

An Overview of an Eductive Evaluation Mechanism for Logic Programs

by
David Rolston
AI Laboratory
ESL, Sunnyvale California
and
Tony Faustini[†]
Department of Computer Science
Arizona State University
Tempe, AZ. 85268
faustini@lu.eas.asu.edu

Abstract

We have developed a new eductive model for the execution of logic programs and we present an overview of this approach in this paper.

This new model provides a mechanism for the parallel execution of logic programs. The model is also capable of executing indexically extended Horn clause languages, such as Chronolog.

This new model is the first suitable for execution in a fine-grain, parallel, dataflow environment and has the following innovative characteristics: support for parallelism at the algorithmic level and at the instruction level; support for all major forms of Prolog parallelism including AND-Parallelism, OR-Parallelism, AND/OR-Parallelism, Stream Parallelism, and Unification Parallelism - or any combination of these. In addition, the model supports all-solution or single-solution evaluation, and permits straightforward exploitation of mixed levels of granularity in the implementation.

0 Introduction

In this paper we present an overview of an approach to the eductive evaluation of Prolog Programs. For the purposes of this paper the term “Prolog” refers “Pure Prolog” or Prolog without the extra-logical features such as assert/retract, cut, etc. Those interested in the details of the approach should read [Rolston 92, Orgun 92]. Unlike existing approaches to the parallel execution of Logic Programs, which are based on a coarse-grain, process-oriented execution of the traditional Prolog model of execution, we have developed an innovative approach based on the eductive model of computation. The eductive model of computation is based on the use of a

[†] Please address all correspondence to this author.

collection of autonomous processing elements for the parallel computation of referentially transparent languages. These values result from the evaluation of the Prolog program's logical variable in separate distinct contexts.

The development of the eductive model of logic computation is based on the following basic observations:

- For any given logical variable from a Prolog program the set of bindings assumed by that logical variable during traditional execution can be thought of as a set of related subvariables.
- Computing values (by whatever means) for the bindings that the subvariables would receive during traditional Prolog execution is equivalent to executing the original Prolog program using the traditional execution scheme.
- The relationships between a logical variable and the subvariables associated with it can be represented using *intensional* concepts - concepts that can be implemented well using eduction. Using an intensional approach a collection of related elements such as the different bindings of a logical variable $\{X_{b1}, X_{b2}, X_{b3}, X_{b4}\}$ can be thought of as simply being different variants of a single basic element, namely X . The various subelements occur when the basic element is evaluated in different contexts. Using this approach a single multi-valued variable is reduced to a collection of single-valued subvariables - where each of the single-valued subvariables is named by subscripting the original variable name with a *context descriptor* that identifies the context in which the original variable is evaluated.
- Computation within the eductive model consists simply of generating demands for variables at different contexts and satisfying those demands by evaluating independent Unilog equations using autonomous processing units.
- It is then straightforward to simulate the traditional parallel processing schemes for Prolog simply by altering the way in which demands are generated. This alteration has no impact on the eductive computation model (which is simply computing collections of independent values and is not even "aware" that it is executing any overall scheme).

It would be straightforward to develop the required variable associations if the following two problems could be solved:

1. The Variable Identification Problem -
It must be possible to uniquely identify each of the variables that would normally be used in an equivalent traditional Prolog execution (where each use of a variable is considered a separate unique variable).

2. The Variable Definition Problem -

It must be possible to establish formal expressions that define the computation required to compute bindings for each of the above-named variables (typically in terms of other variables) - where the "binding" of a variable is the value that the subject variable would receive during an equivalent traditional Prolog execution.

Given that the above conditions are met it would then be possible to construct a *Binding Computation Table* - a simple tabular presentation of variable names, defining expressions, and associated variable bindings as shown in Figure 1:¹

var1	Defining expression	Computed Binding
var2	Defining expression	Computed Binding
var3	Defining expression	Computed Binding
•	•	•
•	•	•
varN	Defining expression	Computed Binding

Figure 1 Tabular Presentation of Variable Binding Computation

Note that given the above approach there is a **single** binding associated with each variable (namely that binding that it would assume during the traditional Prolog solution process) and that the binding can be computed using the associated defining expression.

Adopting this view of the problem leads to the following observation:

If this collection of definitions were available the entire computation scheme could be reduced to a process of simply computing values for various variables (using the definitions in the definition column).² The sole task of a computational element would then be to evaluate the defining expression for a given variable³ and the entire process would be free of any need for external sequencing or control.

¹ Computing bindings using this tabular approach is equivalent to executing Prolog using the traditional approach.

² At the start of execution the value of every variable would be "unknown".

³ and filling the result into the "binding" column for the subject variable. Note that this "storage" of the result is not a fundamental part of the process and is **not required** for computation (i.e., computation could proceed to completion without any recording of results). It is recorded only as a convenience for reference in future computation (i.e., caching of results).

Applying this simple concept is in itself novel and serves to "unravel" the otherwise complex solution process into a collection of simple independent activities. (Note that this is an elegant solution to the problem of dividing and allocating computational activity that has always been a central issue in all forms of parallel processing [Huang, 1984]). Most existing parallel Prolog implementations expend a great deal of energy in addressing this problem (typically through complex process allocation and often require programmer annotation and intervention to solve the problem).

The single most powerful aspect of this approach (from a parallel processing viewpoint) is that it allows independent processing without any need for sequencing or overall control. Given this independence, and the resulting freedom from sequencing requirements, it is possible to select any given value for execution (and thus, in effect, to begin execution at any point - or at every point - in the traditional solution process).

This leads to an additional observation:

To produce maximum parallelism it would be possible (given sufficient computational resources) to simply request computation of all values simultaneously [i.e., fully eager evaluation].

The flexibility provided by this *minimalist approach* is the basis for the definition of an execution model that supports execution using any overall organizational scheme - mirroring all known forms of parallelism ⁴ (e.g., all solution, v.s. single solution, and is capable of OR-Parallel, AND-Parallel, and AND/OR-Parallel evaluation).

In fact this approach is so flexible that different forms of parallelism and solution modes could be selected dynamically in response to different specific queries.

This is a very important distinction because the practical usefulness of any given form of parallelism to solve any given problem is very strongly a function of the solution mode employed. For example, in an all-solution execution of a program with multiple solutions all of the processors in an OR-Parallel execution perform useful work whereas in a single-solution execution most of the OR-Parallel execution is wasted effort.⁵ Because of the importance of these issues a great deal of attention in the Prolog community is focussed on the virtues of various schemes. For example,

⁴ Examples of several different execution schemes are presented later in this paper.

⁵ These distinctions are discussed in [Fagin, 1987, pg. 183].

the Aquarius project [Despain, 1984], is based on the stated position that single-solution Prolog is unequivocally superior. (Much of the motivational sections of [Fagin, 1987] are spent justifying this position.) [Li, 1986], on the other hand, takes an equally strong position in favor of All-Solution Prolog. Similar arguments exist for OR-Parallel processing, AND-Parallel Processing, etc.

Although various projects advocate various different approaches there is one aspect that is common to all: *the basic computational scheme is a fixed characteristic of the model that results from a fundamental design decision made early in the project.* It would, for example, be very difficult to modify Aquarius [Despain, 1984] to use an all-solution scheme.

In addition, the flexible approach advocated in this research has many advantages over those found in the literature:

- It provides micro-level parallelism in addition to macro-level parallelism
- It incurs no process-generation overhead.
- It provides superior forms of basic parallelism (i.e., it addresses the OR-Parallel binding environment problem and the AND-Parallel binding conflict problem).
- It supports full (AND/OR) parallelism.
- It is suitable for execution in a highly parallel architecture composed of simple processors (i.e., dataflow, education.)
- Unlike traditional Prolog and most existing parallel implementations this approach (using an appropriate driver) is complete.⁶

Again, the flexibility of the model, which results from the minimalist approach, allows execution using any known form of parallelism, even dynamic selection of the parallelism mode.

1 Execution Example

The greatest impediment to the realization of this approach is, of course, the challenge of developing techniques for identifying unique variables and associated definitions (i.e., solving the variable definition problem and the variable description problem). It would appear to be very difficult if not impossible to reduce the entire operational process to independent static definitions.

⁶ Note that there are existing models (e.g., the Reduce-OR Process model) [Ramkumar, 1991] that are complete.

The lack of this capability has prevented past efforts from following established dataflow principles.⁷

One of the contributions of this research, and one that empowers the basic approach, is the description of a scheme for developing the required definitions and descriptions. The development of this scheme is best understood by starting with the simple example shown in Figure 3:

Example Prolog program:

```
P1(A) ← P2(A), P3(B).
P2(K) ← P4(K).
P3(L) ← P5(L).
P4(6) ← .
P5(7) ← .
```

Query: P1(Q)?

Associated AND/OR Tree:

From an inspection of this example it is clear that the complete set of variables (for which bindings are to be computed) consists of: *A*, *B*, *K*, *L*, and *Q*. The desired computational table then can be formed as in Figure 3:

A	?	Unknown
B	?	Unknown
K	?	Unknown
L	?	Unknown
Q	?	Unknown

Figure 2 Binding Computation Table for Example Program

⁷ For example, [Wise, 1986, pg. 13] sets its goal as the application of the dataflow model for parallel Prolog execution. It acknowledges that the established dataflow model would require fine grain non-sequential operations. In reality, however, the EPILOG project is eventually based on coarse grain **sequential** process-oriented execution (requiring user annotation) with destructive assignment due to an inability to identify the appropriate low-level definitions.

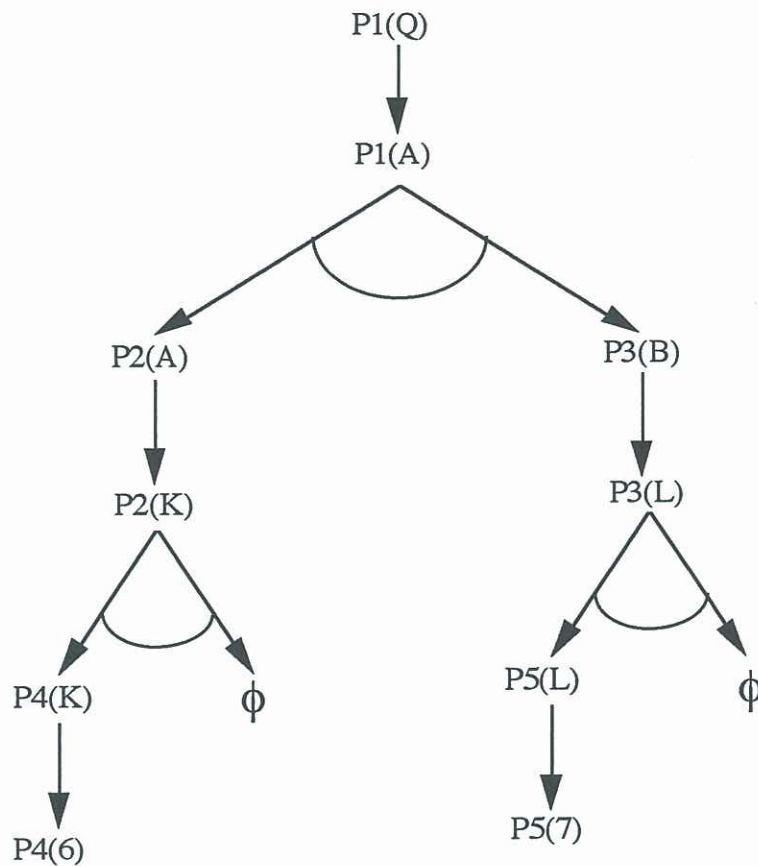


Figure 3 Variable Definition Example

The next challenge is then to develop definitions for each of these variables, where each definition is required to be a *mathematical statement* composed entirely of well-defined operations that use only other variables (or ground terms) as operands. Note that by using this approach there is no need for process generation or complex multi-stage sequential operation. *Moreover, there is no sense of a binding environment that must be stored and tracked or located.* Variables are uniquely defined and each value is computed at a single place.

The next critical insight is to recognize that the required definitions for the subject example can be developed by “simulating” the computation that would occur during a traditional Prolog execution:

Given the query $P1(Q)$? -

- $P1(Q)$ would unify with $P1(A)$ (binding Q to A).
- The subgoals $P2(A)$ and $P3(B)$ would be pushed onto the stack for execution.
- $P2(A)$ would be selected for execution and would unify with $P2(K)$ (binding A to K and pushing $P4(K)$ onto the stack).
- $P4(K)$ would be selected for execution and would unify with $P4(6)$. K would be bound to 6 in this process and - because $P4(6) \leftarrow .$ is a unit clause - execution would return to the caller without additional goals being added to the stack.
- The top of the stack ($P3(B)$) would then be selected for execution and would unify with $P3(L)$ (binding B to L) and pushing $P5(L)$ onto the stack).
- $P5(L)$ would then be selected and unified with $P5(7)$ (binding L to 7).

An analysis of this execution reveals that (in the case of this simple example) the required variable definitions are established simply by noting that each variable will be bound to the term that is located at the next lower level in the AND/OR Tree (e.g., A binds to K , K binds to 6 , etc.).⁸

Given this observation the entries in the Binding Computation Table can be completed as shown in Figure 4.

A	=K	Unknown
B	=L	Unknown
K	=6	Unknown
L	=7	Unknown
Q	=A	Unknown

Figure 4 Binding Computation Table Including Definition Fields

Note that even given this simple example it is possible to demonstrate the ease of using several different overall execution schemes (or *drivers*) to perform evaluation.

For example, in a *fully lazy* scheme a demand (i.e., request) would be issued for the value of Q (the top-level binding).

⁸ The actual definitions used in Unilog are, of course, much more complex. They are presented in detail [Rolston.92]

To compute a value for Q the definition of Q would be evaluated. Evaluation of this definition ($=A$) identifies a need for the value of A . Computing the value of A results in a demand for K . The definition of K ($=6$) involves only a ground term and the value is therefore immediately returned (and placed into the value field for K). Given this value it is now possible to complete the computation of A and, given A , it is then possible to compute Q .⁹

To execute in an AND-Parallel mode demands for A and B would simply be issued simultaneously. To execute in a *fully eager*, all-solution mode, demands would be simultaneously issued for all program variables.

Note that given this approach the basic underlying computation mechanism remains unchanged - the computation of Q still demands A - but in this case the value of A would be immediately available due to the parallel execution. Note also that it would be perfectly correct (although probably not especially useful) to demand the values in random order.

3 Solving the Variable Identification Problem

The solution of the variable identification problem requires that a technique be identified for uniquely identifying each "variable" used in a traditional Prolog execution - where a "variable" is any item that receives a binding during execution.

Although the identification of variables in the above example program is straightforward to fully solve this problem it must first be recognized that the concept of a "variable" is applied at three different levels in Prolog.

3.1 Basic Source Variable

The set of *Basic Source Variables* (BSV) is simply the set that results from an enumeration of those that occur lexically in a program.¹⁰ Figure 5 shows an example of BSV formation.

⁹ Note that the fully lazy mode results in an execution sequence that is equivalent to traditional Prolog execution

¹⁰ Given that the variables in the program have been systematically renamed to ensure lexical uniqueness.

Prolog Source

```

P1(X,H) ?
P1(K,J) ← P2(K,J), P3(K,J).
P1(L,M) ← P3(L,M), P4(L,M).
P2(4,7) ← .
P2(2,3) ← .
P3(5,6) ← .
P3(4,7) ← .
P4(5,6) ← .
P4(6,7) ← .

```

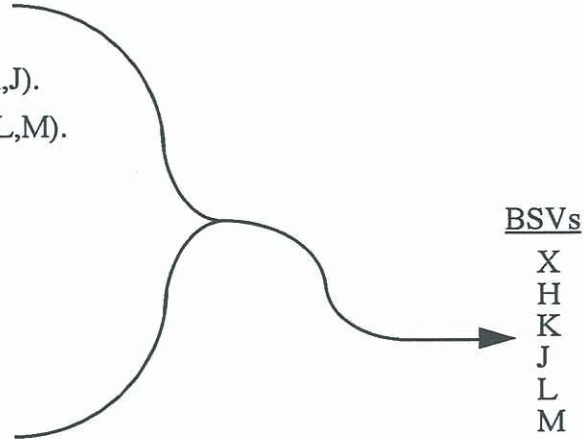


Figure 5 BSV Generation from Prolog Source

3.2 Source Variable Components

At the next lower level of detail it must be recognized that any given BSV is used in many different senses in a given traditional Prolog execution. Even if there were only a single defining clause for the given BSV (i.e., without regard for the fact that the variable receives different bindings when unified with different clauses) each of the different senses in which a BSV is used is recognized as a different *Source Variable Component* (SVC).

For example, a given variable, say the variable X , is instantiated with a binding when a goal is unified with the head of the clause in which it appears. This binding is then passed to the various subgoals. Based on the results of subgoal execution the binding of X may be revised and ultimately a final binding is returned to the calling routine. Thus several different components (SVCs) of X must be recognized as distinct elements.

Based on the above discussion, the following are examples of different possible SVCs for the BSV X :¹¹

```

Xunif = X as it results from unification.
Xs1s = X as it is sent to subgoal 1.
Xs1r = X as it is returned from subgoal 1
•           •
•           •

```

¹¹ These are simply example SVCs that are introduced here to demonstrate the fundamental concept. The actual process of defining SVCs is quite complex as described in [Rolston 92]

•
 $X_{sns} = X$ as it is sent to subgoal "n".
 $X_{snr} = X$ as it is returned from subgoal "n".
 $X_{ret} = X$ as it is returned to the caller.

Several examples of SVCs that would be generated for the example BSVs are shown in Figure 6. (Note that the relationship between BSVs and SVCs is static. Every set of BSVs can be decomposed into an associated set of SVCs - according to a given set of decompositional rules [Rolston 92])

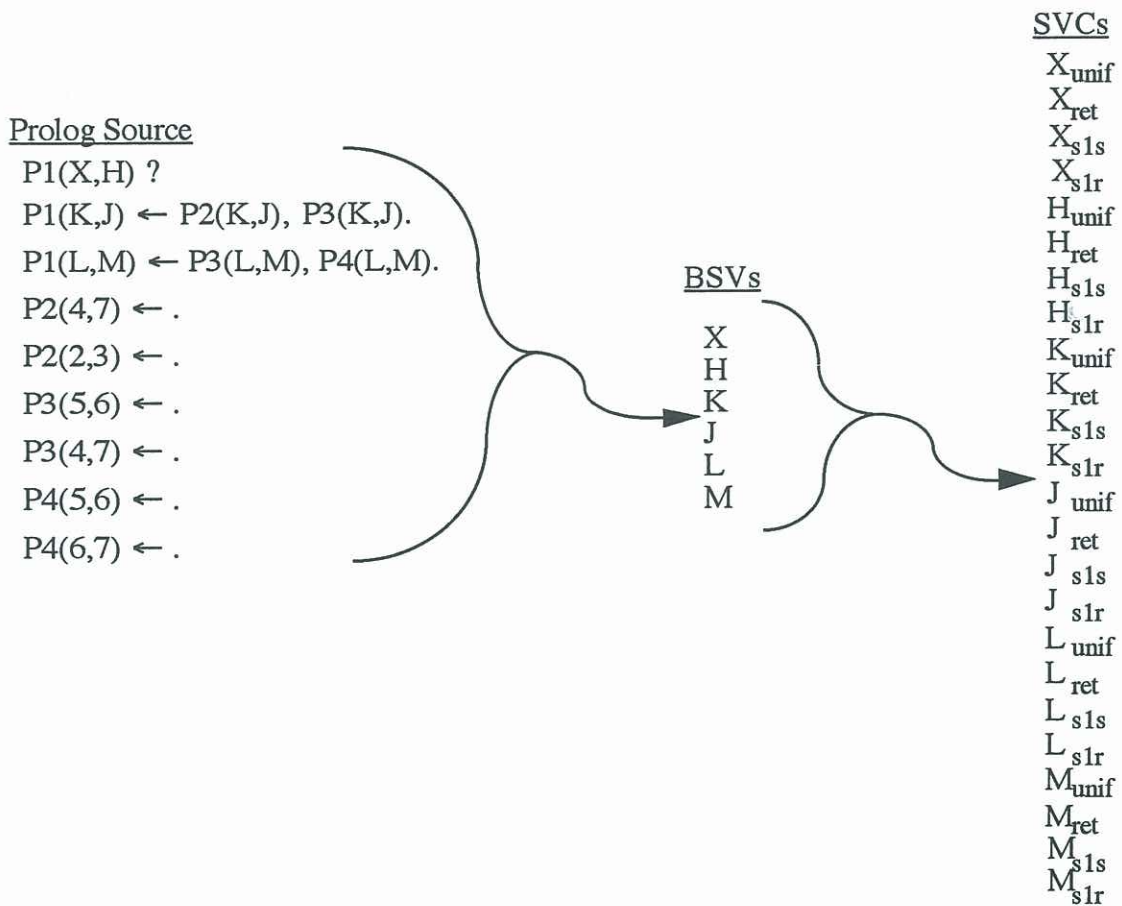


Figure 6 BSV to SVC Conversion

3.3 Dynamic Component Versions

Dynamic Component Versions (DCV) represent the various bindings that are assumed by an SVC during the traditional execution process. The need for DCVs results from the following:

As described above, any given BSV can be decomposed into its component SVCs through a static analysis of the subject program. Each SVC represents a different role played by the BSV in the traditional execution process. For example, given the source program from Figure 7, the SCV X_{ret} represents the final value returned for X . Note, however, that although the role played by X_{ret} is invariant¹² it actually assumes several different values based on the matching of a goal with several different clauses. For example, when $PI(X,H)$ is matched with $PI(K,J)$, X_{ret} will be bound to 4; when $PI(X,H)$ is matched with $PI(L,M)$, X_{ret} will be bound to 5.

Thus it is necessary (and sufficient) to identify each separate version of each SVC (i.e., separate DCVs). The DCVs represent the values that are actually used for computational purposes in Unilog.¹³

Figure 7 shows an example of DCVs (where individual DCVs are represented as simple subscripted values - based on the assumption that each SVC has three distinct value versions).¹⁴

3.3.1 Defining Dynamic Component Versions

Given this scheme to develop descriptions of DCVs from SVCs the question remains as to how DCVs are to be defined (i.e., how to describe different versions of SVCs). The solution of this problem is based on the following insight:

To represent multiple versions of a given SVC a form of *intensional logic* is used to represent separate versions.¹⁵ This approach can be understood as follows:

¹² X_{ret} , even though it assumes multiple values, always represents X in the sense of the final return value for X .

¹³ That is, the DCVs are the elements whose values are being computed in the Binding Computation Table originally identified in the Introduction.

¹⁴ The actual representation of separate DCVs is quite complex, see [Rolston 92].

¹⁵ As is explained later in this section, intensional logic is a form of logic in which the truth state of an expression is a function of the context in which it is evaluated.

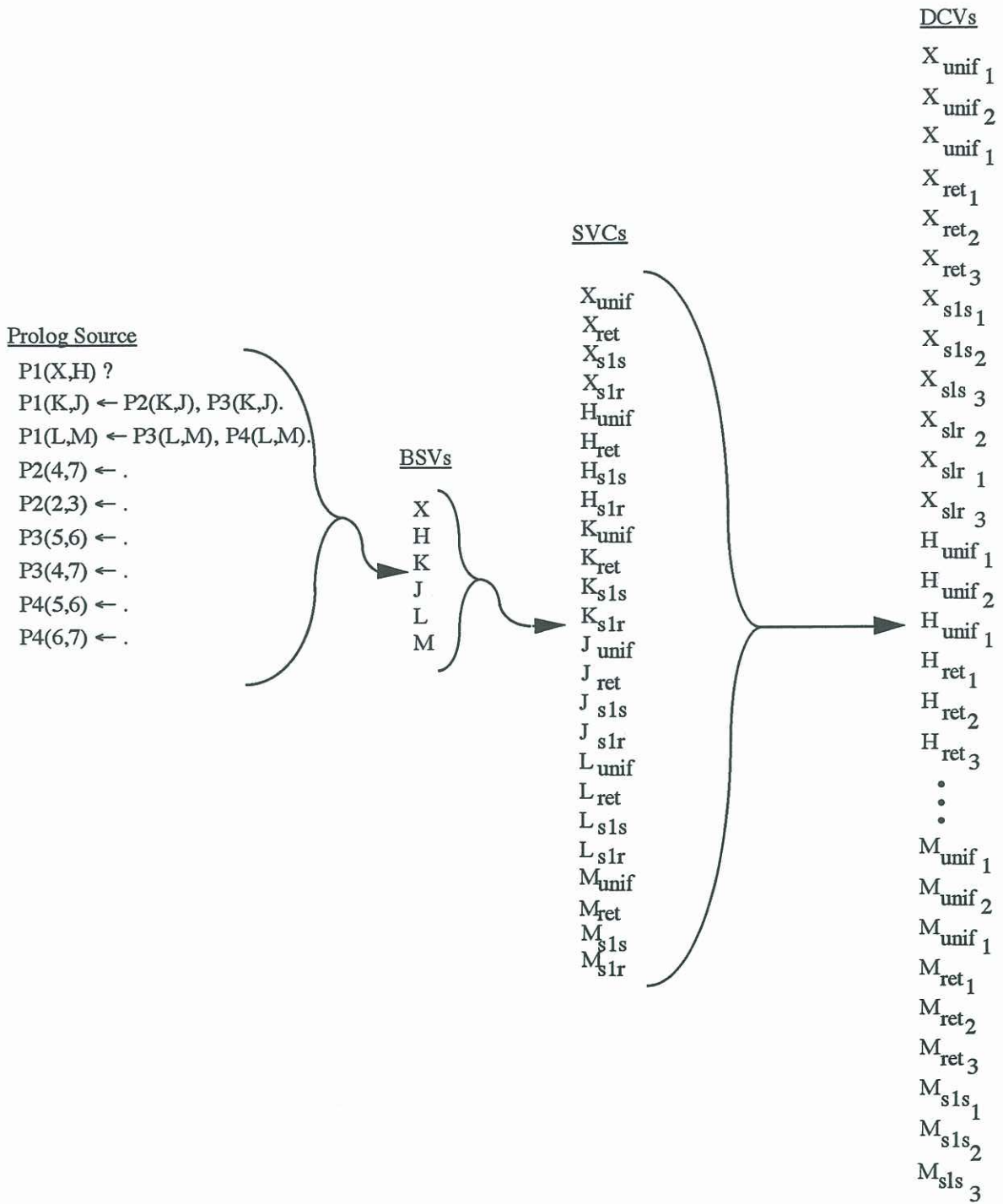


Figure 7 SVC-to-DCV Development

Intensional logic [van Benthem, 1988] is a form of tense logic [McArthur, 1976] which is itself a type of modal logic [Chellas, 1980]. Intensional logic is a form of logic in which the value of a given expression is in some sense dependent on some specified context.¹⁶ This concept is applied as follows: In classical logic the meaning of an expression is that which it denotes. In intensional logic this concept is expanded to include the idea of multiple meanings. Specifically, in intensional logic the *extension* of an expression is its meaning in a specific context (e.g., the meaning of the word "set" when used in a mathematical context v.s. a tennis context). The *intension* of an expression is its full or complete meaning, namely, a collection of its extensions.¹⁷

To apply this approach in the Prolog domain it is first necessary to view each segment of a traditional solution process as a separate context (or *world*).¹⁸ The final step in the resolution of the variable identification problem is the identification of separate DCVs as being the individual instances of an SVC that result from viewing a given SVC in different contexts.

Given this approach each DCV is then a unique single-valued variable.

The collection of DCVs is then the desired set of unique variables and identification of the set of DCV defining expressions is the solution to the variable identification problem.

The *extension* of an SVC is its meaning in a given world (directly analogous to the concept that a given word has a specific meaning or sense in which it is used in a given conversation or context). The *extension* is a DCV - formed from the SVC. The complete or total meaning of the SVC is its *intension* - the collection of all its extensional meanings (e.g., the "full view" of the meaning of the word "set" is the collection of the eighteen or so recognized senses in which the word is used). The intension will represent the set of values that would be assumed by the SVC during a complete traditional execution.¹⁹

Thus DCVs can be thought of as subscripted versions of the root SVC rather than as simple symbols (i.e., X_1, X_2, \dots, X_n represent the different DCVs formed from the SVC X). Note that this concept is directly applicable in eductive evaluation.

¹⁶ Intensional logic was originally developed as a mechanism for representing natural human language. It is appropriate in that application because the meaning of a given word is often a function of the context in which it is used.

¹⁷ Note that the various extensions are not totally unrelated - each extension is a different version of the underlying concept.

¹⁸ Techniques for identifying individual contexts are introduced [Rolston92].

¹⁹ Note that this approach provides another significant contribution - it allows a complete (*possibly infinite*) value - the intension - to be computed piecewise in fully defined individual elements (the extensions). This provides the basis for implementing formally defined processing of infinite object - See [Rolston 82].

3.3.2 Identification of Contexts

It is clear that for the above representation scheme to be useful it must be possible to identify the appropriate contexts. Unfortunately, it is also clear that in reality the actual contexts used in Unilog must be much more complex than the simple integers shown above (because it is not possible to develop a simple list of all possible contexts through a static analysis of the program).

Due to the structure that is inherently present in a Prolog execution process (i.e., it is a structured search of a well-defined AND/OR tree). This problem can be addressed in the following way: The set of all possible contexts is viewed as a (possibly infinite) space of instances - the *intensional space*. An example is shown in Figure 8.

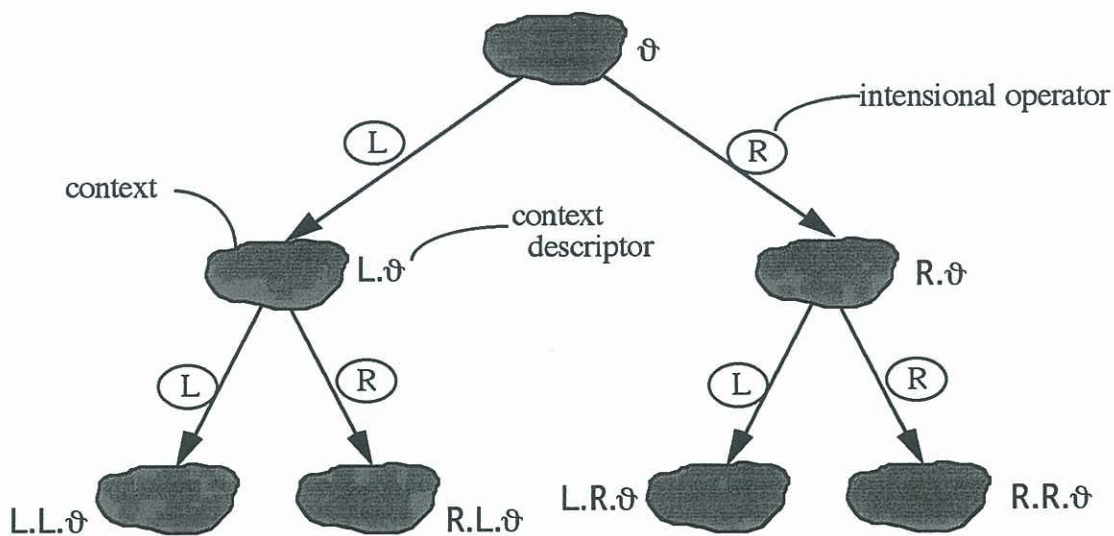


Figure 8 Intensional Space Example

Note that the various worlds are not simply randomly arranged; rather there are specific relationships that exist among them.

Although it may not be possible to list all the contexts from the intensional space it is always possible to identify any given world by identifying the transition required to reach it from some other specified world.

Specified *intensional operators* are used to identify these transitions - where each operator accepts the *context descriptor* from a given context (the one in which it is being evaluated) and produces a new descriptor based on a *context descriptor manipulation rule* (or *tag manipulation rule*) that is associated with the operator. (These rules add characters to - or remove characters from - the input context descriptor.)

For example, consider the simple space shown in Figure 8. In this example the following two operators would be used to generate the subject space (starting at the top level context which has a tag of \varnothing).

<u>Operator</u>	<u>Tag Manipulation Rule</u>
-----------------	------------------------------

\mathcal{R}	Let: the context in which the operator is evaluated be represented as \varnothing , Then: the new context = $\mathcal{R}.\varnothing$.
---------------	--

\mathcal{L}	Let: the context in which the operator is evaluated be represented as \varnothing , Then: the new context = $\mathcal{L}.\varnothing$.
---------------	--

Note from this example that the context descriptors are not simple atomic values; rather they have a significant internal structure that results from the generative application of the intensional operators.

Thus, the final step in solving the variable identification problem is to recognize a set of DCVs - the complete collection of SVCs tagged with the context descriptors of all the contexts included in the intensional space for the subject program. A simple example of this situation (using the intensional operators introduced above) is shown in Figure 9.

Computation of $X_{ret} \alpha$

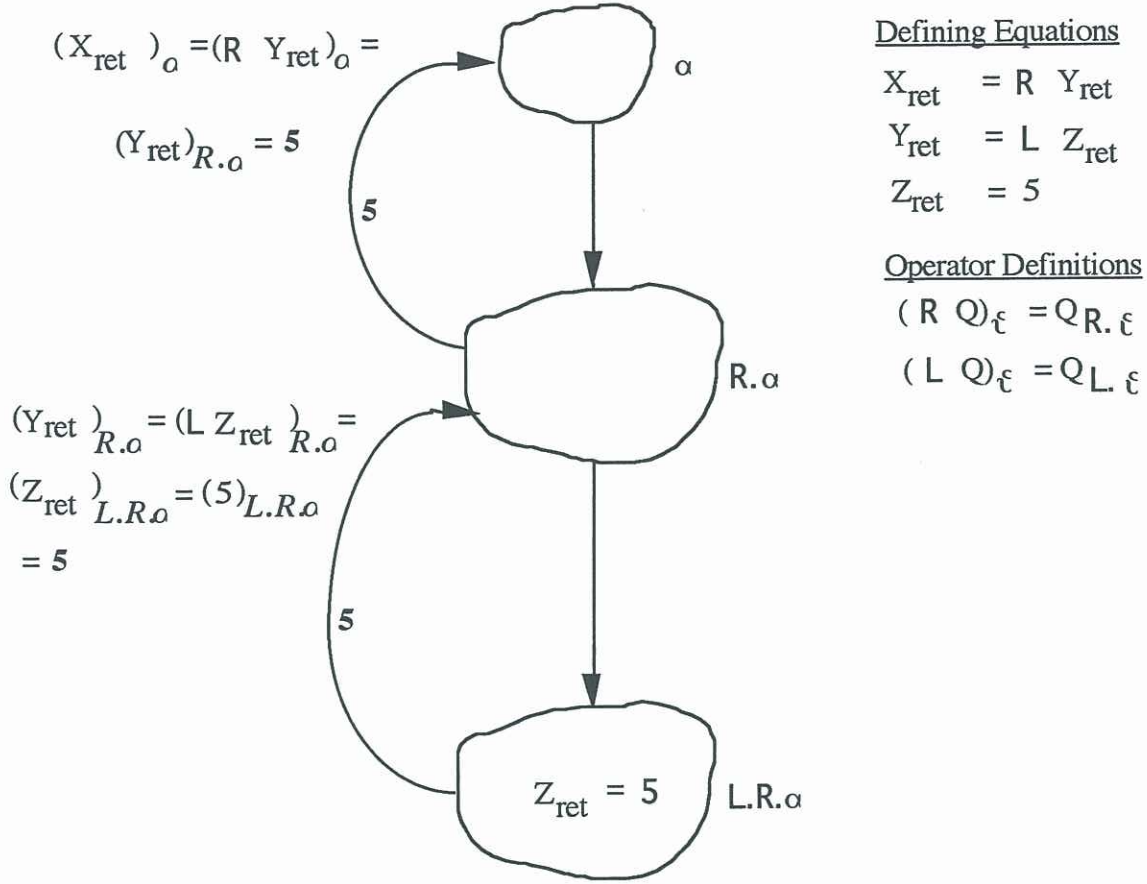


Figure 9 DCV Generation

This example can be understood as follows:

A request is issued for the single unique value of $X_{ret} \alpha$. To compute the value of X_{ret} in context α it is necessary to find a definition for X_{ret} .²⁰ Looking up the definition of X_{ret} in the list of defining expressions it is found to be the following: $X_{ret} = R Y_{ret}$.

Thus, to find the value of X_{ret} in context α it is simply necessary to find the value of $R Y_{ret}$ in context α [i.e., $(X_{ret})_{\alpha} = (R Y_{ret})_{\alpha}$]. To find a value for $(R Y_{ret})_{\alpha}$ the intensional operator

²⁰ Note that the defining expression for X_{ret} can be used to evaluate X_{ret} in any context of the intensional space.

\mathcal{R} is evaluated with an input context descriptor of α (using the definition of \mathcal{R}). This computation yields $(Y_{ret})_{R.\alpha}$ [i.e., $(\mathcal{R} Y_{ret})_{\alpha} = (Y_{ret})_{R.\alpha}$].

To find the value of $(Y_{ret})_{R.\alpha}$ it is necessary to use the definition of Y_{ret} in the $R.\alpha$ context [i.e., $(Y_{ret})_{R.\alpha} = (\mathcal{L} Z_{ret})_{R.\alpha}$]. Using the definition of \mathcal{L} the expression $(\mathcal{L} Z_{ret})_{R.\alpha}$ is reduced to $Z_{ret}_{L.R.\alpha}$. The definition of Z_{ret} (i.e., $Z_{ret} = 5$) is used to solve $(Z_{ret})_{L.R.\alpha}$ yielding a value of $(5)_{L.R.\alpha}$ [i.e., $(Z_{ret})_{L.R.\alpha} = (5)_{L.R.\alpha}$].

The value of $(5)_{L.R.\alpha}$ reduces simply to 5 which requires no further computation.²¹ Thus the value of Z_{ret} is found to be 5 using the following evaluation chain:

$$(X_{ret})_{\alpha} = (\mathcal{R} Y_{ret})_{\alpha} = (Y_{ret})_{R.\alpha} = (\mathcal{L} Z_{ret})_{R.\alpha} = Z_{ret}_{L.R.\alpha} = (5)_{L.R.\alpha} = 5.$$

This form of evaluation (using realistic SVCs, intensional operators, and context descriptors) is the basis for computation in Unilog. A detailed presentation of the actual context descriptors and operators used in Unilog are given in [Rolston 92].

3.4 Solving the Variable Definition Problem

To solve the variable definition problem it is necessary to develop a scheme for producing formal definitions for each DCV in a given program (i.e., the formal defining expressions that are evaluated to compute values for individual DCVs).²²

It is clear from the computational examples given earlier that the set of definitions becomes the "program" that is actually used as the basis for computation.

Using this approach each defining expression relates to only a single variable from the Prolog program (i.e., a single BSV).²³ Thus to describe the values assumed by the variables X and Y in the Prolog clause $P1(X,Y) \leftarrow P2(X,Y), P3(Y,X)$ it is necessary to produce independent Unilog equations that define X and Y separately:²⁴

²¹ A constant reduces to itself when evaluated in any context.

²² The subject expressions must be declaratively valid mathematical statements.

²³ Note that Bic [Bic 84] restricted his model to unary predicates to achieve a similar result. Unfortunately he provided no mechanism to reduce Prolog to a unary form.

²⁴ As described above, multiple defining expressions are actually required to fully define the values assumed by even a single variable from the Prolog program (i.e., one equation for each SVC that is decomposed from a given BSV).

To develop single-variable descriptive clauses it is necessary to transform (i.e., compile) the Prolog program into an equivalent set of independent equations- using an appropriate language - that define single variables.

Following this approach Prolog source are input into a compiler which are then be compiled into Unilog, as shown in Figure 10. Unilog would then be evaluated by an eductive evaluator called Unival (following the computational approach described previously).²⁵

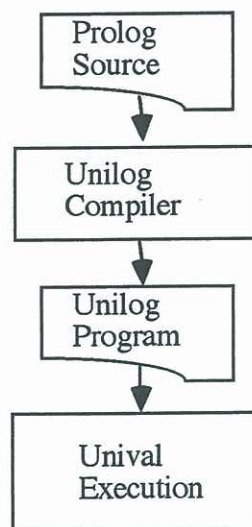


Figure 10 Prolog - to - Unilog Compilation

Specifically, each Unilog equation (i.e., transformed clause) is a valid formal logic clause that represents the computation of a binding for a *single SVC*.²⁶ During the compilation process the original program variable (i.e., the BSV) is "promoted" to become the root of the predicate name in the head of the Unilog clause. (For example, the Unilog clause $X1\beta \leftarrow Y1\delta$ is used to describe the computation for the SVC $X1$ - where X originally appears as an argument in a literal (say $P1(X,Y)$) in the source Prolog program. This intuition leads to the following important insight:

²⁵ As noted in the introduction, development of Unilog is the focus of this research. Any pure Prolog source can be transformed to an equivalent Unilog program.

²⁶ The evaluation of the SVC-oriented equation in different contexts produces values for the SVC in different contexts - i.e., for different DCVs.

More specifically, the expression $B\omega_\pi$ (in a Unilog program) is equal to some value v iff B (a source program variable acting in the ω role) would have been bound to v within the specified place π of the original source program execution.²⁷

The quasi-stream of values that results from the evaluation of $B\omega_p$ in different contexts corresponds to a subset of the values assumed by the ω component of the original source program variable B during traditional execution.²⁸

(Note that using this approach the original program predicates disappear but their "ghosts" live on as constraints between the Unilog equations. Note also that this representation serves to "flatten" and normalize all programs into a common form - namely a collection of nullary equations.)

For example, $XI\omega_\pi = 3$ represents the fact that the ω component of the XI variable (a variable from the source program used in the ω role) is bound to 3 within context π . (Of course, it is also possible that XI could be bound to a nonground term.) Again, to reiterate the fundamental concept, the equations in Unilog represent the variables from the source and computing values for the Unilog equations is equivalent to finding binding for the corresponding variables in the source program.²⁹

In general the body of each Unilog equation will consist of a set of *intensional subgoals* where each intensional subgoal consists of an intensional operator and a traditional nullary literal. Evaluation of an intensional subgoal - from within a given context - has the effect of evaluating the traditional literal - relative to the context identified by an evaluation of the intensional operator.

As noted previously, using intensional operators defined earlier, evaluation of the expression $(Right\ XI\delta)_{L.S}$ would result in the evaluation of $(XI\delta)_{R.L.S}$ (i.e., would have the effect of solving $XI\delta$ in the $R.L.S$ world and returning the value to the $L.S$ world). Given this approach, computing a solution to a Unilog goal is equivalent to computing a binding for the original program variables. Thus, the equations described above represent the fundamental relationships between the basic program variables.

²⁷ $B\alpha$ is one SVC formed from the BSV B . Others (such as $B\beta$, $B\delta$, and $B\epsilon$) will also be generated see [Rolston 92].

²⁸ This discussion is based on the simplifying assumption that all source program arguments are variables (for purposes of illustration only). The general case is considered in [Rolston 92].

²⁹ Unilog equations are sometimes referred to as "variables" simply because they are nullary equations acting in the place of Prolog variables.

5 Conclusion

This paper gives an overview of an approach to the eductive evaluation of “Pure Prolog” programs. A complete description of this approach is given in [Rolston 92] in which the development of Unilog and Unival are addressed. In particular the following issues are dealt with:

- Definition of SVCs to be generated from BSVs (i.e., in answer to the question, "Into what components should a BSV be broken?");
- Definition of Unilog operators;
- Development of an overall scheme for compiling “Pure” Prolog into Unilog;
- Definition of specific Prolog-Unilog transformations;
- Definition of the format of context descriptors and intensional space;
- Definition of an eduction-based evaluation system;
- Definition of example drivers;
- Description of the use of the Unilog base for providing extended logic functionality (Chrolonolg).

We recommend that those interested in the approach read [Rolston 92].

5 Bibliography

Bic, L., “Data-Driven Model for Parallel Intrepretation of Logic Programs,” Proceedings of the International Conference on Fifth Generation Computer Systems, pp 517-523, 1984.

Chellas,B., “Modal Logic” Cambridge University Press, 1980.

Despain,A., Pratt, Y., “The Aquarius Project Digest of Papers”, Proceedings of Compcon 84, pp364-367,1984

Fagin,B.S., “A Parallel Execution Model for Prolog,” Report # UCB/CSD 87/352, University of California at Berkeley, CA., 1987.

Hwang,,K., and Briggs,F., “Computer Architecture and Parallel Processing”, McGraw-Hill, New York, NY, 1984

Li,,P., Martin, A., “The Sync Model: A Parallel Execution Model for Logic Programming,” Proceedings of the 3rd IEEE Symposium on Logic Programming, pp 223-234, IEEE, 1986.

McArthur,, R., “Tense Logic”, Reidel, Boston, MA, 1976

M.A.Orgun and W.W.Wadge, “Intensional Logics for Programming” pp 23--50.
in , Theory and practice of temporal logic programming.
L. Farinas del Cerro and M. Pentonnen, editors,

Oxford University Press, 1992.

Ramkumar,,B., Kale.L., "An Abstract Machine for the Reduce-OR Process Model for Parallel Prolog", in Ramam, S., Chandrasekar, R., Anjaneyulu, K. (Eds), Knowledge Based Computer Systems , Narosa Publishing, New Delhi, India, 1990.

Rolston,D., "Parallel Logic Programming Using an Intensional Model of Computation", Ph.D. Dissertation, Department of Computer Science, Arizona State University, May 1992.

van Benthem,, J., "A Manual of Intensional Logic", Second Edition, Lecture Notes No. CSLI-1, Stanford University, 1988.

Wise,M. "Prolog Multiprocessors", Prentice-Hall, New York, NY, 1986.