Less Restrictive Constructs for Structured Programs*

David Lorge Parnas          William W. Wadge

Abstract

The syntax and formal semantics of new control constructs that resemble Dijkstra’s guarded commands [1] is given. Like programs built using the standard “structured programming” constructs, programs built using these constructs are easily parsed into a hierarchy of components and the meaning of the constructed program is a simple function of the semantics of its components. Program structures that were previously considered heretical by advocates of structured programs, such as multiple-entrance programs and side-effects in Boolean expressions, are shown not to complicate either the syntax and semantics. The use of the new constructs is illustrated on some small examples.

1 Introduction by William W. Wadge (2001)

The paper presented here is being published for the first time—even though it was written about fifteen years ago. It is the followup to Dave’s “A generalized control construct and its formal definition” [4], in which he introduced the $it$–$ti$ construct.

On a purely technical level, these papers present this construct as a generalization of Dijkstra’s $do$–$od$ command, together with a simple relational semantics. However, Dave makes it clear from the start, in his first paper, that this is done to support three very general and very important points:

- that language semantics should use simple mathematics,
- that one good general structure can be better than many specialized ones, and
- that nondeterministic programs can be simpler than deterministic ones that solve the same problem.

Each of the three points passes Dave’s own “tautology” test: If we negate them, the results are not obviously and trivially false.

For example, many people are firmly convinced that programing language semantics absolutely require the most sophisticated tools of modern mathematics: nonstandard topologies, category theory, linear logic, infinite games, and the like.

Furthermore, current programming languages typically offer a bewildering variety of similar constructs. Loops are a good example: $for$, $while$, $until$ and $repeat$, often with $exit$ and $loop$ back commands, with each language (C,Java, Perl, etc.) offering its own idiosyncratic selection.

Finally, it has become almost the conventional wisdom that nondeterminism (and concurrency) are inherently complex and to be avoided if possible.

I met Dave about the time the first $it$–$ti$ paper appeared, when I arrived at the University of Victoria to join the Computer Science Department where he was already a member. Dave, who likes to practice what he preaches, was teaching a special section of the introductory programming course using the $it$–$ti$ construct in the context of a formal specification-based methodology. Since I would be teaching the course as well, he gave me a copy of the lecture notes, which I proceeded to

examine under a mathematical microscope, using my background in pure mathematics and formal semantics.

My attention was eventually drawn to the restriction on guarded commands of the form $g \rightarrow p$, which required that $p$ terminate in every state for which $g$ was true. The only problem with this otherwise reasonable requirement is that (according to the somewhat informal notes) it had to hold in isolation, i.e., for all values of the program variables, not just those that arise during the execution of the program in which the guarded command appears. For example, if $p$ involves dividing by $n$, then $g$ must test that $n$ is nonzero, even if $n$ has just been assigned the value 2. Strict adherence to the rule as written would force programmers to pad their guards with obviously unnecessary tests.

When I consulted the paper itself, I discovered that the notes were in fact faithful to the official formal semantics. However, many of the example programs in the same paper lacked the required redundant tests. They were not legal programs; they were semantically meaningless, and the formal conclusions drawn about them were null and void.

Dave took the criticism in the constructive spirit in which it was intended, and we worked together to try to understand how much was really lost; the result was the short note that appeared in May 1984 [5].

The note explains the problem very clearly but is somewhat terse in describing the solution. There is no real problem changing the semantics so that the example programs become meaningful (without altering them). The changes, however, complicate the way in which the meaning of an $it$–$ti$ construct depends on the meaning of the parts. The root of the problem, as I would describe it, is that it turns out to be impractical (impossible, if there are infinitely many states) to include LD-union as a language construct, except on an unreasonably restricted class of operands.

Of course, there was always one simple solution, which was to abandon LD-relations and (following other researchers) use only a simple relational approach. However, in the presence of nondeterminism, simple relations have the (in my mind) fatal flaw that they are unable to distinguish between programs that might terminate and programs that must terminate. With LD-relations, this is not an issue, because the competence set specifies those starting states that always lead to termination. Dave deserves credit for not taking the easy way out.

Our short note was not, in fact, the end of the story. Dave and I worked together on the full-length followup paper. We were quite happy with the final result but the referees were not. We both had other priorities and the “lost paper” never saw the light of day. I very much regretted this, because (as you will see) the followup paper does not merely rework the $it$–$ti$ semantics; it also describes Dave’s very clever solution to the (pragmatic) problem of $it$–$ti$ programmers having to duplicate guards. Briefly, he extended the language to allow explicit reference to a value stack, including interpreting the “#” as the top of the stack. At the time this looked very odd (in spite of a simple semantics). Now, however, popular languages like Perl use similar devices that are syntactically and semantically much more complex. In my opinion, Dave’s $it$–$ti$ provides another example of his being well ahead of his time. So perhaps the world is ready for the almost-lost second $it$–$ti$ paper!

2 Introduction

“Structured” control constructs have a very useful property; programs constructed using them can be decomposed into a hierarchy of easily understood parts using simple parsers—without even distinguishing one identifier from another. The semantics of the total program can be determined from the semantics of those parts, using simple set-theoretic operations [4]. Further, the semantics of the program can be determined in a simple order, evaluating inner parts first and constructs at the same level either left to right or right to left as one prefers. In contrast, the use of goto and arbitrary transfers makes it difficult to find a decomposition in which the components have simple semantics.

This property is important because it makes it possible to study a large structured program a small part at a time, and to do so without a previous understanding of the overall structure of that
program. When a program is constructed using labels and unrestricted jumps, some understanding of the program is needed to decompose the program into parts that can be studied in isolation. We consider a construct to be a structured construct if it permits syntactic decomposition as described above.

It is well known that restricting the programmer to the use of standard structured constructs can result in an increase in the space or time requirements of the program. The constructs described in this paper result from a search for structured constructs that do not force such inefficiencies. They are less restrictive than the constructs described in [4, 5].

The meaning of the new constructs is defined using LD-relations [4].

3 The state of a computing machine

Our presentation of these constructs is based on viewing digital computers as finite-state machines in the form introduced by Huffman, Mealy and Moore. It is conventional to view the state as consisting of two components, a data state and a control state. The data state can be represented as a vector in which each element represents the state of one of the program variables. The control state includes any additional information that is needed to determine the future behavior of the machine. As Mills [3] has shown, this conventional distinction is arbitrary and language-dependent. In the sequel, when we talk of the state of a machine, we mean its complete state—all information that determines its future behavior. When we mean the data state or the control state, we will say so explicitly. For the languages in this paper, “data state” means the state of the declared variables, “control state” refers to all other state information.

4 Programs

We view a program as a text describing a set of state-change sequences. A special class of programs consists of “non-procedural” programs. These describe only the relation between initial and final states without describing intermediate states. We can view “non-procedural” programs as program specifications and translators for such programs as automatic programming systems. Following Dijkstra [1], we allow nondeterminism in programs.

Two programs are considered the same if they have the same text. Two programs are said to describe the same algorithm if they use the same representation of the data state and describe the same state-change sequences.

5 Program specifications

When a computing machine is started, the program is represented as part of its initial state. The machine will go through a sequence of state changes. If, for a given starting state, that sequence is finite, we say that the execution of the program terminates or, more simply, that the program terminates. In this paper we shall restrict our attention to situations in which the behavior of programs before they terminate and, consequently, the sequence of state transitions of non-terminating executions, is not of interest. For those starting states in which termination of the execution is guaranteed, we only want to know the possible final states. If the execution might not terminate, we want to know any possible final states and the fact that non-termination is a possibility. Under these conditions the effect of executing a program may be described by an LD-relation on the set of complete states of the machine.

By including the complete state of the machine, LD-relations can even be used to describe self-modifying programs, i.e., programs that modify the portion of the state that we conventionally think of as the program.

Familiarity with LD-relations is a prerequisite for complete understanding of the remainder of this paper. They are defined in Appendix B of this paper.
A set of LD-relations may be issued as a specification of a program. Below are two useful interpretations of a specification that consists of a single LD-relation.

1. A program $p$ satisfies an LD-relation $S$ if the competence set of $S$ is a subset of the competence set of the LD-relation of $p$, and the relation component of the LD-relation of $p$ is a subset of the relation component of $S$.

2. A program $p$ satisfies an LD-relation $S$ if the competence set of $S$ is a subset of the competence set of the LD-relation of $p$, and the relation component of the LD-relation of $p$, restricted to the domain of the relation component of $S$, is a subset of the relation component of $S$.

We use (1) if we wish to require non-termination outside the domain of a specification. Definition (2) makes behavior outside the domain a “do not care” condition.

If the specification consists of a set of LD-relations, a program satisfies that specification if it satisfies any member of the set.

In this paper we write a specification by giving the characteristic predicates of the components of the LD-relations. In describing the characteristic predicate of the relation component, we will denote the value of the program variable $x$ before execution of the program by $x$ and the value of that variable after execution of the program by $x'$. The notation is taken from [2] but the interpretation of the predicates is different.

If the competence set of an LD-relation is identical to the domain of the relation component, we give only the relation component.

6 Primitive programs

With every programming language we are given a set of built-in programs. The built-in programs are considered primitive because we do not examine how they are constructed. Common examples are programs to perform arithmetic operations and programs to assign new values to variables.

7 Control constructs and constructed programs

In addition to the primitive programs, programming languages contain control constructs, which are nor programs themselves but allow the construction of new programs from previously defined programs. We call the resulting programs constructed programs.

8 Defining the semantics of constructed programs

We assume that the LD-relations of each of the primitive programs will be known. The definition of the control constructs defines the LD-relation of the constructed programs as a function of the LD-relation of the programs from which it is constructed.

9 The value of a program

In this paper, a value is associated with every execution of every program. These values are part of the machine state but are not considered part of the data state. Arithmetic expressions are the most familiar example of the use of such values. While evaluating an arithmetic expression may change the explicit state (such a change is often called a side-effect), the evaluation also computes a value. That value may not be recorded in the data state, but it is available for use in the next program to be executed. To define that value we introduce $\#$, a function from the set of machine states to the set of possible program values; $\#(s)$ is the value computed by the program whose execution terminated in $s$. In the languages we describe in this paper, variables and constants...
are treated as primitive programs that do not change the data state; they have their usual values. The value of all constructed programs is determined by the definition of the constructs used to build them.

In the examples in this paper; we allow programs to refer to this value using the symbol “#”. We will also define and use a mechanism for stacking these values.

10 The syntax of the constructs

In this section we give the syntax of the control constructs. Nonterminals are enclosed in angle brackets. Characters enclosed in single quotes are meant to be taken literally, not as meta-characters used for syntax definition.

\[
\begin{align*}
\langle \text{simple program} \rangle & ::= \langle \text{primitive program} \rangle \\
& \quad | \langle \text{program} \rangle \\
& \quad | \langle \text{limited component list} \rangle \\
& \quad | \text{it} \langle \text{program} \rangle \text{ti}
\end{align*}
\]

\[
\langle \text{guard} \rangle ::= \langle \text{simple program} \rangle
\]

\[
\langle \text{limited component} \rangle ::= \langle \text{guard} \rangle \rightarrow \langle \text{simple program} \rangle
\]

\[
\langle \text{limited component list} \rangle ::= \langle \text{limited component} \rangle \\
| \langle \text{limited component list} \rangle \text{'} | \langle \text{limited component} \rangle
\]

\[
\langle \text{composed program} \rangle ::= \langle \text{simple program} \rangle ; \langle \text{simple program} \rangle \\
| \langle \text{composed program} \rangle ; \langle \text{simple program} \rangle
\]

\[
\langle \text{program} \rangle ::= \langle \text{simple program} \rangle \\
| \langle \text{composed program} \rangle
\]

\[
\langle \text{key} \rangle ::= \text{stop} \\
| \text{go}
\]

The nonterminal \(\langle \text{primitive program} \rangle\) is not fully defined by this syntax. We give only examples.

\[
\langle \text{primitive program} \rangle ::= \langle \text{expression} \rangle \mid \langle \text{assignment} \rangle \mid \text{init} \mid \langle \text{key} \rangle \mid \# \mid \{ \mid \cdots
\]

11 Notation

We use \(R_L\) and \(C_L\) to denote the relation and competence set components of an LD-relation \(L\).

If \(p\) is a program, we define:

\(L_p\) as the LD-relation of \(p\),
\(C_p\) as the competence set of \(L_p\), and
\(R_p\) as the relation component of \(L_p\).
12 Guard semantics

Any simple program may be a guard. Guards are used in constructing limited components as discussed below. We call a guard easy if it does not change the data state when its value is “FALSE”. Programming languages in which the guards are restricted to easy guards can be implemented much more efficiently than the general constructs that we will define below. Our definitions are valid for those restricted languages as well.

The reader should note that we allow guard values other than “TRUE” and “FALSE”. All guard values other than “FALSE” will be treated as “TRUE”.

13 The semantics of a limited component

Consider the limited component $g \rightarrow p$, where $g$ is a guard and $p$ is any program. We define the LD-relation $K$ as follows:

$$
C_K = \{ x : \#(x) \text{ is not “FALSE”} \}
$$

$$
R_K = \{ (x, x) : x \text{ is in } C_K \}.
$$

We define $L_{g \rightarrow p}$ as $L_g \cdot K \cdot L_p$.

We define $D_{g \rightarrow p}$ to be:

$$
\{ x : x \text{ is not in } C_g \text{ or there is a } y \text{ such that } (x, y) \text{ is in } R_g, \#(y) \text{ is not “FALSE”, and } y \text{ is not in } C_p \}.
$$

$D_{g \rightarrow p}$ is the set of spaces in which executing $g$, or executing $p$ when $g$ has a value other than “FALSE”, can lead to abortion (inability to terminate). Because no practical implementation can always avoid abortion in such states, they must be excluded from the competence set of constructed programs.

14 The semantics of limited component lists

If the limited component list is a single $\langle$limited component$\rangle$, the semantics are those of the $\langle$limited component$\rangle$.

Consider the component list $A | B$, where $A$ is a limited component and $B$ a limited component list, we define its meaning as follows:

$$
R_{A | B} = R_A \cup R_B
$$

$$
D_{A | B} = D_A \cup D_B
$$

$$
C_{A | B} = (C_A \cup C_B) \cap \neg D_{A | B}.
$$

The competence set of $A | B$ cannot include states that are in either $D_A$ or $D_B$, even if those states are in $C_A$ or $C_B$.

15 The semantics of “;”

$$
L_{A;B} = L_A \cdot L_B.
$$

16 The semantics of “stop”, “go” and “init”

These programs are used to control the behavior of iterative programs, as described in section 17.

Keys are programs that affect hidden portions of the control state. Execution of a key leaves all other aspects of the state, including the value, unchanged. The two keys are “stop” and “go”.
To define the semantics of the keys and the iteration construct, we introduce a partial function on the set of states, \( \$ (s) \). The range of \( \$ \) is \{Go, Stop, Start\}. We define \( L_{Go} \) by:

\[
\begin{align*}
C_{Go} &= \{ x : x \text{ is a state} \} \\
R_{Go} &= \{ (x, x') : x' \text{ is identical to } x \text{ in all respects except,} \\
&\quad \text{possibly, the value of } \$(x') \text{. } \$(x') = Go. \}
\end{align*}
\]

We define \( L_{Stop} \) by:

\[
\begin{align*}
C_{Stop} &= \{ x : x \text{ is a state} \} \\
R_{Stop} &= \{ (x, x') : x' \text{ is identical to } x \text{ in all respects except,} \\
&\quad \text{possibly, the value of } \$(x') \text{. } \$(x') = Stop. \}
\end{align*}
\]

For all primitive programs other than \texttt{Stop} and \texttt{Go}, \( \$ \) in the stopping state is the same as \( \$ \) in the starting state.

We define \texttt{Init} as a primitive program with:

\[
\begin{align*}
C_{Init} &= \{ x : x \text{ is a state} \} \\
R_{Init} &= \{ (x, x') : x' \text{ is identical to } x \text{ in all respects except,} \\
&\quad \text{possibly, the value of } \# \text{. } \#(x') \text{ is true if and only if } \$(x) = \text{Start.} \}
\end{align*}
\]

17 The semantics of the iterative construct (it–ti)

An iterative program consists of the brackets “it” and “ti” surrounding a program, which we call the \textit{body}. The LD-relation of the body can be determined without consideration of the context in which it appears.

\texttt{It B ti} is an iterative program with body \texttt{B}.

The definition below states that if the range of \( R_B \) contains a state \( s \) with \( \$(s) = \text{Start} \), the program may abort.

We will define the \( L_{it \ B \ ti} \) in terms of \( L_B \).

We define LD-relations \( N, G \) and \( S \) (independent of \( L_B \)) as follows:

\[
\begin{align*}
C_N &= \{ x : x \text{ is a state} \} \\
R_N &= \{ (x, x') : x' \text{ is identical to } x \text{ in all respects except,} \\
&\quad \text{possibly, the value of } \$(x') \text{. } \$(x') = \text{Start.} \}
\end{align*}
\]

\[
\begin{align*}
C_G &= \{ x : \$(x) = Go \} \\
R_G &= \{ (x, x) : x \text{ is in } C_G \} \\
C_S &= \{ x : \$(x) = \text{Stop} \} \\
R_S &= \{ (x, x) : x \text{ is in } C_S \}.
\end{align*}
\]

\( N \) is the LD-relation of a program that leaves the data state unchanged but stops in a state \( s \) with \( \$(s) = \text{Start} \). \( G \) is the LD-relation of a program that does nothing if \( \$ \) in the starting state is \( \text{Go} \), but aborts otherwise. \( S \) is the LD-relation of a program that does nothing if \( \$ \) in the starting state is \( \text{Stop} \), but aborts otherwise.

Let the LD-relation \( K \) be the union \( K_0 \cup K_1 \cup K_2 \cup \cdots \), with:

\[
K_0 = S \quad \text{and} \quad K_{i+1} = S \cup G \cdot L_B \cdot K_i.
\]

In words, \( K_i \) describes the possible effect of executing the body at most \( i \) times in states with \( \$ = \text{Go} \) before terminating in a state with \( \$ = \text{Stop} \). \( K \) describes the union of those possibilities.
Let $M$ be $N \cdot L\cdot K$. We define $L_{it}\cdot B\cdot ti$ by:

\begin{align*}
C_{it}\cdot B\cdot ti &= C_M \quad \text{and} \\
R_{it}\cdot B\cdot ti &= \{(x, y') : (x, y) \text{ is in } R_M \text{ and } y' \text{ is identical to } y \text{ except,} \\
&\quad \text{possibly, the value of }\$\text{(y').} \quad \$\text{(x)} = \$\text{(y').}\}
\end{align*}

In words, when the constructed program is executed, $\$\text{ in any stopping state will be the same as }\$\text{ in the starting state.}

The use of stop and go outside of the it–ti brackets has no visible effect except that the value of init will become “FALSE”. The value of init outside of it–ti brackets is not defined.

18 The semantics of parentheses

For any program or limited component list $p$,

$L(p) = L\ p$

Note that if $p$ is a limited component list, its semantics are described by a set $D$ in addition to the LD-relation. However, when we convert $p$ to a program by enclosing it in parentheses, $D$ is no longer of interest.

19 The value of “#”

The effect of a primitive program on # must be defined as part of its semantics. Because # is a function of the state, the value of constructed programs is implied by the definitions above. For the convenience of the reader we provide an informal description here.

- The value of an assignment statement is the value assigned.
- The value of a $(\langle \text{program} \rangle)$ is the value of $(\text{program})$.
- When # appears in a limited component, the value is independent of any other components in the limited component list.
- The value of the limited component $g \rightarrow p$ is determined by $p$ started in the state in which $g$ terminates.
- The value of a limited component list is determined by one of those components whose guard is not “FALSE”.
- The value of $A; B$ is determined by $B$ started in the state in which $A$ terminates.
- The value of an it–ti is the value of the final execution of the body.
- No value is defined if a program aborts.

20 The value stack

In the examples below, “{” denotes a program that saves the value of # on a stack. The value of # is not affected. We denote by “}“ a program that pops the value stack. The value of “}“ is the value removed from the stack. To avoid notational clutter, we may omit “;” before or after these symbols.

- The use of “}“ when the stack is empty is not allowed and the behavior is not defined.
- For any composed program $p$ that does not contain parentheses, let $N(p)$ be the number of occurrences of “{“ minus the number of occurrences of “}“ in $p$. 

For any limited component, let \( N(g \to p) \) be \( N(g) + N(p) \).

For any limited component list \( c_1 \cdots c_n \), let \( N \) be \( \max(N(c_1), \ldots, N(c_n)) \).

If \( p \) is a program, \( N((p)) = N(p) \).

\( N(it\ B\ ti) = N(B) \).

If \( N \) is zero for every \( it\-ti \) constructed program, and if all elements of a limited component list have the same value of \( N \), the stack depth at any point in the program can be computed before execution and efficient code can be compiled.

21 Exits and entrances

One of the surprising implications of the simple syntax and semantics described above is that it allows multiple-entry, multiple-exit programs. Recall that the distinction between program and data and between data state and control state is arbitrary. For example, we could consider the instruction counter as a variable. Alternatively, the value of a program can often be usefully interpreted as a “transfer of control”. Both are changes to the control state. A guard of the form \( \# = x \), where \( x \) is a constant, can be interpreted as a label. A program that begins with a limited component list in which there are three guards of the form just described can be viewed as a three-entrance program. A program with three possible values on termination can be viewed as a three-exit program. Such a multiple-exit program can be connected by “;” to a multiple-entrance program. The semantics specifies exactly the behavior that one would expect. The body of an \( it\-ti \), and the constructed program that results, can also have several entrances and exits. It is clearly within our compiler capabilities to generate machine code that implements such programs using direct transfer from exit to entrance rather than tests and branches. Surprisingly, allowing more than one exit or entrance does not complicate either the syntax or semantics. Neither changes. In some cases, viewing a program as a multi-entrance multi-exit program makes it easier to explain.

22 A very simple example done 3 ways

Some examples are given in the next two sections. These examples include arithmetic expressions. The formal semantics of a class of expressions is given in Appendix A. The primitive program \texttt{entier} returns as its value the largest whole number in its argument.

The following programs add an amount to \( x \) depending on whether the value of \( x \) is even or odd, nonpositive or nonnegative. More formally, we wish to write a program that satisfies the following specification:

\[
\begin{align*}
\text{(odd}(`x`) \land \ `x` \geq 0 \land `x`' = `x` + 1) & \lor \\
\text{(!odd}(`x`) \land `x` \geq 0 \land `x`' = `x` + 2) & \lor \\
\text{(odd}(`x`) \land `x` \leq 0 \land `x`' = `x` + 3) & \lor \\
\text{(!odd}(`x`) \land `x` \leq 0 \land `x`' = `x` + 7)
\end{align*}
\]

1. In this program we use \( \# \) to avoid writing a guard twice, as would be necessary in other languages based on Dijkstra’s guarded commands. The information obtained from the first computation is stored in the variable \( L \) for future use.

\begin{verbatim}
entier(x/2)*2 = x;
( # -> L:='EVEN' | # -> L:='ODD');
x>0;
( # -> (L='EVEN' -> x:=x+2 | L='ODD' -> x:=x+1) \\
| # -> (L='EVEN' -> x:=x+7 | L='ODD' -> x:=x+3)
)
\end{verbatim}
2. Below, the value stack is used instead of the variable \( L \). The program may be understood by looking at the first two lines as a two-exit program, with exits 'EVEN' and 'ODD'. The limited component lists

\[
(\#='\text{EVEN}' \rightarrow x:=x+2 \mid \#='\text{ODD}' \rightarrow x:=x+1)
\]

and

\[
(\#='\text{EVEN}' \rightarrow x:=x+7 \mid \#='\text{ODD}' \rightarrow x:=x+3)
\]

may each be viewed as two-entrance programs with entrance labels 'EVEN' and 'ODD'. The use of the brackets makes it possible to combine these programs in a limited component list and produce a two-entrance program with the same entrances.

\[
\text{entier}(x/2)*2 = x;
\]

\( (\# \rightarrow \text{'}EVEN\text{'} | \neg\# \rightarrow \text{'}ODD\text{'});\)

\( (\{ x>0;\)

\( (\# \rightarrow (\{ (\#='\text{EVEN}' \rightarrow x:=x+2 \mid \#='\text{ODD}' \rightarrow x:=x+1))

\( \mid \neg\# \rightarrow (\{ (\#='\text{EVEN}' \rightarrow x:=x+7 \mid \#='\text{ODD}' \rightarrow x:=x+3))
\)

3. In this version, 'EVEN' and 'ODD' are not used. The first line constitutes a two-exit program with values 'TRUE' and 'FALSE'. The other limited component lists are two-entrance programs with labels 'TRUE' and 'FALSE'.

\[
\text{entier}(x/2)*2 = x;
\]

\( (\# \rightarrow \text{'}TRUE\text{'} | \neg\# \rightarrow \text{'}FALSE\text{'});\)

\( (\{ x>0;\)

\( (\# \rightarrow (\{ (\# \rightarrow x:=x+2 \mid \neg\# \rightarrow x:=x+1))

\( \mid \neg\# \rightarrow (\{ (\# \rightarrow x:=x+7 \mid \neg\# \rightarrow x:=x+3))
\)

23 The deed problem

We are to write a program that will output DEED if INPUT contains two D’s and two E’s, NO DEED otherwise. To specify the program we assume a predicate “hasit” which is ‘TRUE’ if the input stream contains two D’s and two E’s. The symbol “||” indicates concatenation of strings. The problem is to write a program whose LD-relation is characterized by:

\[
\text{hasit} \land \text{OUTPUT} = \text{OUTPUT} || \text{’DEED’} \lor
\]

\[
\neg\text{hasit} \land \text{OUTPUT} = \text{OUTPUT} || \text{’NO DEED’}
\]

This problem was used in [4] to illustrate a restricted version of these constructs. Here we show a program that evaluates expressions less often than the best solution given there. In this example, “next(input)” is a primitive program that sets \# to the next character in the input stream. The value of input is ‘EMPTY’ if and only if there are no more data to be read. The body of the it-ti can be understood as consisting of two sections. The first has two exits. ‘FOUND’ is taken if we have found the required D’s and E’s, ‘ONWARD’ is taken otherwise. The second portion of the body has two entrances. It produces the requisite output and determines whether or not the iteration should continue.
( d:=0;
 e:=0
 );
 it
 init;
 ( # -> 'ONWARD'
 | ¬# -> ( data = 'D';
 ( # -> d:=d+1
 | ¬# -> ( data = 'E';
 ( # -> e:=e+1
 | ¬# -> 'FALSE'
 ) )
 )
 )
 ( # -> ( (d<2) \ (e<2);
 ( # -> 'ONWARD'
 | ¬# -> 'FOUND'
 )
 )
 | ¬# -> 'ONWARD'
 )
 )
 ( # = 'ONWARD' -> ( input = 'EMPTY';
 ( ¬# -> ( data := next(input);
 go
 )
 | # -> ( output := output || 'NODEED;
 stop
 )
 )
 )
 | # = 'FOUND' -> ( output := output || 'DEED';
 stop
 )
 )

24 Conclusions

There has been a long-standing and widespread assumption that improving the structure of programs costs time and space. We believe that to be false. Previous conclusions have been based on a syntactic definition of “structured”.

This paper’s definition is based on a practical need, the ability to decompose a program into comprehensible parts without first understanding how it is intended to work. We have found that we can satisfy this constraint and still allow side-effects in guards and programs with more than one entrance or exit without complicating our semantics. Side effects in ‘FALSE’ guards are expensive in terms of run-time computer resources, but the semantics are clear and simple. Side effects in guards when the guard evaluates to ‘TRUE’ are useful and cause no problems at all.

In part, the earlier opinions were based on an unjustified distinction between program and data. Mills [3] has shown how arbitrary such a distinction is. The interchangeability of program and data can be used to reduce the complexity of a language without introducing restrictions.
Acknowledgements

J.W.J. Williams stimulated this research by pointing out some restrictions in the constructs proposed in [4]. Paul Clements, Stuart Faust, Bruce Labaw, and John McLean have provided many helpful comments on earlier versions of this paper. Harlan Mills has influenced this work in many ways and has made helpful comments on earlier versions of this paper.

References


A Semantics of a class of expressions

For the purposes of the examples in this paper, we define the meaning of a class of expressions. These expressions are built up from constants, variables, and #, using unary and binary operators.

A constant denotes a program that always terminates, and whose only effect is to return the indicated value. Thus, $C_{\text{string}}$ is $\{x : x \text{ is a state}\}$ and $R_{\text{string}}$ is

$$\{(x, x') : x \text{ is a state and } x' \text{ is identical to } x \text{ in all respects except, possibly, the value of } #. \ #(x') = \text{'}\text{string}'\}.\$$

A variable denotes a program that always terminates, and whose only effect is to return the indicated value. Thus (for example) $C_{\text{count}}$ is $\{x : x \text{ is a state}\}$ and $R_{\text{count}}$ is

$$\{(x, x') : x \text{ is a state and } x' \text{ is identical to } x \text{ except, possibly, the value of } #. \ #(x') = \text{the value of count in } x.\}$$

The meaning of # is the identity LD-relation, defined by

$$C_{#} = \{x : x \text{ is a state}\}$$

$$R_{#} = \{(x, x) : x \text{ is a state}\}.$$ 

Let $A$ be a program and $\circ$ be a unary operation. Then $\circ A$ is a program like $A$, except that the value returned by $\circ A$ is the result of applying $\circ$ to the value returned by $A$. More precisely, $C_{\circ A}$ is

$$\{x \text{ in } C_A : \text{if } (x, y) \text{ is in } R_A \text{ then } #(y) \text{ is in the domain of } \circ\}$$

and $R_{\circ A}$ is

$$\{(x, y') : \text{for some } y, (x, y) \text{ is in } R_A, \ #(y) \text{ is in the domain of } \circ, \text{ and } y' \text{ is identical to } y \text{ except, possibly, the value of } #(y'). \ #(y') = \circ #(y)\}.\$$

12
Finally, let $A$ and $B$ be any two programs and let $\Delta$ be a binary operator. $R_{A \Delta B}$ is

$$\{(x, z') : \text{for some } y \text{ and } z, (x, y) \text{ is in } R_A, (y, z) \text{ is in } R_B, (\#(y), \#(z)) \text{ is in the domain of } \Delta, \text{ and } z' \text{ is identical to } z \text{ except, possibly, } \#(z'). \}$$

The competence set $C_{A \Delta B}$ is

$$\{x \in C_A; B : \text{for all } y, z \text{ such that } (x, y) \text{ in } R_A \text{ and } (y, z) \text{ in } R_B, (\#(y), \#(z)) \text{ is in the domain of } \Delta \}.$$ 

For any expression $X$, $L(X) = L_X$. 

The meaning of $A_1 \Delta A_2 \Delta \ldots A_n$ is the meaning of $A_1 \Delta (A_2 \Delta (\ldots \Delta A_n) \ldots)$

These definitions do not allow one to mix operators without the use of parentheses to indicate operator precedence.

One must be very careful about using $\#$ in expressions. For example, $\#>4$ and $4<\#$ are not the same. The latter will never be ‘TRUE’. $(2+3);\#=5$ is a constructed program with value ‘TRUE’. If $x$ and $y$ are variables, $(x+y);\#=5$ always has value ‘TRUE’. $(x+y);\#=5$ has value ‘TRUE’ only if $x+y$ has value 5.

## B Limited-domain relations

The following definitions are used in defining the control constructs. The first four are standard and are included for completeness. This appendix is an extract from [4].

### Universe

In the following, we assume the existence of a set known as the universe, $U$. All elements discussed below are members of $U$. We also assume that the concept of an ordered pair of elements from $U$ and the usual set-theoretic concepts are understood.

### Relation

A relation $R$ is a set of ordered pairs; both elements of the pair are members of $U$. We indicate that a pair $(x, y)$ is in $R$ by the notation, $(x, y) \in R$.

### Domain

The domain of a relation $R$, denoted $\text{dom}_R$, is defined by:

$$\text{dom}_R = \{x : \text{there is a } y \in U \text{ such that } (x, y) \in R\}.$$ 

### Limited-domain relation (LD-relation)

A limited-domain relation $L$ is an ordered pair $(R_L, C_L)$, where $R_L$ is a relation and $C_L$ is a subset of $\text{dom}_R$. We call $C_L$ the competence set of $L$. For any two LD-relations $A$ and $B$,

$$A = B \iff R_A = R_B \text{ and } C_A = C_B$$

$$A \subseteq B \iff R_A \subseteq R_B \text{ and } C_A \subseteq C_B.$$ 

### The union of two LD-relations

For any two LD-relations $A$ and $B$, we define $A \cup B$ by:

$$R_{A \cup B} = R_A \cup R_B$$

$$C_{A \cup B} = C_A \cup C_B.$$
Composition of LD-relations

Let $A$ and $B$ be LD-relations. $A \cdot B$ is defined by:

$$R_{A \cdot B} = \{(x, y) : \text{there exists a } z \text{ such that } (x, z) \in R_A \text{ and } (z, y) \in R_B\}.$$ 

In other words, $R_{A \cdot B} = R_A \cdot R_B$.

$$C_{A \cdot B} = \{x : x \in C_A \text{ and for all } y \text{ such that } (x, y) \in R_A, y \in C_B\}.$$ 

Readers should note that the convention for relational composition is not consistent with the convention often used for functional composition. The order is reversed!

Some theorems about LD-relations

Let $A$, $B$ and $L$ be LD-relations,

$$A \cup (B \cup C) = (A \cup B) \cup C$$

$$A \cup B = B \cup A$$

$$C \cdot (A \cup B) \supseteq C \cdot A \cup C \cdot B$$

$$(C \cdot A) \cdot B = C \cdot (A \cdot B)$$

LD-relations and programs

When we use LD-relations to describe programs, $U$ is the set of machine states. The relation component of an LD-relation describing a program is the set of states $(x, y)$ such that when the program is executed starting in state $x$ it may terminate in state $y$. The competence set of that LD-relation is the set of states in which the program described is guaranteed to terminate.