi)\)), then (\(\vdash_C \forall x \varphi\)) \(\bot\) is obtained by supposing \(\neg \forall x \varphi\). By (\(C\)), \(\exists x (\neg \varphi)\) follows.

17. Proof in LK of Łukasiewicz’s axiom

\[
\frac{\varphi : \varphi, \chi}{\varphi, \varphi \Rightarrow \chi} : \psi : \psi
\]

\[
\frac{(\varphi \Rightarrow \chi) \Rightarrow \psi : \varphi, \psi}{\varphi, \omega : \varphi}
\]

\[
\frac{(\varphi \Rightarrow \chi) \Rightarrow \psi, \psi \Rightarrow \varphi, \omega : \varphi, \varphi}{\varphi \Rightarrow \chi} \Rightarrow \psi : (\psi \Rightarrow \varphi) \Rightarrow (\omega \Rightarrow \varphi)
\]

\[
((\varphi \Rightarrow \chi) \Rightarrow \psi) \Rightarrow ((\psi \Rightarrow \varphi) \Rightarrow (\omega \Rightarrow \varphi)).
\]

21. At least three elements: \(\exists x_1 x_2 x_3 (x_1 \neq x_2 \land x_2 \neq x_3 \land x_3 \neq x_1)\). At most three elements: \(\exists x_1 x_2 x_3 \forall x (x = x_1 \lor x = x_2 \lor x = x_3)\). Exactly three: the conjunction of the two formulas.

23. Associativity is defined in \(\Sigma'' = \{e, m\}\) by

\[\forall x y z s t u (m(x, y, s) \land m(s, z, u) \land m(y, z, t) \Rightarrow m(x, t, u))\]

26. If \(m > n\), prove \(AE \vdash \neg (s^m(0) = s^n(0))\) by induction over \(n\). If \(n > m\), \(\neg (s^m(0) = s^n(0))\) follows from \(\neg (s^n(0) = s^m(0))\) by contraposition. \(AE \vdash s^m(0) + s^n(0) = s^p(0)\), if \(m + n = p\), is proven by induction over \(n\).
Chapter 5

2. Let \( f : 2^2 \rightarrow 2 \) define a binary connective \( c \) with which every connective can be defined. By considering negation, show first that \( f(0,0) = 1 \) and \( f(1,1) = 0 \). If \( f(x,y) \neq f(y,x) \), then \( f(x,y) = 1 - x \) or \( 1 - y \), and so \( f \) cannot be used to express disjunction. To be proven by induction over terms formed with \( c \).

3. Show that for every \( f : 2 \rightarrow 2 \) defined using these connectives, \( f(1) = 1 \).

8. For \( (p \Rightarrow q) \lor (q \Rightarrow p) \), use \( U = \{a, b, c\} \), with \( a < b, a < c \), and \( I(b) = \{p\} \), \( I(c) = \{q\} \). For \( (p \Rightarrow (q \lor r)) \Rightarrow ((p \Rightarrow q) \lor (p \Rightarrow r)) \), use the same \( U \) with \( I(b) = \{p, q\} \), \( I(c) = \{p, r\} \). For \( (p \Rightarrow q) \Rightarrow (\neg p \lor q) \), use \( U = \{a, b, c, d\} \), with \( a < b < d, a < c < d \) and \( I(b) = \{q\} \), \( I(c) = \{p, q\} \).

10. The theory of fields of characteristic zero (\( TC_0 \)) is axiomatized by adding the axioms \( 1 + \ldots + 1 \) (\( p \) times) \( \neq 0 \), for all \( p \geq 2 \), to the theory of fields (\( TC \)). If \( TC_0 \vdash \varphi \), then \( TC_0 \cup \{\neg \varphi\} \) is inconsistent, so a finite subset of \( TC_0 \cup \{\neg \varphi\} \) is inconsistent, by compactness. Let \( p \) be the largest \( i \) such that the axiom \( i \cdot 1 \neq 0 \) appears in this finite subset. Then \( \varphi \) is a consequence of \( TC \) and of the axioms \( i \cdot 1 \neq 0 \) for \( i \leq p \).

14. If \( \mathcal{E} \) and \( \text{Alg}_2 \setminus \mathcal{E} \) are axiomatized by \( \Gamma \) and \( \Delta \), respectively, then \( \Gamma \cup \Delta \) is inconsistent, hence by compactness, a finite subset \( \Gamma' \cup \Delta' \) is inconsistent, with \( \Gamma' \subseteq \Gamma \) and \( \Delta' \subseteq \Delta \). Show that \( \mathcal{E} = \text{Mod}(\Gamma') \).

15. Let \( \mathcal{A} \) and \( \mathcal{B} \) be two models of \( \Gamma \). If \( \Gamma \) is complete, let \( \varphi \) be a proposition. Then, either \( \Gamma \vdash \varphi \), hence by the soundness of \( \vdash \), \( \mathcal{A} \models \varphi \) and \( \mathcal{B} \models \varphi \); or \( \Gamma \vdash \neg \varphi \), hence \( \mathcal{A} \models \neg \varphi \) and \( \mathcal{B} \models \varphi \). Therefore, \( \mathcal{A} \equiv \mathcal{B} \). Conversely, if all of the models of \( \Gamma \) are elementarily equivalent for every proposition \( \varphi \), \( \varphi \) or \( \neg \varphi \) is true in all the models of \( \Gamma \), i.e., \( \Gamma \vdash \varphi \) or \( \Gamma \vdash \neg \varphi \). Hence, by the completeness of \( \vdash \), \( \Gamma \vdash \varphi \) or \( \Gamma \vdash \neg \varphi \).

20. With a unary Skolem function \( D \), one obtains:

\[
S(\varphi) = \left\{ \forall \varepsilon (\varepsilon > 0) \lor D(\varepsilon) > 0, \right. \\
\left. \forall \varepsilon (\varepsilon > 0) \lor |x - a| < D(\varepsilon) \lor |f(x) - f(a)| < \varepsilon \right\}.
\]

22. A subset \( E \) of \( \mathcal{F} \) is consistent if for every \( x \), the set of \( f(x) \), for \( f \in E \) and \( x \in D(f) \), has at least one element.

25. Every function is a solution of \( f(x) = f(x) \). Every constant function over \( \mathbb{N} \) is a solution of \( f(x) = f(x + 1) \). In both cases the least solution is the constant \( 1 \).
Chapter 6

11. One can go from one presentation to the next by \( u = ab^2, v = b \) and \( a = (uv)^2, b = v \). The defined group is that of permutations of three elements, where \( u \) and \( v \) are the transpositions \((1 \ 2)\) and \((1 \ 3)\).

12. Modify the algebra of colored integers by taking three colors instead of two, addition conserving the 'best' color of its two arguments.

13. These properties of \( T_{\Sigma[\Xi]} \) follow only from the universal property. For composition, consider a \( \Sigma_0 \)-algebra \( \mathcal{X} \) and a \( \Sigma_0 \)-morphism \( \xi \) of \( \mathcal{X} \) in a \( \Sigma' \)-algebra. Extend \( \xi \) to a \( \Sigma \)-morphism defined over \( T_{\Sigma[\Xi]}[\mathcal{X}] \), then to a \( \Sigma' \)-morphism over \( T_{\Sigma[\Xi]}[T_{\Sigma[\Xi]}[\mathcal{X}]] \), . . . Then \( T_{\Sigma[\Xi]}[\mathcal{X}] = T_{\Sigma}[\mathcal{X}], T_{\Sigma[\Xi]}[\mathcal{X}] = \mathcal{X} \), and \( T_{\Sigma[\Xi]}[T_{\Sigma'}] = T_{\Sigma'} \).

15. The completion of \( \mathcal{E}_0 = \{ x^{-1}(xy) = y \} \) gives three rules \( x^{-1}(xy) \rightarrow y \), \((x^{-1})^{-1}y \rightarrow xy \) and \( x(x^{-1}y) \rightarrow y \).

16. The complete system is \( ee \rightarrow e, (xy)z \rightarrow x(yz) \) and \( e(ex) \rightarrow ex \).

17. If \( E \cup \{ M_i \neq N_i \} \) has a solution \( \sigma \), it is in particular a solution of \( E \), hence an instance \( \mu_\theta \) of a most general unifier \( \theta \) of \( E \). Therefore, \( \sigma(M_i) \neq \sigma(N_i) \) implies \( \theta(M) \neq \theta(N) \). Hence \( \theta \) is a solution of \( E \cup \{ M_1 \neq N_1, \ldots, M_p \neq N_p \} \).

18. Take \( M = \lambda n \lambda f \lambda x . f(nf x) \). Associate a type variable to each bound variable and each application node of the term's syntactic tree,

\[
M = (\lambda n : \alpha)(\lambda f : \beta)(\lambda x : \gamma) . \odot^\delta \odot^\epsilon n^\alpha f^\beta x^\gamma,
\]

by writing the constraints in these nodes,

\[
\beta = \epsilon \rightarrow \delta, \quad \zeta = \gamma \rightarrow \epsilon, \quad \alpha = \beta \rightarrow \zeta,
\]

and by synthesizing the types of the abstractions, at the root, obtain: \( \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \delta \). Solving the constraints gives

\[
((\epsilon \rightarrow \delta) \rightarrow (\gamma \rightarrow \epsilon)) \rightarrow (\epsilon \rightarrow \delta) \rightarrow \gamma \rightarrow \delta.
\]

Chapter 7

2. D-lists are useful to concatenate lists without using append. Define a predicate quicksort_dl whose first argument is a list and whose second is a D-list, to call quicksort_dl(\( X, Y-[] \)).
quicksort_dl([],X-X).
quicksort_dl([H|T],S-V) :-
  partition(T,H,Littles,Bigs),
quicksort_dl(Littles,S-[H|U]),
quicksort_dl(Bigs,U-V).

6. Show by induction over \( n \geq 1 \) that if \( u \in \Phi_{D}^{1n} \) and \( |u| < n \), then \( u \in \Phi_{D}^{1n} \).

7. Let \( A \in T_{F}^{1\omega} \). Since the set of clauses of \( P \) is finite, from ‘for every \( n \), there exists a clause such that \ldots ’, deduce that ‘there exists a clause such that for an infinity of \( n \ldots \)’. Use the assumption on the variables to work with a single substitution \( \sigma \) for every \( n \), and use the monotonicity of \( T_{F} \).

12. Of the seventeen cases to examine (four structural rules of \( LJ \), two times six logical rules, identity), certain rules are not applicable because of the structure of \( P \)-formulas or of the consequent. There remain only eight rules: \( L_{C}, L_{A}, L_{P}, L_{\land}, L_{\Rightarrow}, L_{\forall}, R_{v} \), and \( R_{\exists} \).

Chapter 8

5. \( f(n) = P(n)g(n) + (1 - P(n))h(n) \).

6. To prove that \( f \) is primitive recursive, it is defined in terms of \( g \):

\[
\begin{align*}
  f(0, \vec{n}) &= g(0, \vec{n}), \\
  f(m + 1, \vec{n}) &= f(m, \vec{n}) + g(m + 1, \vec{n}).
\end{align*}
\]

11. Prove by induction over \( n \) that the singleton sets \( \{n\} \) are recursive, and that a set with \( p \) elements is recursive by induction over \( p \).

15. Define \( \Gamma M N \) using \text{app} and apply \text{lemma 8.19}.

18. A set containing a recursive (respectively non-recursive) set is not necessarily recursive (respectively non-recursive). For example, \( \emptyset \subseteq K \subseteq \mathbb{N} \). The set \( \mathcal{U}(D_{m}) \) always being infinite, it is impossible that the indices of a finite class of sets be r.e.

19. \( 33 \in A \) if and only if \( A \in \mathcal{U}([33]) \). \( A \neq \emptyset \) if and only if \( A \in \bigcup_{n \geq 0} \mathcal{U}([n]) \), and \( \{n\} = D_{2n} \).

20. Since the r.e. set \( U = \{(e, x); \exists q T(e, x, q)\} \) is diophantine (Matjasevič’s theorem), there exists a polynomial \( P \) such that \( U = \{(e, x); \exists \vec{y} P(e, x, \vec{y}) = 0\} \). Hence \( W_{e} = \{x; \exists \vec{y} P(e, x, \vec{y}) = 0\} \).
21. If \( f(N) \) is recursive, the binary development of \( s \) is computable: the \( p \)-th numeral is a 1 if and only if \( p \in f(N) \), which is decidable. Conversely, if \( s \) is recursive, the binary development of \( s \) constitutes an increasing recursive enumeration of \( f(N) \), which must then be recursive.

24. Let, for example, \( n = 3 \):

\[
U_3 = \{(e, x) \in N^2; \exists x_1 \forall x_2 \exists x_3 T^3(e, x, x_1, x_2, x_3)\}
\]

is universal for the \( \Sigma_n \). In fact, if \( A \) is \( \Sigma_3 \), then

\[
A = \{x \in N; \exists x_1 \forall x_2 \exists x_3 R(x, x_1, x_2, x_3)\}
\]

for a recursive relation \( R \). Apply the universality theorem to the r.e. set \( \exists x_3 R \).

25. If \( \theta \) is a formula of arithmetic with one variable \( x \) such that \( N \models \varphi \) if and only if \( N \models \theta(\ulcorner \varphi \urcorner) \), apply lemma 8.11 to \( \neg \theta \).

26. An r.e. set is recursive if and only if its complement is r.e., and therefore has an index. On the other hand, \( x \in W_p \) if and only if \( \exists q T(p, x, q) \), from which \( x \notin W_p \) if and only if \( \forall q \neg T(p, x, q) \). Hence \( x \in A \) if and only if \( \exists e \forall x (\forall q \neg T(p, x, q) \iff \exists q T(e, x, q)) \). By putting this formula in prenex form, and by contracting the successive quantifiers of the same strength, a prefix \( \exists \forall \exists \) is obtained, which proves that \( A \) is \( \Sigma_3 \).
Bibliography


<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract interpretation</td>
<td>188</td>
</tr>
<tr>
<td>abstraction</td>
<td>32, 90, 136</td>
</tr>
<tr>
<td>abstraction depth</td>
<td>36</td>
</tr>
<tr>
<td>activation box</td>
<td>277</td>
</tr>
<tr>
<td>algebra</td>
<td>152, 154, 201</td>
</tr>
<tr>
<td>applicative</td>
<td>182</td>
</tr>
<tr>
<td>Boolean</td>
<td>154</td>
</tr>
<tr>
<td>canonical Herbrand</td>
<td>161</td>
</tr>
<tr>
<td>combinatory</td>
<td>183</td>
</tr>
<tr>
<td>complete</td>
<td>187</td>
</tr>
<tr>
<td>finitely generated</td>
<td>202</td>
</tr>
<tr>
<td>free</td>
<td>153, 203</td>
</tr>
<tr>
<td>generated</td>
<td>202</td>
</tr>
<tr>
<td>Herbrand</td>
<td>154, 169, 262</td>
</tr>
<tr>
<td>Heyting</td>
<td>166</td>
</tr>
<tr>
<td>initial</td>
<td>153, 187, 202</td>
</tr>
<tr>
<td>of finite type</td>
<td>202</td>
</tr>
<tr>
<td>product</td>
<td>201, 260</td>
</tr>
<tr>
<td>sub</td>
<td>191, 202</td>
</tr>
<tr>
<td>term</td>
<td>153, 201</td>
</tr>
<tr>
<td>terminal</td>
<td>202</td>
</tr>
<tr>
<td>trivial</td>
<td>152</td>
</tr>
<tr>
<td>with equality</td>
<td>159</td>
</tr>
<tr>
<td>algebraic</td>
<td>175</td>
</tr>
<tr>
<td>algorithm</td>
<td>44, 45, 315, 348</td>
</tr>
<tr>
<td>approximating</td>
<td>359</td>
</tr>
<tr>
<td>polynomial</td>
<td>353</td>
</tr>
<tr>
<td>alphabet</td>
<td>9</td>
</tr>
<tr>
<td>$\alpha$-conversion</td>
<td>36, 113</td>
</tr>
<tr>
<td>answer</td>
<td>269</td>
</tr>
<tr>
<td>qualified</td>
<td>270</td>
</tr>
<tr>
<td>application</td>
<td>32</td>
</tr>
<tr>
<td>applicative combination</td>
<td>93</td>
</tr>
<tr>
<td>arithmetic</td>
<td></td>
</tr>
<tr>
<td>elementary</td>
<td>142</td>
</tr>
<tr>
<td>Peano</td>
<td>141, 319</td>
</tr>
<tr>
<td>Presburger</td>
<td>320, 328, 355</td>
</tr>
<tr>
<td>arithmetic hierarchy</td>
<td>344</td>
</tr>
<tr>
<td>arity</td>
<td>11, 12, 93, 252</td>
</tr>
<tr>
<td>assumption</td>
<td>27</td>
</tr>
<tr>
<td>active, discharged</td>
<td>116</td>
</tr>
<tr>
<td>discharged</td>
<td>59, 115</td>
</tr>
<tr>
<td>atom</td>
<td>23, 111</td>
</tr>
<tr>
<td>closed</td>
<td>111, 293</td>
</tr>
<tr>
<td>Automath</td>
<td>36</td>
</tr>
<tr>
<td>automatic theorem-proving</td>
<td>172</td>
</tr>
<tr>
<td>axioms</td>
<td>12, 140, 159</td>
</tr>
<tr>
<td>Clark</td>
<td>291</td>
</tr>
<tr>
<td>equality</td>
<td>141, 212, 289, 291, 295</td>
</tr>
<tr>
<td>Morgan</td>
<td>127, 133, 149</td>
</tr>
<tr>
<td>Peano</td>
<td>141</td>
</tr>
<tr>
<td>Peirce</td>
<td>191</td>
</tr>
<tr>
<td>base</td>
<td></td>
</tr>
<tr>
<td>fact</td>
<td>262</td>
</tr>
<tr>
<td>Herbrand</td>
<td>154, 262</td>
</tr>
<tr>
<td>basis</td>
<td></td>
</tr>
<tr>
<td>combinatory</td>
<td>92, 135</td>
</tr>
<tr>
<td>connective</td>
<td>157</td>
</tr>
<tr>
<td>finite</td>
<td>223</td>
</tr>
</tbody>
</table>
\(\beta\)-contraction, 39  
\(\beta\)-conversion, 51  
\(\beta\)-reduction, 50  
  immediate, 49  
  relative, 101  
  weak, 67  
binder, 34  
binding, 69  
  dynamic, lexical, static, 69  
binding height, 35  
Boolean, 54  
border, 232  
broadcasting, 227  
C–Prolog, 251  
calculus  
  predicate, 140  
  propositional, 22, 111  
  sequent, 131, 258  
calculus of constructions, 123  
call  
  by name, 68, 178  
  by pattern, 72, 79  
  by value, 68, 77, 178  
CAML, 75  
category, 152  
Church, 48, 52, 54, 303, 314, 330, 338  
Church numeral, 54, 303  
Church–Rosser property, 52, 183, 240  
Clark, 291  
class of algebras  
  (finitely) axiomatizable, 164  
  stable under isomorphism, 210  
  stable under product, 210, 260  
  stable under quotient, 210  
  stable under subalgebra, 191, 210  
clause, 172  
definite, 257  
Horn, 257  
closed real field, 342  
closed world assumption, 287  
closure, 69, 77  
  existential, universal, 113  
  reflexive, transitive, 50  
transitive, 51  
combination  
  applicative, 93  
combinator, 37, 80, 89–93  
fixpoint, 55, 88  
S, K, I, 91, 135  
common part, 232  
compact, 175  
compiler, 336  
compiler generator, 336  
completeness, 2, 143, 161  
  strong, 275  
completion  
  Knuth–Bendix, 240–8, 298  
  program, 291  
complexity, 352  
  algorithmic, 86  
  logical, 86  
  spatial, temporal, 352  
composition, 303  
computation, 312  
  termination, 177  
concatenation, 9  
conclusion, 12, 13  
configuration, 312  
  halting, 312  
confluence, 2, 51, 122, 238, 268  
  local, 51, 102  
  strong, 51  
confrontation, 230  
congruence, 202  
  generated, 203  
  Thue, 218  
conjunction, 22  
connective, 22, 111, 154  
consequence  
  semantic, \(\vdash\), 160  
conservation, 100  
consistency, 52, 142, 161, 175, 206, 331  
constant  
  typed, 57  
context, 25, 50, 66, 255  
  typing, 59
continuation, 86
contraction, 39
contradiction, 126
contraposition, 123
coroutine, 285, 326
correspondence
  Curry–Howard, 5, 119, 121, 122, 135
  Galois, 207
cpo, 175
critical pair, 236, 244
  persistent, 246
Curry, 89, 121
curryification, 33
cut, 134, 258, 279, 286
  green, 280
  red, 280
cut-free system, 132
D-lists, 255
dag, see directed acyclic graph
de Bruijn, 36, 42
debug, 276
debugging, 270
decidability, 143
deduction, ⊢, 117
definite clause
  body, 257
  head, 257
definition
  completed, 290
  homogeneous, 289
  homogeneous converse, 290
  inductive, 11–9, 306
  recursive, 187, 318
degree, 106
  total, 106
degree of unsolvability, 347
denotation, 6
derivation, 13, 114, 116
  closed, 29
  SLD-, 269
determinism, 283
development, 102
diagram
  equational, 205
directed, 175
directed acyclic graph, 30, 117, 231
disjunction, 22
domain, 153, 154, 174, 201, 225
  of interpretation, 152
  Scott, 176
  substitution, 28
  sum, 180
tree, 24
E-algebra, 206
  free, 216
  initial, 216
Edinburgh syntax, 251
effectively computable, 314
efficiency, 351
embedding, 97
encoding
  of λ-calculus, 338
  of finite sets, 184, 340
  of logic, 327
  of pairs, 183, 301
  primitive recursive, 327
evironment, 69, 183, 197, 268
  logical, 268
equality, 141, 159, 207, 296
  extensional, 334
  generated, 209
equation, 206
  guarded, 214
  in λ-calculus, 55
equational proof
  by confluence, 240
  complexity, 241
  mixed, 240
equivalence, 22, 118
  elementary, 165
  proven, 118
evaluation, 68
excluded middle, 126
expansion
  of formula, 167, 172, 261, 270
of query, 270
extension
  conservative, 144, 162
definitional, 144
functional, relational, 143
parallel, 180
strict, 178

fact, 257
fairness, 71, 243, 246, 270, 282, 288
false, 54, 154
filter, 225
general, 225
fine complexity, 245
fixpoint, 17, 55, 176, 263, 290, 338
flounder, 294
forest, 108, 299
formula
  atomic, 111
clausal, 257
closed, open, 113
conjunctive, disjunctive, 139
existential, universal, 168
first-order, 111
free, bound variables, 113
Horn, 258
prenex, 138, 147
total provable, 334
frame, 165
free monoid, 9
function, 253
  Ackermann, 66, 307
base, 304
characteristic, $1_A$, 322
continuous, 176
$\lambda$-definable, 303
non-strict, 178
partial, 174, 177, 306, 316, 317
partial recursive, 317, 325
primitive recursive, 304
recursive, 307
semi-characteristic, 322, 325
sequential, 180
strict, 177, 179

$T$-definable, 319
total recursive, 317
Turing-computable, 313
Turing-computable partial, 316
universal partial recursive, 333
functor, 205
  in Algebra, 200
  in ML, 199
generalization, 61, 63
Gentzen, 114, 131, 331
goal, 257
Gödel, 128, 161, 319, 330
graft, 24, 25, 37
Hauptsatz, 134
height, 12
Henkin property, 161
Herbrand, 154, 159, 167–72, 227, 261
Herbrand model, 292
Herbrand universe, 154
Herbrand–Gödel system, 307
hierarchy
  fast, 85
Hilbert, 135, 331
history, 302, 305, 346
homogeneous, 288
Howard, 121
identity rule, 132
implementation, 146
implication, 22
impredicativity, 318
index, 332, 334
  branching, 42
induction
  complete, 16
  continuous, 181
monotone, 17
noetherian, see ind., well-founded
over terms, 12
Scott, 178
structural, 12
transfinite, 19, 331
well-founded, 15, 104, 246
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf-semilattice</td>
<td>62</td>
</tr>
<tr>
<td>well-founded</td>
<td></td>
</tr>
<tr>
<td>inference system</td>
<td>12, 49, 128, 213</td>
</tr>
<tr>
<td>type</td>
<td>59</td>
</tr>
<tr>
<td>initial</td>
<td>153, 187, 202, 216, 263</td>
</tr>
<tr>
<td>initial state</td>
<td>268</td>
</tr>
<tr>
<td>instance</td>
<td>61, 225</td>
</tr>
<tr>
<td>generic</td>
<td>74</td>
</tr>
<tr>
<td>integers</td>
<td>54, 252</td>
</tr>
<tr>
<td>interpretation</td>
<td>152, 201</td>
</tr>
<tr>
<td>abstract</td>
<td>59, 188-91</td>
</tr>
<tr>
<td>BHK</td>
<td>121</td>
</tr>
<tr>
<td>computed</td>
<td>187</td>
</tr>
<tr>
<td>formal</td>
<td>188</td>
</tr>
<tr>
<td>Kripke</td>
<td>165</td>
</tr>
<tr>
<td>of a signature</td>
<td>154</td>
</tr>
<tr>
<td>of connectives</td>
<td>154</td>
</tr>
<tr>
<td>of derivations</td>
<td>158</td>
</tr>
<tr>
<td>of first-order formulas</td>
<td>158</td>
</tr>
<tr>
<td>of propositions</td>
<td>156</td>
</tr>
<tr>
<td>of terms</td>
<td>155</td>
</tr>
<tr>
<td>interpreter</td>
<td>335</td>
</tr>
<tr>
<td>intuitionistic type theory</td>
<td>123</td>
</tr>
<tr>
<td>ISWIM</td>
<td>74</td>
</tr>
<tr>
<td>iterator</td>
<td>54, 81, 84, 108</td>
</tr>
<tr>
<td>judgment</td>
<td>12, 114</td>
</tr>
<tr>
<td>K</td>
<td>55</td>
</tr>
<tr>
<td>Kleene</td>
<td>48, 303, 317, 333, 337</td>
</tr>
<tr>
<td>Kleene sequence</td>
<td>188</td>
</tr>
<tr>
<td>Knuth–Bendix</td>
<td>237, 240, 298</td>
</tr>
<tr>
<td>Kripke</td>
<td>165</td>
</tr>
<tr>
<td>Kripke interpretation</td>
<td>165</td>
</tr>
<tr>
<td>λ-calculus</td>
<td>48-60</td>
</tr>
<tr>
<td>λΣ-term</td>
<td>72</td>
</tr>
<tr>
<td>λ-term</td>
<td>2, 32</td>
</tr>
<tr>
<td>closed</td>
<td>37</td>
</tr>
<tr>
<td>language</td>
<td>9</td>
</tr>
<tr>
<td>applicative</td>
<td>90</td>
</tr>
<tr>
<td>Dyck</td>
<td>218</td>
</tr>
<tr>
<td>first-order</td>
<td>L₂[X], 111</td>
</tr>
<tr>
<td>occurrence</td>
<td>24</td>
</tr>
<tr>
<td>second-order</td>
<td>112</td>
</tr>
<tr>
<td>lattice</td>
<td></td>
</tr>
<tr>
<td>complete</td>
<td>19</td>
</tr>
<tr>
<td>lazy</td>
<td>70, 72, 181</td>
</tr>
<tr>
<td>Lazy ML</td>
<td>75, 95</td>
</tr>
<tr>
<td>LCF</td>
<td>74</td>
</tr>
<tr>
<td>lemma</td>
<td></td>
</tr>
<tr>
<td>fixpoint</td>
<td>338</td>
</tr>
<tr>
<td>generalization</td>
<td>274</td>
</tr>
<tr>
<td>Newman</td>
<td>104, 105, 238</td>
</tr>
<tr>
<td>parallelogram</td>
<td>102</td>
</tr>
<tr>
<td>length</td>
<td>9</td>
</tr>
<tr>
<td>lexical status</td>
<td>100</td>
</tr>
<tr>
<td>list</td>
<td>78, 254</td>
</tr>
<tr>
<td>literal</td>
<td>139, 257</td>
</tr>
<tr>
<td>LK, LJ</td>
<td>131</td>
</tr>
<tr>
<td>logic</td>
<td></td>
</tr>
<tr>
<td>classical</td>
<td>126</td>
</tr>
<tr>
<td>combinatory</td>
<td>89, 135</td>
</tr>
<tr>
<td>equational</td>
<td>7, 213</td>
</tr>
<tr>
<td>intuitionistic</td>
<td>123, 165</td>
</tr>
<tr>
<td>minimal</td>
<td>115</td>
</tr>
<tr>
<td>L₀[X]</td>
<td>113</td>
</tr>
<tr>
<td>L₀[X]</td>
<td>111</td>
</tr>
<tr>
<td>Łukasiewicz</td>
<td>25, 150</td>
</tr>
<tr>
<td>macro</td>
<td>38</td>
</tr>
<tr>
<td>matcher</td>
<td>61, 64, 73</td>
</tr>
<tr>
<td>merge</td>
<td>230, 231</td>
</tr>
<tr>
<td>metaobject</td>
<td>28</td>
</tr>
<tr>
<td>metasymbol</td>
<td>27</td>
</tr>
<tr>
<td>metaterm</td>
<td>27, 206</td>
</tr>
<tr>
<td>mgu</td>
<td>see unifier, most general</td>
</tr>
<tr>
<td>minimization</td>
<td>304, 307</td>
</tr>
<tr>
<td>bounded</td>
<td>305</td>
</tr>
<tr>
<td>ML</td>
<td>74-88</td>
</tr>
<tr>
<td>model</td>
<td></td>
</tr>
<tr>
<td>Herbrand</td>
<td>159, 263</td>
</tr>
<tr>
<td>of λ-calculus</td>
<td>182-4</td>
</tr>
<tr>
<td>of theory</td>
<td>159</td>
</tr>
<tr>
<td>modularity</td>
<td>195-201</td>
</tr>
<tr>
<td>modus ponens</td>
<td>115</td>
</tr>
<tr>
<td>monotonicity</td>
<td>165</td>
</tr>
<tr>
<td>morphism</td>
<td>152, 201</td>
</tr>
<tr>
<td>elementary</td>
<td>165</td>
</tr>
</tbody>
</table>
image, 202
initial, 202
kernel, 203
$M(u)$, $M/u$, 23
multi-equation, 231
multiplcation, 101
multiset, 15, 99, 231, 241
$\mu \Phi$, $\mu \Phi$, 17

narrowing, 297
natural deduction, 114–30
negation, 23, 123, 133, 166, 286–95
negation as failure, 286, 293
NJ, 123
NK, 128, 158
NM, 115
non-determinism, 272, 354
don’t care, 229, 272
don’t know, 273, 282
non-terminal, 28
normal form, 53
head, 185
Kleene, 332
weak, 67
normalization, 51, 58, 229
of proofs, 240–6
notation
infix, 26, 80
prefix, 25
suffix, 26
NP-complete, 356
NU-Prolog, 251, 281
numeral, 318

object, 54
mutable, 255
occur check, 228, 233, 256
occurrence, 23, 30, 33, 258
discharged, 115
free, bound, 35
independence, 10
independent, 101
negative, positive, 112
of a formula, 112

$O(M)$, 23
operator
continuous, 18, 176
finitary, 17, 183, 263
monotone, 16, 263
optimization, 352, 359
oracle, 346
order, 57
complete, 175
flat, 175, 177
lexical, 15, 45
prefix, 10
simplification, 97
ordinal, 16, 18, 84, 264, 331
limit, 16

package, 197
generic, 199
pair, 54, 186, 309
lazy, 70, 181
non-strict, 70
strict, 70
parallelogram, 102
parallelogram property, 102
parser, 26
partial evaluator, 336
pattern, 61, 72
pattern matching, 64, 66, 72, 77
Peano, 141
Peirce, 132, 150
persistency, 243
polymorphism, 73, 83
$P\omega$, 183, 193
Prawitz, 122
predicate
definition, 252
prefix, 10
premise, 12
preorder, 10, 61, 97
subsumption, 61
presentation, 205, 217–8
problem
complete, difficult, 345
decidable, 322, 323
halting, 339, 347, 350
in class NP, 354
in class P, 353
instance, 321
reduction, 342
semi-decidable, 323, 325
solution, 345
tractable, 354
universal, 204, 216
program
completed, 291
definite, 252, 257
functional, 66
general, 292
logic, 252
with finite loops, 304, 314, 351
programming
functional, 3
logic, 8, 118
structured, 45
Prolog, 251
Prolog-Π+, 251, 256, 259
Prop[A], 23
proposition, 23, 113
atomic, 23
equational, 206
quantifier, 111, 158, 294
bounded, 163, 305
relative, 146, 163
quasi-order, see preorder
query, 252, 258, 268, 321
RAM, 353
rational, 41, 256
rational expression, 155, 277
recursion
complete primitive, 306
primitive, 84, 303
recursive, 314
recursive functional, 346
recursive path ordering, 99, 140, 247
recursive real, 341
recursor, 83, 108, 310
redex, 39, 65, 77
head, 185
marked, 101
reduction, 1, 100
of a problem, 345
polynomial, 353
Turing, 346
weak, 91
reduction graph, 51
reduction strategy, 53
refutation, 270
relation, 253
covering, 62
noetherian, 14
order, 10
progressive, 15
strict order, 10
well-founded, 14, 242
well-order, 16
renaming, 61
residue, 100, 101, 269
resolution, 2, 8, 172
resolution principle, 8, 265
restriction, 154
result, 269
rewriting system, 65
canonical, convergent, 239
compact, 239
noetherian, 66
reduced, 244
sound, 239
rpo, see recursive path ordering
rule, 257
derived, 119
inference, 12
local, 276
logical, 132
reduced, 244
rewrite, 65
selection, 275
structural, 131
satisfiability (|=), 159
Scheme, 88, 326
Scott, 174, 338
second-order arithmetic, 123
semantics, 6, 40
classical, 156
denotational, 174
of λ-calculus, 182–4
natural, 69, 109
of first-order logic, 152–9
structural operational, 69
separation, 268
sequent, 114, 131, 258
set
arithmetic, 343
c-o-r.e., 343
countable, 111
creative, 349
diophantine, 340
$K$, 339
productive, 349
recursive, 10, 322, 346
rec. enumerable, r.e., 323
recursively separable, 338
primitive recursive, 304
semi-recursive, 323, 346
$\Sigma_n, \Pi_n, \Delta_n$, 343
transitive, 16
universal, 334, 362
$\Sigma$-algebra, see algebra
sign
inherited, synthesized, 112
of a subformula, 112
signature, 11, 201
adequate, 198
applicative, 21, 182
extension, 220
functional, relational, 110
higher order, 57
in ML, 196
many-sorted, 21, 211, 214–6
predicate calculus, 110
skolemization, 147–8
SLD
-derivation, 269
-resolution, 265, 268
SLDNF-resolution, 293
$S$-mapping, 211
solution
to equation, 222
to query, 258, 262
solution state, 268
solvable, 185
sort, 20, 21, 211
soundness
strong, 269
$S$-set, 211
stability
under substitutions, 208
Standard ML, 74–88
state
narrowing, 297
resolution, 268
search, 273
solution, 266
success, 268
Strachey, 174
strategy
leftmost, 53
normalizing, 53
search, 276
weak reduction, 68
structure
algebraic, 195
data, 195
in ML, 197–201
subformula, 112
Gentzen, 133
subset
closed, 17
substitution, 28, 38, 61–5, 100, 153,
224–7
idempotent, 226, 268
subterm, 23, 30, 33, 41
immediate, 14, 16
success, 268
supercombinator, 93
superposition, 236, 297
suspension
of evaluation, 285
switching, 272
switching property, 268
symbol, 2, 9
  constant, 11
  functional, 11
  functional, relational, 111
syntactic form, 28
syntax
  abstract, 24, 33, 119
  concrete, 25
system of equations, 224
  finite resolved form, 224
  rational resolved form, 230
  reduced form, 224
tautology, 117, 140, 159
term, 24, 28, 206
  \(\lambda\)-calculus, 32
  applicative, 21, 90, 135
  closed, 28
  de Bruijn, 36
  finite, 2, 11, 24, 111, 181
  first-order, 11
  formal, 40, 181
  head normal form, 185
  infinite, 41
  \(\lambda\), 2
  linear, 73
  normal, 39, 49
  normalizable, 51
  partial, 40
  rational, 2, 41–3, 230, 256
  size, 51, 62
  solvable, 185
  strongly normalizable, 53
  typable, 59
  typed, 57
terminal, 28, 202, 263
termination, 174, 177
theorem, 13, 140
  Birkhoff, 211
  Church undecidability, 330
  compactness, 164
  completeness, 161–2, 214
deduction, 117, 136
Dershowitz, 98
fixpoint, 17, 176
Gödel incompleteness, 330
Herbrand, 170, 261
Herbrand–Skolem, 172
Knuth–Bendix, 237, 243, 244
Rice, 339
Rice–Shapiro, 340
s-m-n, 335
soundness, 160
strong completeness, 275
strong normalization, 58
strong soundness, 269
switching, 272
universality, 333
weak normalization, 58, 106
text, 140
complete, 143
equational, 206
essentially undecidable, 328
group, 141, 247
(in)consistent, 142, 161
model, 164
recursively axiomatizable, 327
set, 145
(un)decidable, 143, 328
thesis
Church, 314
Church, strong version, 315
Church–Kleene, 317
Cook, 354
\(T^k\), \(U\), \(\{e\}^k\), 332
\(T_P\), 263
trace, 276, 278, 312
transformation
  polynomial, 353
tree, 22, 24, 108, 299
  binding, 35, 116
  leaf, 24
  node, 24
  root, 24
  search, 273
true, 54, 154
truth table, 156
T_Σ[X], 153
Turing, 311, 346
Turing machine, 311
deterministic, 312, 352
2, 154
type
  atomic, 56
  concrete, 21, 79
  order, 57
  principal, 60
  recursive, 79
  simple, 56
type inference, 60
type product, 121
type scheme, 73
type sum, 121
type variable, 57
type-checking, 60
undecidability
  of β-conversion, 338
  of arithmetic, 330
  of halting problem, 339
unification, 7, 40
  finite, 223–30
  rational, 230, 256
unifier, 224
  most general, 226, 269
union, 54, 121
universality, 332–4
validity, 159
valuation, 155
value, 69
vanishing, 100
variable, 28, 32, 111
  applicative, functional, logic,
    sequential, 26
  assumption, unknown, 27
  free, 37, 113
  type, 59, 79
  unknown, 252
variant, 61, 268
variety, 207
well-founded inf-semilattice, 62
well-preorder, 97
word, 2, 9
word problem, 218
COMPUTATION AS LOGIC
René Lallement
Ecole Nationale des Ponts et Chaussées, Paris, France

Translated by John Plaice

Logic plays a dual role in computer science. First, computer science's foundations are studied using logic. Second, computer programming is itself a logical activity.

This book presents two programming paradigms from a logical point of view. Reduction, studied through the \( \lambda \)-calculus and rewriting systems, is the basis for functional programming, where the concept of type and the confluence and termination properties are crucial. Resolution is used in logic programming where soundness, completeness and the fairness of computation are important. Logic's inference systems and semantic methods are presented and used to study these properties, as well as computability.

Two recent programming languages, one functional, Standard ML, the other logic, NU-Prolog, are used throughout the work to illustrate these principles.