Formal Aspects of Computing (1991) 3: 58–101

Formal Aspects of Computing

A Demonstrably Correct Compiler

Susan Stepney, Dave Whitley, David Cooper and Colin Grant
Logica Cambridge Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK

Keywords: Correct compiler; Formal specification; Denotational semantics; Prolog; DCTG

Abstract. As critical applications grow in size and complexity, high level languages, rather than better-trusted assembly languages, will be used in their development. This adds potential for extra errors to creep in, especially in the now necessary compiler. To avoid these new errors, it is necessary to have a formal specification of the high level language, and a formal development of its compiler. We outline what we believe is a practical route for achieving a demonstrably correct compiler, and describe a prototype compiler we have built by this route for a small, but non-trivial, language.

1. Introduction

It has been argued that the only "safe" way to write critical applications is by using assembly language, because this is the only way to be sure about what will happen during program execution. Languages further removed from the hardware cannot be trusted for two reasons:

- It is impossible to know what the language means; to know how it translates to the "real" machine.
- Even given some idea of the meaning of the high level constructs, it is impossible to know that the compiler correctly implements this meaning.

There is a grain of truth in this argument, but as the applications grow larger and more complex, using assembly language becomes infeasible; high level languages, with all their software engineering advantages, will become essential. How can these conflicting requirements be reconciled? As a first step along the way, (at least) the following need to be satisfied:

Correspondence and offprint requests to: Susan Stepney, Logica Cambridge Ltd, Betjeman House, 104 Hills Road, Cambridge CB2 1LQ, UK.

- It must be possible to deduce the logical behaviour of a particular program independent of its execution on a particular target.
- The high level language must have a *formally defined semantics.* Otherwise it is impossible to deduce even what *should* be the effect of executing a particular program.
- The formal semantics must be established and made available for peer review and criticism.
- The compiler must be correct, hence it must be derived directly from the formal semantics.
- The compiler for a critical language must be *seen to be correct.* Hence it must be written legibly, and must be easily related to the formal semantics.
- The code produced by the compiler must be clear, and easily related to the source code. This gives the required *visibility* to the compilation process for a critical language.

The last two points are important for critical applications, in order to conform with the much more stringent validation requirements these have.

In addition to the above requirements, the equally thorny problems of showing that the *application* is correct, and of showing that the hardware correctly implements the meaning of the machine language, must be addressed. These are beyond the scope of this paper.

In this paper, we will describe how a compiler can be constructed from the formal definition of a language. This compiler has the property of being *correct by construction,* and hence demonstrably correct. We will do this by defining a semantics for a small (but by no means trivial) high level language, (which for the purposes of this paper we will call Tosca – "not a *To*y language, for *S*afety *C*ritical *A*pplications"), then constructing a compiler from them. Note that we are not proposing a new language, but are rather demonstrating how the formal semantics of a given language can be used in trusted compiler development.

Earlier work on compiler correctness includes [McP66, MiW72, Mor73, Coh79, Pol81]. Work on generating a compiler automatically from a denotational semantics definition of the language includes [Mos75, Pau81, Pau82, Wan84, Lee89] and more recent work on semantics-directed compiler generation includes [HoJ90].

2. Semantics

In order to write a correct compiler, it is necessary to have a formally defined semantics of the language. There are several ways of defining the semantics of programming languages, each appropriate for different purposes:

An *axiomatic semantics* defines a language by providing axioms and rules of inference for reasoning about programs, for example:

$$\textbf{skip};\ \langle stmt \rangle = \langle stmt \rangle = \langle stmt \rangle;\ \textbf{skip} \qquad (1)$$

It is appropriate for showing that two programs have the same meaning (useful, for example, when doing program transformations for the purpose of optimisations), but is rather too abstract for defining a compiler.

An *operational semantics* defines a language in terms of the operation of a (possibly abstract) machine running programs, and so is mostly concerned with implementations. It is too concrete for a machine-independent definition of a language (although such a definition will become necessary for the back-end compiler development, see later).

A *denotational semantics* defines a language by assigning a mathematical value – "meaning" – to each language construct, hence allowing the calculation of abstract machine-independent meanings of programs [Sto77]. It is at the right level of abstraction for building a compiler.

2.1. Denotational Semantics

We will use denotational semantics to specify our example language, Tosca. Its modelling of abstract meanings of programs, independent of any machine implementation, satisfies the requirement that a Tosca program must have the same logical behaviour no matter which hardware is used to run it. For a good introduction to denotational semantics, see, for example [All86] or [Gor79].

2.2. Non-Standard Semantics

Using denotational semantics has another important advantage. Certain "non-standard" interpretations of the semantics can be made, which allow various analyses of a program to be carried out. The best known of these is the use of static semantics for type checking. Various other semantic analyses can similarly be carried out [CoC77], of which many traditional analyses (for example, [BeC85, Bra84]) are a subset. Such analyses can be added as a further component of the compiler, all within the same consistent denotational semantics paradigm.

2.3. Size of Task

A denotational semantics of Modula-2, written in VDM (or more accurately, written in the functional subset of Meta-IV, essentially a programming language), is about 200 pages long [Andnd]. Modula-2 had to be specified retrospectively. A new language purpose-designed for critical applications (designed either from scratch, or by carefully subsetting an existing language) would probably be of a similar size to Modula-2 in terms of syntax, but could be much simpler semantically: since it would be designed using denotational semantics, features which are difficult or "messy" to specify could be left out. Indeed, it can be argued that if a language feature is difficult to specify cleanly, it is difficult to understand, and hence should not be included in a language to be used for critical applications [Car89]. Note that the converse does not apply: that a particular feature is easy to specify is not sufficient reason for including it in the language.

3. From Semantics to a Compiler

The denotational semantics provides the formal specification of the source language. For the development of a compiler for a particular target machine, the semantics of the target machine language is also required. The compiler's job is to translate each high level construct, such as

if ⟨*test_expr*⟩ **then** ⟨*then_cmd*⟩ **else** ⟨*else_cmd*⟩ (2)

into a corresponding target language template, such as

```
  ⟨test code, label1⟩
  ⟨then code⟩
  JUMP label2
label1:
  ⟨else code⟩
label2:
```

The program fragments in angle brackets may be similarly translated, recursively. The above template defines the *operational semantics* of the **if_then_else** statement in the target language.

An obvious question arises: how can one have any confidence that this is the correct target language template? In order to answer this question, and hence to write a correct compiler for a particular target, it is necessary also to have a formal semantics of the target machine language. There is little point in being rigorous about what a high level language program means if one cannot be similarly rigorous about what the ostensibly equivalent target language program means! Given such a semantics, it is possible to *calculate* the meaning of the template in the target language. This can be compared with the meaning of the corresponding high level fragment, and shown to be the same (see Appendix A for an example calculation).

The formalism provides a *structuring* mechanism for the proof process. Arguments are advanced on a *node by node basis,* using structural induction. Each node has an operational semantics in the form of a target language template. Using the denotational semantics of the target language, the meaning of the template can be calculated, and shown to be the same as the meaning given by the denotational semantics for that node. The complete proof is constructed by working through the tree of the language, until all nodes have a suitable argument supporting them; this then completes the argument in support of the compiler as a whole. Hence the complete proof is composed, using a divide-and-conquer strategy, from a number of smaller, independent subproofs. This structure makes the total proof much more tractable, and more understandable, than would a single monolithic approach.

4. Executable Specification Language

There are various notations available for writing denotational semantics, including the conventional mathematical notation. In particular, an executable specification language can be used. This has the advantage that executing the denotational semantics of a language immediately provides an *interpreter* for that language [All86]. This interpreter can, if desired, be used as a *validation tool* for checking that the formal specification of the language satisfies any

informal requirements there may be (see [StL87] for an example of executing a specification in order to validate it).

It is possible, given a denotational semantics in some abstract notation, to translate it into an imperative language, such as Pascal [All86], in order to produce an interpreter. However, such a process is in itself potentially error-prone, does not necessarily produce a transparently correct interpreter, and the correspondence between it and the operational semantics needed for the associated *compiler* are not obvious.

Another approach is to write the denotational semantics directly in an *executable specification language,* obviating the need for a translation step. In order to provide a transparently correct interpreter, the chosen specification language would need to be of sufficiently high level, enabling the semantics to be written clearly and abstractly, unlike the case if written directly in, say, Pascal. The specification language itself would also need to have a formal semantics, so that the correspondence between the denotational and operational semantics required to produce a compiler could be reasoned about.

Alternatively, if the abstract notation form is translated into a high-enough level language, the interpreter may be transparently correct.

In either case, the requirement is that the executable specification language is very high level – much higher than a conventional imperative language. A functional language or logic language seems a natural choice. Although there are no obvious technical advantages of logic languages over functional languages from the point of view of implementing the compiler, the picture changes when the development environment is considered.

We have chosen to use Poplog Prolog as the executable specification language for the Tosca compiler. Prolog is a mature language that is well supported and has a large user community. Provided that some of its trickier features are handled in a disciplined manner (for example, using only "green" cuts and no "red" cuts [StS86]), Prolog programs can be written in the clear manner required for the Tosca compiler.

5. Prolog Example

5.1. Denotational Semantics

Before launching into a definition of the semantics of Tosca, let us look at how to write denotational semantics in Prolog. The following language is trivial enough that a simple *State* [All86, chapter 5] is sufficient, rather than needing to separate the *Environment* and *Store.* In this language the semantics of the **if_then_else** and **while_do** commands, in conventional notation, look something like:

$$\mathit{State}: \mathit{Identifier} \to \mathit{Int} \tag{3}$$

$$\mathbf{C}: \mathit{Cmd} \to \mathit{State} \to \mathit{State} \tag{4}$$

$$\mathbf{E}: \mathit{Expr} \to \mathit{State} \to \mathit{Int} \tag{5}$$

$$\mathbf{C}[\![\textbf{if } \epsilon \textbf{ then } \gamma_1 \textbf{ else } \gamma_2]\!]\sigma = (\text{if } \mathbf{E}[\![\epsilon]\!]\sigma = 1 \text{ then } \mathbf{C}[\![\gamma_1]\!] \text{ else } \mathbf{C}[\![\gamma_2]\!])\sigma \tag{6}$$

$$\mathbf{C}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]\sigma = (\text{if } \mathbf{E}[\![\epsilon]\!]\sigma = 1 \text{ then } \mathbf{C}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!] \circ \mathbf{C}[\![\gamma]\!] \text{ else } \mathit{Identity})\sigma \tag{7}$$

(In this example, there are only integers, so *True* is represented by the integer 1). These could be written instead in Prolog as something like:

```
command(if(Test, Then, Else),
   PreState, PostState):-
  expr(Test, PreState, Value),
  ( Value=bTRUE,
    command(Then, PreState, PostState)
  ;
    Value=bFALSE,
    command(Else, PreState, PostState)
  ).
command(while(Test, Body), PreState, PostState):-
  expr(Test, PreState, Value),
  ( Value=bTRUE,
    command(Body, PreState, MidState),
    command(while(Test, Body), MidState, PostState)
  ;
    Value=bFALSE,
    PostState=PreState
  ).
```

In the usual Prolog schizophrenic manner, this can be read either declaratively, giving the semantics, or operationally, giving the interpreter.

5.2. Code Templates

Consider the process of translating a statement such as **if** ⟨**test**⟩ **then** ⟨**then**⟩ **else** ⟨**else**⟩ into a hypothetical machine code such as

```
  ⟨test code, label1⟩
  ⟨then code⟩
  JUMP label2
label1:
  ⟨else code⟩
label2:
```

[War80] shows one direct translation into Prolog, which gives explicit templates for each language construct:

```
encodestatement(if(Test, Then, Else), Dictionary,
      (Testcode;
       Thencode;
       instr(jump, L2);
     label(L1);
       Elsecode;
     label(L2) )
):-
  encodetest(Test, Dictionary, L1, Testcode),
  encodestatement(Then, Dictionary, Thencode),
  encodestatement(Else, Dictionary, Elsecode).
```

Although this sort of Prolog is quite clear for such a small example, it soon becomes unwieldy, and a more powerful structuring mechanism is needed for a full language with multiple semantics.

6. Definite Clause Translation Grammars

Standard Prologs have a grammer mechanism called Definite Clause Grammars (DCGs) built into them, to allow shorthand expressions such as

```
sentence→noun_phrase, verb_phrase.
noun_phrase→determiner, noun.
```

to be manipulated (see, for example [ClM87, chapter 9]). These are *automatically* converted into their standard Prolog equivalents

```
sentence(S0, S) :-noun_phrase(S0, S1),
                  verb_phrase(S1, S).
noun_phrase(S0, S) :-
  determiner(S0, S1), noun(S1, S).
```

by the Prolog system itself. This approach is suitable for defining syntax. For defining semantics as well, there is a more powerful approach, called Definite Clause Translation Grammars (DCTGs) [AbD89, chapter 9]. These provide a general mechanism for grammar computations; a parse tree is formed as the result of a successful derivation, and semantics rules can be attached to non-terminal nodes in the parse tree. These rules give the semantic properties of a node in terms of the semantic properties of its subtrees, and is the logic formalism equivalent of Attribute Grammars [Knu68]. The DCTG formalism does not distinguish between inherited and synthesized attributes, however, since Prolog's unification mechanism makes this largely unnecessary. Although not directly supported by the Prolog system in the same way as DCGs, Prolog operators and clauses can be defined to support the DCTG approach.

As a simple example, consider a possible DCTG definition for adding two expressions to produce an expression:

```
expr ::= expr^^Tree1, tPLUS, expr^^Tree2
⟨:⟩
value(V) ::-
   Tree1^^value(V1), Tree2^^value(V2), V is V1+V2.
```

The first part of the term (before the ⟨:⟩) defines the syntax. In this example it says that an expression can be an expression followed by a token followed by another expression. The subexpressions are labelled with their derivation trees. The second part of the term defines the semantics of the composite expression in terms of its subexpressions. Here it says the value of the composite expression is the arithmetic sum of the values of the two subexpressions.

At first sight, this technique may look more complicated than using straightforward Prolog. It does, however, have the advantage of cleanly separating the syntax and semantics. Another important advantage of this approach is that a DCTG can be used to support multiple sets of *different* semantics attached to each node. So the **if_then_else** example could be

written using the DCTG formalism, and including both the code template and the denotational semantics, as:

```
command ::=
  tIF, test^^B,
  tTHEN, command^^C1,
  tELSE, command^^C2
⟨:⟩
  (code(Dictionary, [TestCode, ThenCode, jump(L2),
     label(L1), ElseCode, label(L2)])
  ::-
    B^^code(Dictionary, L1, Testcode),
    C1^^code(Dictionary, Thencode),
    C2^^code(Dictionary, Elsecode)
  ),
  (meaning(PreState, PostState) ::-
    B^^meaning(PreState, Value),
    ( Value=bTRUE,
      C1^^meaning(PreState, PostState)
    ;
      Value=bFALSE,
      C2^^meaning(PreState, PostState)
    )
  ).
```

Non-standard semantics can be attached to the DCTG nodes in the same manner; each node would also hold at least the static semantics used for type checking. If required, other non-standard semantics for various other types of analyses can be incrementally added to each node in a similar, consistent manner.

Notice how, with the DCTG approach, the denotational semantics and the operational semantics (code templates) are in similar forms, and occur textually close together in the specification. This is of great advantage in the process of demonstrating the correctness of the compiler.

7. Tosca

Tosca is large enough to be non-trivial, but small enough to fit the confines of this paper and (hopefully) to be understable without too much effort on the part of the reader. It is block structured, allowing local declarations. It has two types, integer and boolean, but only integers can be declared; booleans are restricted to controlling loops and choices. It has a **while_do** loop and an **if_then_else** choice. There is a simple procedure declaration, but it has no parameters. Other loops (for example, **repeat_until** or **for** loops) and choices (for example, **case**) would merely add bulk, not new insight, to the discussion.

The following sections define Tosca's syntax, and three semantics: the dynamic (execution) semantics and two static semantics, type checking and use checking.

A program must be syntactically correct before any of the semantics are defined. If a program does not type check – if $\mathbf{TC}[\![\gamma]\!]\tau_{init} = \mathit{TypeWrong}$ – then

the use checking semantics and dynamic semantics are *undefined.* Similarly, if the program does not use check, then the dynamic semantics is undefined. This means that there is no need to check, for example, that expressions are of the right type, or that variables have been initialised before they are used, in the dynamic semantics. For the purposes of exposition, these static semantics are described after the dynamic semantics in this paper, although the corresponding checks would have to be applied earlier to a program being compiled. This ability to separate concerns simplifies each of the individual semantics, since "error cases" do not have to be considered each time.

The DCTG form of the following definitions is given in Appendix B.

8. Tosca's Syntax

Tosca's syntax is defined below using a BNF-like notation. Language keywords are shown using a sans-serif font (**like this**), and general program constructs are given using the following Greek characters:

Binary operator Ω
Command γ
Command list Γ
Constant χ
Declaration δ
Expression ϵ
Identifier ξ
Unary operator Ψ

Literal constants and identifiers are not further defined here.

8.1. Declarations

A declaration is either a variable (an integer) or a procedure. The syntax for declarations is:

$$Decl : \delta ::= \textbf{var}\ \xi \tag{8}$$
$$\mid\ \textbf{proc}\ \xi = \gamma \tag{9}$$
$$\mid\ \delta\ ;\ \delta \tag{10}$$

8.2. Operators

8.2.1. Binary Operators

Expressions can be combined by binary arithmetic operators ($+, -$, etc.), by binary comparison operators ($<, =$, etc.) and by binary logical operators (**or**, **and**):

$$BinArithOp : \Omega_\alpha ::= + \mid - \mid \cdots \tag{11}$$
$$BinCompOp : \Omega_\chi ::= < \mid > \mid = \mid \cdots \tag{12}$$
$$BinLogicOp : \Omega_\lambda ::= \textbf{or} \mid \textbf{and} \tag{13}$$
$$BinOp = BinArithOp + BinCompOp + BinLogicOp \tag{14}$$

The disjoint union operator (+) used above to define *BinOp*, and elsewhere, tags its elements in some way, to distinguish which set they originally came from. This has the consequence that if $x \in X$, then $x \notin X + Y$. A mapping function is needed to extract the original elements from the disjoint union. However, we shall freely abuse the notation and write $x \in X + Y$, since it does not lead to confusion in this application.

8.2.2. Unary Operators

Expressions can be built from the unary arithmetic operator ($-$) and the unary boolean operator (**not**).

$$UnyArithOp : \Psi_\alpha ::= - \tag{15}$$

$$UnyLogOp : \Psi_\lambda ::= \textbf{not} \tag{16}$$

$$UnyOp = UnyArithOp + UnyLogOp \tag{17}$$

8.3. Expressions

The simplest expressions are boolean and integer constants (**true**, **false** and numbers), and identifiers. Expressions can be combined by binary and unary operators. Rather than introduce precedence, binary expressions are parenthesised. The syntax for expressions is:

$$Expr : \epsilon ::= \chi \tag{18}$$

$$\mid \xi \tag{19}$$

$$\mid (\epsilon \Omega \epsilon) \tag{20}$$

$$\mid \Psi \epsilon \tag{21}$$

8.4. Commands

The commands in Tosca are: forming a **begin_end** block for local declarations and/or multiple commands, **skip**, assignment, **if_then_else**, **while_do**, procedure call, **input**, **output** and sequential combination. The syntax for commands is:

$$Cmd : \gamma ::= \textbf{begin } \delta ;; \Gamma \textbf{ end} \tag{22}$$

$$\mid \textbf{begin } \Gamma \textbf{ end} \tag{23}$$

$$\mid \textbf{skip} \tag{24}$$

$$\mid \xi := \epsilon \tag{25}$$

$$\mid \textbf{while } \epsilon \textbf{ do } \gamma \tag{26}$$

$$\mid \textbf{if } \epsilon \textbf{ then } \gamma \textbf{ else } \gamma \tag{27}$$

$$\mid \xi \tag{28}$$

$$\mid \textbf{input } \xi \tag{29}$$

$$\mid \textbf{output } \epsilon \tag{30}$$

The syntax for multiple commands is:

$$CmdList : \Gamma ::= \gamma \tag{31}$$

$$\mid \quad \gamma ; \Gamma \tag{32}$$

9. Tosca's Dynamic Semantics

The dynamic semantics describes the meaning of executing a program. Remember that this semantics is defined only if the program type checks and use checks (defined later).

9.1. A Note on Notation

Defining the denotational semantics of the language consists of defining various valuation (meaning) functions. Each function maps a syntactic program construct onto a well-defined mathematical object, that construct's denotation. Conventionally, double square denotation brackets, $[\![\ ]\!]$, are used to enclose these syntactic program structures.

Confusion can occasionally arise because of the notation used. For example, if a minus sign appears inside the double square brackets, it is merely syntax; it is the job of the semantic equations to define what it *means*. On the other hand, if it appears outside the brackets, it is the well-known mathematical operator, with its well-known meaning(s). Similarly, a digit sequence appearing inside the double square brackets is merely syntax (for example, the numeral consisting of the character "4" followed by the character "2") whereas one outside is the corresponding mathematical *number* (forty-two).

This potential for confusion occurs for any of the operators defined below that "look the same" inside and outside the brackets; it should be borne in mind that they are different things. In order to minimise confusion, a sans-serif font (**like this**) is used for Tosca syntax, a serif font (like this) for mathematics and explanatory text, and a typewriter font (`like this`) for target language syntax (see later). In principle, it would be possible to use distinct symbols all the time, as is done for **and** and $\wedge$, for example, but this would just cause different confusion, not less.

9.2. The Domains

In Tosca, expressible values are either integers or booleans. So *Value* is defined to be the disjoint union of *Int* and *Bool*:

$$Int = \text{the domain of integers} \tag{33}$$

$$Bool = \text{the domain of booleans} - True, False : Bool \tag{34}$$

$$Value = Int + Bool \tag{35}$$

The *Store* is the mapping from store locations to the value stored there (a store location could be represented by integers – memory addresses – for example, but this will not be discussed further here). Only integers are storable values:

$$\sigma : Store = Locn \rightarrow Int \tag{36}$$

Input and output are lists of integers:

$$\iota : Input = nil + Int \times Input \tag{37}$$

$$o : Output = nil + Int \times Output \tag{38}$$

The state of a computation is given by the store, the input and the output:

$$\Sigma : State = Store \times Input \times Output \tag{39}$$

The environment describes the mapping from identifiers to what they denote. For a variable, it is the relevant store location, which will map to a value that can change as the computation progresses. For a procedure declaration, it is the computation denoted by the procedure body, which does not change.

$$\rho : Env = Identifier \rightarrow Locn + Proc \tag{40}$$

A *Proc* is a mapping from one state to another that results from executing the procedure body:

$$Proc = State \rightarrow State \tag{41}$$

Of use later is the identity function, *Identity*, which leaves the state unchanged:

$$Identity : State \rightarrow State \tag{42}$$

$$Identity\ \Sigma = \Sigma \tag{43}$$

9.3. Types of the Meaning Functions

Binary operator **B**: $BinOp \rightarrow Value \times Value \rightarrow Value$ (44)

Command **C**: $Cmd \rightarrow Env \rightarrow State \rightarrow State$ (45)

Command list **CL**: $CmdList \rightarrow Env \rightarrow State \rightarrow State$ (46)

Declaration **D**: $Decl \rightarrow Env \rightarrow Env$ (47)

Expression **E**: $Expr \rightarrow Env \rightarrow State \rightarrow Value$ (48)

Unary operator **O**: $UnyOp \rightarrow Value \rightarrow Value$ (49)

9.4. Declarations

The meaning function for declarations, **D**, takes a declaration and environment, and gives a new environment containing the declaration.

9.4.1. Variable Declaration

The declared variable is added to the environment by mapping it to a previously unallocated location:

$$\mathbf{D}[\![\mathbf{var}\ \xi]\!]\rho = \rho \oplus \{\xi \mapsto loc\} \tag{50}$$

where

$$loc : Locn \notin \operatorname{ran} \rho$$

Note that this declaration does not change the store, hence *loc* is not in the domain of the store, and so $\sigma(\rho[\![\xi]\!]) = \perp$.

9.4.2. Procedure Declaration

The declared procedure is added to the environment by mapping its name to the computation denoted by its body, which computation is performed in the declaration's environment:

$$\mathbf{D}[\![\textbf{proc}\ \xi = \gamma]\!]\rho = \rho \oplus \{\xi \mapsto \mathbf{C}[\![\gamma]\!]\rho\} \tag{51}$$

Hence Tosca's procedures are not recursive. Such requirements are often laid on safety-critical languages, in order to prevent programs overflowing the available memory during execution.

9.4.3. Multiple Declarations

Sequential combination of declarations means apply the second declaration to the result of applying the first declaration.

$$\mathbf{D}[\![\delta_1 ; \delta_2]\!] = \mathbf{D}[\![\delta]\!] \circ \mathbf{D}[\![\delta_1]\!] \tag{52}$$

9.5. Operators

The meaning functions **B** and **O** take binary and unary operators and map them to the corresponding mathematical operators (which can be written in their conventional infix form):

$$\mathbf{B}[\![+]\!](x, y) = x + y \tag{53}$$

$$\mathbf{B}[\![<]\!](x, y) = x < y \tag{54}$$

$$\mathbf{B}[\![\textbf{and}]\!](x, y) = x \wedge y \tag{55}$$

etc.

$$\mathbf{O}[\![-]\!]x = -x \tag{56}$$

$$\mathbf{O}[\![\textbf{not}]\!]x = \neg x \tag{57}$$

9.6. Expressions

The meaning function **E** takes an expression, environment and state, and gives the relevant value. Since there is no change to the state, this means that Tosca is a language with no side effects.

9.6.1. Constants

The meaning of a constant is just the constant's actual value.

$$\mathbf{E}[\![\chi]\!]\rho\Sigma = \chi \tag{58}$$

(Note that the χ inside the brackets represents the syntactic literal constant, and the one outside the brackets represents its mathematical value.)

9.6.2. Identifiers

The meaning of an expression consisting of an identifier is the identifier's value, found by first using the environment function to get the store location, then the store function to get the corresponding value.

$$\mathbf{E}[\![\xi]\!]\rho(\sigma, \iota, o) = \sigma(\rho[\![\xi]\!]) \tag{59}$$

9.6.3. Binary Operators

The value of the expression consisting of a binary operator applied to two subexpressions is the corresponding mathematical operator applied to the values of the subexpressions (remember there are no side effects, so evaluating the first subexpression changes neither the environment nor the store):

$$\mathbf{E}[\![(\epsilon_1 \Omega \epsilon_2)]\!]\rho\Sigma = \mathbf{B}[\![\Omega]\!](\mathbf{E}[\![\epsilon_1]\!]\rho\Sigma, \mathbf{E}[\![\epsilon_2]\!]\rho\Sigma) \tag{60}$$

For particular operators, this can be written in infix form:

$$\mathbf{E}[\![(\epsilon_1 + \epsilon_2)]\!]\rho\Sigma = \mathbf{E}[\![\epsilon_1]\!]\rho\Sigma + \mathbf{E}[\![\epsilon_2]\!]\rho\Sigma \tag{61}$$

$$\mathbf{E}[\![(\epsilon_1 < \epsilon_2)]\!]\rho\Sigma = \mathbf{E}[\![\epsilon_1]\!]\rho\Sigma < \mathbf{E}[\![\epsilon_2]\!]\rho\Sigma \tag{62}$$

$$\mathbf{E}[\![(\epsilon_1 \textbf{ and } \epsilon_2)]\!]\rho\Sigma = \mathbf{E}[\![\epsilon_1]\!]\rho\Sigma \wedge \mathbf{E}[\![\epsilon_2]\!]\rho\Sigma \tag{63}$$

$$\text{etc.} \tag{64}$$

9.6.4. Unary Operators

The meaning of a unary operator applied to an expression is the corresponding mathematical operator applied to the meaning of that expression:

$$\mathbf{E}[\![\Psi\epsilon]\!]\rho\Sigma = \mathbf{O}[\![\Psi]\!](\mathbf{E}[\![\epsilon]\!]\rho\Sigma) \tag{65}$$

9.7. Commands

The meaning function **C** (**CL**) takes a command (list of commands), environment and state, and gives the new state that results from executing the command (list of commands). Two meaning functions are defined, in order to facilitate the subsequent Prolog translation.

9.7.1. Local Block

Forming a local block with local declarations means executing the commands in the environment modified by the declaration:

$$\mathbf{C}[\![\textbf{begin } \delta \,;; \Gamma \textbf{ end}]\!]\rho = \mathbf{CL}[\![\Gamma]\!](\mathbf{D}[\![\delta]\!]\rho) \tag{66}$$

Forming a local block with no local declarations means executing the commands in the original environment:

$$\mathbf{C}[\![\textbf{begin } \Gamma \textbf{ end}]\!] = \mathbf{CL}[\![\Gamma]\!] \tag{67}$$

The meaning of a list of commands is that of the tail of the list applied to the result of the first command in the list. A list consisting of a single command has the same meaning as that command:

$$\mathbf{CL}[\![\gamma ; \Gamma]\!]\rho = (\mathbf{CL}[\![\Gamma]\!]\rho) \circ (\mathbf{C}[\![\gamma]\!]\rho) \tag{68}$$

$$\mathbf{CL}[\![\gamma]\!] = \mathbf{C}[\![\gamma]\!] \tag{69}$$

Notice that the environment for the rest of the list is the same as for the first command. A command cannot leave the environment changed (as can be seen from the type of **C**); although it can locally create a new environment via a **begin_end** block, it reverts to the old environment when it has completed.

9.7.2. Skip Statement

The skip statement does nothing; it leaves the state unchanged:

$$\mathbf{C}[\![\textbf{skip}]\!]\rho = \mathit{Identity} \tag{70}$$

9.7.3. Assignment

Assignment updates the store by mapping the identifier's store location to the value of the expression:

$$\mathbf{C}[\![\xi := \epsilon]\!]\rho\Sigma = (\sigma \oplus \{\rho[\![\xi]\!] \mapsto \mathbf{E}[\![\epsilon]\!]\rho\Sigma\}, \iota, o) \tag{71}$$

where

$$\Sigma = (\sigma, \iota, o)$$

9.7.4. Loop

The meaning of a **while_do** command is to use the value of the expression to choose between applying the subcommand and then the whole command again, and doing nothing.

$$\begin{aligned}\mathbf{C}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]\rho\Sigma = (&\text{if } \mathbf{E}[\![\epsilon]\!]\rho\Sigma = \mathit{True}\\ &\text{then } \mathbf{C}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]\rho \circ \mathbf{C}[\![\gamma]\!]\rho\\ &\text{else } \mathit{Identity})\Sigma\end{aligned} \tag{72}$$

9.7.5. Choice

The meaning of an **if_then_else** command is to use the value of the expression to choose between selecting the first or second subcommand.

$$\begin{aligned}\mathbf{C}[\![\textbf{if } \epsilon \textbf{ then } \gamma_1 \textbf{ else } \gamma_2]\!]\rho\Sigma = (&\text{if } \mathbf{E}[\![\epsilon]\!]\rho\Sigma = \mathit{True}\\ &\text{then } \mathbf{C}[\![\gamma_1]\!] \text{ else } \mathbf{C}[\![\gamma_2]\!])\rho\Sigma\end{aligned} \tag{73}$$

9.7.6. Procedure Call

The meaning of a procedure call is the computation that forms the body of the procedure. Note that the computation takes place in the environment of the declaration, and the state of the call.

$$\mathbf{C}[\![\xi]\!]\rho\Sigma = \rho[\![\xi]\!]\Sigma \tag{74}$$

Since $\rho[\![\xi]\!] = \mathbf{C}[\![\gamma]\!]\rho_{decl}$, where γ is the body of the procedure declaration and ρ_{decl} the declaration environment, the procedure call's meaning can also be written as

$$\mathbf{C}[\![\xi]\!]\rho\Sigma = \mathbf{C}[\![\gamma]\!]\rho_{decl}\Sigma \tag{75}$$

9.7.7. Input and Output

Input removes an integer from the input list, and assigns it to the variable:

$$\mathbf{C}[\![\textbf{input } \xi]\!]\rho(\sigma, \iota, o) = (\sigma \oplus \rho[\![\xi]\!] \mapsto \text{head } \iota, \text{ tail } \iota, o) \tag{76}$$

Output appends the value of the expression to the output list:

$$[\![\textbf{output } \epsilon]\!]\rho\Sigma = (\sigma, \iota, \text{append}(o, \mathbf{E}[\![\epsilon]\!]\rho\Sigma)) \tag{77}$$

where

$$\Sigma = (\sigma, \iota, o)$$

10. Tosca's Static Type Checking Semantics

The form of this definition closely mirrors the form for the dynamic semantics above.

The static type checking can be used to check the well-typedness of a program. Only if the whole program is well-typed is the dynamic semantics defined. This explains why there is no need to check, for example, in the dynamic definition of the procedure call that the identifier denotes a procedure and not an integer – if the program is well-typed, it does.

10.1. The Domains

For this non-standard interpretation, the state describes the mapping from identifiers to their types. There is no need for the equivalent of environment and store – the type is constant throughout the scope of the declaration.

$$\textit{Type} = \{\textit{Int}, \textit{Bool}, \textit{Proc}, \textit{Wrong}\} \tag{78}$$

$$\tau : \textit{TState} = \textit{Identifier} \to \textit{Type} \tag{79}$$

10.2. Types of the Meaning Functions

Binary operator **TB**: $BinOp \rightarrow Type \times Type \rightarrow Type$ (80)

Command **TC**: $Cmd \rightarrow TState \rightarrow \{TypeOk, TypeWrong\}$ (81)

Command list **TCL**: $CmdList \rightarrow TState \rightarrow \{TypeOk, TypeWrong\}$ (82)

Declaration **TD**: $Decl \rightarrow TState \rightarrow TState$ (83)

Expression **TE**: $Expr \rightarrow TState \rightarrow Type$ (84)

Unary operator **TO**: $UnyOp \rightarrow Type \rightarrow Type$ (85)

10.3. Declarations

The meaning function **TD** takes a declaration and type state, and gives a new type state, with the declaration added.

A declared variable is added to the type state with type *Int*:

$$\mathbf{TD}[\![\mathbf{var}\ \xi]\!]\tau = \tau \oplus \{\xi \mapsto Int\} \tag{86}$$

A declared procedure is added to the type state with type *Proc* provided that its body is well-typed:

$$\begin{aligned}\mathbf{TD}[\![\mathbf{proc}\ \xi = \gamma]\!]\tau = \text{if } &\mathbf{TC}[\![\gamma]\!]\tau = TypeOk \\ &\text{then } \tau \oplus \{\xi \mapsto Proc\} \\ &\text{else } \tau \oplus \{\xi \mapsto Wrong\}\end{aligned} \tag{87}$$

Sequential combination of declarations means apply the second declaration to the result of applying the first declaration.

$$\mathbf{TD}[\![\delta_1 ; \delta_2]\!] = \mathbf{TD}[\![\delta_2]\!] \circ \mathbf{TD}[\![\delta_1]\!] \tag{88}$$

10.4. Operators

The meaning functions **TB** and **TO** take binary and unary operators and map them to functions between types. Arithmetic operators expect *Ints* and return *Ints*, comparison operators expect *Ints* and return *Bools*, and logical operators expect *Bools* and return *Bools*:

$$\mathbf{TB}[\![\Omega_\alpha]\!](t_1, t_2) = \text{if } (t_1 = Int \wedge t_2 = Int) \text{ then } Int \text{ else } Wrong \tag{89}$$

$$\mathbf{TB}[\![\Omega_\chi]\!](t_1, t_2) = \text{if } (t_1 = Int \wedge t_2 = Int) \text{ then } Bool \text{ else } Wrong \tag{90}$$

$$\mathbf{TB}[\![\Omega_\lambda]\!](t_1, t_2) = \text{if } (t_1 = Bool \wedge t_2 = Bool) \text{ then } Bool \text{ else } Wrong \tag{91}$$

$$\mathbf{TO}[\![\Psi_\alpha]\!]t = \text{if } t = Int \text{ then } Int \text{ else } Wrong \tag{92}$$

$$\mathbf{TO}[\![\Psi_\lambda]\!]t = \text{if } t = Bool \text{ then } Bool \text{ else } Wrong \tag{93}$$

10.5. Expressions

The meaning function **TE** takes an expression and type state, and gives the type.

10.5.1. Constants

The type of an expression consisting of a constant is *Bool* for **true** and **false**, and *Int* for a number:

$$\mathbf{TE}[\![\chi]\!]t = \text{if } (\chi = \mathbf{true} \vee \chi = \mathbf{false}) \text{ then } \mathit{Bool} \text{ else } \mathit{Int} \tag{94}$$

(Note that the χ on the left of the equation represents the syntactic literal constant, and the one on the right represents its mathematical value.)

10.5.2. Identifiers

The type of an expression consisting of an identifier is given by the type state function:

$$\mathbf{TE}[\![\xi]\!]t = \text{if } \tau[\![\xi]\!] = \mathit{Int} \text{ then } \mathit{Int} \text{ else } \mathit{Wrong} \tag{95}$$

10.5.3. Binary Operators

The type of a binary operator applied to two subexpressions is determined by the types of the subexpressions, and the definition of **TB** given above:

$$\mathbf{TE}[\![(\epsilon_1 \Omega \epsilon_2)]\!]\tau = \mathbf{TB}[\![\Omega]\!](\mathbf{TE}[\![\epsilon_1]\!]\tau, \mathbf{TE}[\![\epsilon_2]\!]\tau) \tag{96}$$

10.5.4. Unary Operators

The type of a unary operator applied to an expression is determined by the types of the expression, and the definition of **TO** given above:

$$\mathbf{TE}[\![\Psi\epsilon]\!]\tau = \mathbf{TO}[\![\Psi]\!](\mathbf{TE}[\![\epsilon]\!]\tau) \tag{97}$$

10.6. Commands

The meaning function **TC** takes a command and type state, and gives *TypeOk* or *TypeWrong*, depending on whether the command is well-typed or not. Similarly for lists of commands.

10.6.1. Local Block

Type checking a local block with local declarations means checking the commands in the environment modified by the declaration:

$$\mathbf{TC}[\![\mathbf{begin}\ \delta\ ;;\Gamma\ \mathbf{end}]\!]\tau = \mathbf{TCL}[\![\Gamma]\!](\mathbf{TD}[\![\delta]\!]\tau) \tag{98}$$

Type checking a local block with no local declarations means checking the commands in the original environment:

$$\mathbf{TC}[\![\mathbf{begin}\ \Gamma\ \mathbf{end}]\!] = \mathbf{TCL}[\![\Gamma]\!] \tag{99}$$

If any one command in a list of commands is *TypeWrong*, then the whole list is *TypeWrong*. A list consisting of a single command has the same meaning as that command:

$$\mathbf{TCL}[\![\gamma ; \Gamma]\!]\tau = \text{if } \mathbf{TC}[\![\gamma]\!]\tau = \mathit{TypeOk} \text{ then } \mathbf{TCL}[\![\Gamma]\!]\tau \text{ else } \mathit{TypeWrong} \tag{100}$$

$$\mathbf{TCL}[\![\gamma]\!] = \mathbf{TC}[\![\gamma]\!] \tag{101}$$

10.6.2. Skip Statement

The skip statement always type checks *TypeOk*:

$$\mathbf{TC}[\![\textbf{skip}]\!]\tau = \mathit{TypeOk} \tag{102}$$

10.6.3. Assignment

Assignment type checks *TypeOk* if the identifier is an *Int* (not a *Proc*) and the expression has type *Int* (identifiers can only be integers, not booleans or anything else):

$$\mathbf{TC}[\![\xi := \epsilon]\!]\tau = \text{if } (\tau[\![\xi]\!] = \mathit{TypeOk} \wedge \mathbf{TE}[\![\epsilon]\!]\tau = \mathit{Int}) \text{ then } \mathit{TypeOk} \text{ else } \mathit{TypeWrong} \tag{103}$$

10.6.4. Loop

The command type checks *TypeOk* only if the expression has type *Bool* and the subcommand type checks *TypeOk*.

$$\mathbf{TC}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]\tau = \text{if } (\mathbf{TE}[\![\epsilon]\!]\tau = \mathit{Bool} \wedge \mathbf{TC}[\![\gamma]\!]\tau = \mathit{TypeOk}) \text{ then } \mathit{TypeOk} \text{ else } \mathit{TypeWrong} \tag{104}$$

10.6.5. Choice

The command type checks *TypeOk* only if the expression has type *Bool* and both subcommands type check *TypeOk*.

$$\mathbf{TC}[\![\textbf{if } \epsilon \textbf{ then } \gamma_1 \textbf{ else } \gamma_2]\!]\tau = \text{if } (\mathbf{TE}[\![\epsilon]\!]\tau = \mathit{Bool} \wedge \mathbf{TC}[\![\gamma_1]\!]\tau = \mathit{TypeOk} \wedge \mathbf{TC}[\![\gamma_2]\!]\tau = \mathit{TypeOk}) \text{ then } \mathit{TypeOk} \text{ else } \mathit{TypeWrong} \tag{105}$$

10.6.6. Procedure Call

A procedure call type checks *TypeOk* only if the type of the identifier is *Proc*:

$$\mathbf{TC}[\![\xi]\!]\tau = \text{if } \tau[\![\xi]\!] = \mathit{Proc} \text{ then } \mathit{TypeOk} \text{ else } \mathit{TypeWrong} \tag{106}$$

10.6.7. Input and Output

Input type checks *TypeOk* if the identifier is an *Int* (identifiers can only be integers, not booleans or anything else):

$$\mathbf{TC}[\![\mathbf{input}\ \xi]\!]\tau = \text{if } \tau[\![\xi]\!] = \mathit{Int} \text{ then } \mathit{TypeOk} \text{ else } \mathit{TypeWrong} \tag{107}$$

Output type checks *TypeOk* if the expression has type *Int*:

$$\mathbf{TC}[\![\mathbf{output}\ \epsilon]\!]\tau = \text{if } \mathbf{TE}[\![\epsilon]\!]\tau = \mathit{Int} \text{ then } \mathit{TypeOk} \text{ else } \mathit{TypeWrong} \tag{108}$$

11. Tosca's Static Usage Semantics

This non-standard interpretation of the semantics describes a way to check that variables are not used before they are initialised.

This semantics is only defined if the program is well-typed. This simplifies various definitions, for example, if an identifier is used in an expression, it can be assumed that it is both declared and an integer (not a procedure).

11.1. The Domains

The *UStore* is the mapping from usage store locations to the current use state of that variable:

$$\zeta : \mathit{UStore} = \mathit{ULocn} \to \mathit{Use} \tag{109}$$

A use value is either *Use_bad* (for a variable that is used before being initialised, or an expression that uses a *Use_bad* variable in a subexpression) or *Use_ok* (for a variable that has been initialised to something, possibly bad, before being used, or an expression that uses only *ok* variables in its subexpressions).

$$\mathit{Use} = \{\mathit{Use\_bad}, \mathit{Use\_ok}\} \tag{110}$$

The function *worseof* takes two use values and returns the worse one:

$$\mathit{worseof} : \mathit{Use} \times \mathit{Use} \to \mathit{Use} \tag{111}$$

$$\mathit{worseof}(u, v) = \text{if } u = \mathit{Use\_bad} \vee v = \mathit{Use\_bad} \text{ then } \mathit{Use\_bad} \text{ else } \mathit{Use\_ok} \tag{112}$$

The usage environment describes the mapping from identifiers to what they denote. For a variable, it is the relevant store location, which will map to a value that can change as the computation progresses. For a procedure declaration, it is the computation denoted by the procedure body, which does not change.

$$v : \mathit{UEnv} = \mathit{Identifier} \to \mathit{ULocn} + \mathit{UProc} \tag{113}$$

A *UProc* is a mapping from one usage store to another that results from checking the procedure body:

$$\mathit{UProc} = \mathit{UStore} \to \mathit{UStore} \tag{114}$$

11.2. Types of the Meaning Functions

Binary operator **UB**: $BinOp \rightarrow Use \times Use \rightarrow Use$ (115)

Command **UC**: $Cmd \rightarrow UEnv \rightarrow UStore \rightarrow UStore$ (116)

Command list **UCL**: $CmdList \rightarrow UEnv \rightarrow UStore \rightarrow UStore$ (117)

Declaration **UD**: $Decl \rightarrow UEnv \rightarrow UEnv$ (118)

Expression **UE**: $Expr \rightarrow UEnv \rightarrow UStore \rightarrow Use \times UStore$ (119)

Unary operator **UO**: $UnyOp \rightarrow Use \rightarrow Use$ (120)

11.3. Declarations

The meaning function **UD** takes a declaration and a usage environment, and gives a new usage environment with the declaration added.

The declared variable is added to the environment by mapping it to a previously unallocated location.

$$\mathbf{UD}[\![\textbf{var } \xi]\!]v = v \oplus \{\xi \mapsto uloc\} \tag{121}$$

where

$$uloc : ULocn \notin \operatorname{ran} v$$

The declared procedure is added to the usage environment by mapping it to the computation denoted by the body of the declaration, in the declaration's usage environment.

$$\mathbf{UD}[\![\textbf{proc } \xi = \gamma]\!]v = v \oplus \{\xi \mapsto \mathbf{UC}[\![\gamma]\!]v\} \tag{122}$$

Sequential combination of declarations means apply the second declaration to the result of applying the first declaration.

$$\mathbf{UD}[\![\delta_1 ; \delta_2]\!] = \mathbf{UD}[\![\delta_2]\!] \circ \mathbf{UD}[\![\delta_1]\!] \tag{123}$$

11.4. Operators

The meaning functions **UB** and **UO** take binary and unary operators and map them to functions between *Uses*. The binary operator returns the *worse* use, the unary operator returns the identity function on *Use*:

$$\mathbf{UB}[\![\Omega]\!](u_1, u_2) = worseof(u_1, u_2) \tag{124}$$

$$\mathbf{UO}[\![\Psi]\!]u = u \tag{125}$$

11.5. Expressions

UE takes an expression, a usage environment and store, and gives a use value and a possibly modified store (which occurs if an uninitialised variable is used in the expression).

11.5.1. Constants

An expression consisting of a constant is *Use_ok*, and leaves the store unchanged.

$$\mathbf{UE}[\![\chi]\!]v\zeta = (\mathit{Use\_ok},\ \zeta) \tag{126}$$

11.5.2. Identifiers

If the identifier is uninitialised, it will not be in the domain of the store. The new store will be modified to set the identifier to *Use_bad*, and the use value of the expression will be *Use_bad*.

If the identifier is in the domain of the store, the store will not be changed, and the expression's use value will be set to that of the identifier.

This is the only kind of *expression* that directly changes the store (the assignment command can also change the store).

$$\begin{aligned}\mathbf{UE}[\![\xi]\!]v\zeta = \text{if } & \zeta(v[\![\xi]\!]) = \perp \\ & \text{then } (\mathit{Use\_bad},\ \zeta \oplus \{v[\![\xi]\!] \mapsto \mathit{Use\_bad}\}) \\ & \text{else } (\zeta(v[\![\xi]\!],\ \zeta)\end{aligned} \tag{127}$$

11.5.3. Binary Operators

The use of a binary operator applied to two expressions is *Use_ok* only if both expressions are *Use_ok*. Notice that the store can be changed by either or both subexpressions, if they use an uninitialised variable.

$$\mathbf{UE}[\![(\epsilon_1 \Omega \epsilon_2)]\!]v\zeta = (\mathbf{UB}[\![\Omega]\!](u_1,\ u_2),\ \zeta_2) \tag{128}$$

where

$$(u_2,\ \zeta_2) = \mathbf{UE}[\![\epsilon_2]\!]v\zeta_1$$
$$(u_1,\ \zeta_1) = \mathbf{UE}[\![\epsilon_1]\!]v\zeta$$

Notice that this particular definition implies a particular order of evaluation. Alternative definitions could be proposed, using a function like *worseof* (see later).

11.5.4. Unary Operators

The use of a unary operator applied to an expression is only *ok* if the expression is *ok*:

$$\mathbf{UE}[\![\Psi\epsilon]\!]v\zeta = (\mathbf{UO}[\![\Psi]\!]u,\ \zeta_1) \tag{129}$$

where

$$(u,\ \zeta_1) = \mathbf{UE}[\![\epsilon]\!]v\zeta$$

Since **UO** gives the identity on *Use*, this can be simplified to:

$$\mathbf{UE}[\![\Psi\epsilon]\!] = \mathbf{UE}[\![\epsilon]\!] \tag{130}$$

11.6. Commands

The meaning function **UC** takes a command, use environment and store, and gives the new store that results from checking the command. Similarly for lists of commands.

11.6.1. Local Block

Use checking a local block with local declarations means checking the commands in the environment modified by the declaration:

$$\mathbf{UC}[\![\mathbf{begin}\ \delta\ ;;\Gamma\ \mathbf{end}]\!]\upsilon = \mathbf{UCL}[\![\Gamma]\!](\mathbf{UD}[\![\delta]\!]\upsilon) \tag{131}$$

Use checking a local block with no local declarations means checking the commands in the original environment:

$$\mathbf{UC}[\![\mathbf{begin}\ \Gamma\ \mathbf{end}]\!] = \mathbf{UCL}[\![\Gamma]\!] \tag{132}$$

Use checking a list of commands means checking the tail of the list applied to checking the first command in the list. A list consisting of a single command has the same meaning as that command:

$$\mathbf{UCL}[\![\gamma\ ;\Gamma]\!]\upsilon = (\mathbf{UCL}[\![\Gamma]\!]\upsilon) \circ (\mathbf{UC}[\![\gamma]\!]\upsilon) \tag{133}$$

$$\mathbf{UCL}[\![\gamma]\!] = \mathbf{UC}[\![\gamma]\!] \tag{134}$$

11.6.2. Skip Statement

The skip statement does nothing; it leaves the store unchanged:

$$\mathbf{UC}[\![\mathbf{skip}]\!]\upsilon\zeta = \zeta \tag{135}$$

11.6.3. Assignment

The new use store of input depends on the use state of the identifier, after evaluating the expression (to catch usage like $\mathbf{x := (x + 1)}$, where $\mathbf{x}$ has not previously been initialised). Provided the identifier has not yet been used (either properly or improperly, possibly in the expression), its use becomes *Use_ok,* otherwise its use is left unchanged. Notice this is the case whether the expression is *Use_ok* or *Use_bad* – this semantics does not worry if a variable has been set to a *Use_bad* expression, it just notes which variables are used before they are set to anything at all.

$$\mathbf{UC}[\![\xi := \epsilon]\!]\upsilon\zeta = \text{if } \zeta_1(\upsilon[\![x]\!]) = \perp \text{ then } \zeta_1 \oplus \{\upsilon[\![x]\!] \mapsto \mathit{Use\_ok}\} \text{else } \zeta_1 \tag{136}$$

where

$$(u, \zeta_1) = \mathbf{UE}[\![\epsilon]\!]\upsilon\zeta$$

11.6.4. Loop

The new use store is determined by the body of the command, evaluated in the possibly changed store of the expression:

$$\mathbf{UC}[\![\textbf{while } \epsilon \textbf{ do } \gamma]\!]v\zeta = \mathbf{UC}[\![\gamma]\!]v\zeta_1 \tag{137}$$

where

$$(u, \zeta_1) = \mathbf{UE}[\![\epsilon]\!]v\zeta$$

11.6.5. Choice

The new use store is determined by the worse of the two subcommands, evaluated in the possibly changed store of the expression:

$$\mathbf{UC}[\![\textbf{if } \epsilon \textbf{ then } \gamma_1 \textbf{ else } \gamma_2]\!]v\zeta = \mathbf{UC}[\![\gamma_1]\!]v\zeta_1 \otimes \mathbf{UC}[\![\gamma_2]\!]v\zeta_1 \tag{138}$$

where

$$(v, \zeta_1) = \mathbf{UE}[\![\epsilon]\!]v\zeta$$

The function $\otimes$ is defined to take two use stores and return a use store which combines the *worst* properties of each (notice that the same environment is used in both branches of the choice). For example, if neither **y** nor **z** have been previously initialised, then after:

```
if expr then
  begin
    y:= (y + 1); z := 1
  end
else
  begin
    y := 1; z := (z + 1)
  end
```

we would want both **y** and **z** to be set to *Use_bad*. So $\zeta_1 \otimes \zeta_2$ is defined by the *worseof* function where the domains of ζ_1 and ζ_2 coincide:

$$\otimes : UStore \times UStore \to UStore \tag{139}$$

$$\begin{aligned}\zeta_1 \otimes \zeta_2 = {} & \{l \in \text{dom } \zeta_1 - \text{dom } \zeta_2 \bullet l \mapsto \zeta_1 l\} \cup \\ & \{l \in \text{dom } \zeta_2 - \text{dom } \zeta_1 \bullet l \mapsto \zeta_2 l\} \cup \\ & \{l \in \text{dom } \zeta_1 \cap \text{dom } \zeta_2 \bullet l \mapsto worseof(\zeta_1 l, \zeta_2 l)\}\end{aligned} \tag{140}$$

Note that this provides a rather strict definition of potentially unused variables, which will eliminate programs that might otherwise be thought to be "correct". It is probably appropriate to have such a strict definition for a safety-critical language. More to the point, however, it does provide an unambiguous definition that can be reasoned about, and which provides a basis for criticism if necessary.

11.6.6. Procedure Call

The use value of a procedure call is the use that forms the body of the procedure.

$$\mathbf{UC}[\![\xi]\!]v\zeta = v[\![\xi]\!]\zeta \tag{141}$$

Since $v[\![\xi]\!] = \mathbf{UC}[\![\gamma]\!]v_{decl}$, where γ is the body of the procedure declaration and v_{decl} the declaration environment, the procedure call's meaning can also be written as

$$\mathbf{UC}[\![\xi]\!]v\zeta = \mathbf{UC}[\![\gamma]\!]v_{decl}\zeta \tag{142}$$

11.6.7. Input and Output

The new use store of input depends on the use state of the identifier. Provided the identifier has not yet been used (either properly or improperly), its use becomes *Use_ok*, otherwise its use is left unchanged:

$$\mathbf{UC}[\![\textbf{input } \xi]\!]v\zeta = \text{if } \zeta(v[\![\xi]\!]) = \perp \text{ then } \zeta \oplus \{v[\![\xi]\!] \mapsto \mathit{Use\_ok}\} \text{else } \zeta \tag{143}$$

The new use store produced by output depends on that of the expression:

$$\mathbf{UC}[\![\textbf{output } \epsilon]\!]v\zeta = \zeta_1 \tag{144}$$

where

$$(u, \zeta_1) = \mathbf{UE}[\![\epsilon]\!]v\zeta$$

12. Technology Demonstration

In support of the foregoing arguments, in order to demonstrate feasibility, we have prototyped a Tosca compiler as a Prolog DCTG. We have used an enhanced variant of the method in [AbD89, appendix II.3.2] for translating the DCTG into its Prolog equivalent.

Four semantics (type checking, use checking, dynamic and operational) of this language have been specified formally. All but the operational semantics are reproduced in Appendix B. The clarity of such specifications for language systems is immediately apparent.

This specification was then translated into Prolog [AbD89]. This generated a type checker (from the type checking semantics), a use checker (from the use checking semantics), an interpreter (from the dynamic semantics) and a compiler (from the operational semantics) for Tosca.

There is a *formally specified* microprocessor, Viper [Coh87, Kem88a, Kem88b, Kernd]. Such a processor has a better chance than most of executing its assembly language correctly (as defined by that language's semantics). For this reason, we have chosen Viper as our target machine. The appropriate target language for our purposes is a low level assembly language called Vital, and so our demonstration compiler translates Tosca into Vital.

As an example, consider a simple Tosca program to compute the numeric square of selected inputs:

```
begin
  var limit; var n; var sq;;
  input limit;
  n := 1; sq := 1;
  output sq;
  while (n < limit) do
  begin
    sq := ((sq + 1) + (n + n));
    n := (n + 1);
```

```
    output sq
  end
end
```

The interpreter output (where the 4 in italics is input by the user) looks like:

```
⟨⟨⟨input ''limit''. ⟨⟨⟨ 4.
⟩⟩⟩output⟩⟩⟩1
⟩⟩⟩output⟩⟩⟩4
⟩⟩⟩output⟩⟩⟩9
⟩⟩⟩output⟩⟩⟩16
```

The corresponding Vital code generated by the compiler is:

```
    --input
  A:=INPUT(0)
  (limit):=A
    --assignment
  A:=1
  (n):=1
    --assignment
  A:=1
  (sq):=A
    --output
  A:=(sq)
  OUTPUT A,(1)
    --while
label1:
  A:=(n)
  A:=A-(limit)
  TEST A⟩=0
  IF B GOTO label2
    --do
    --assignment
  A:=(sq)
  A:=A+(n)
  A:=A+(n)
  A:=A+1
  (sq):=A
    --assignment
  A:=(n)
  A:=A+1
  (n):=A
  --output
  A:=(sq)
  OUTPUT A,(1)
  GOTO label1
label2:
    --endwhile
  STOP
limit:0
n:0
sq:0
```

We can run each Vital program produced by our compiler by assembling it, and then executing it on a fast Viper simulator we have constructed at Logica (since this simulator successfully executes RSRE's VipTest, it is a valid simulation). Each program has executed correctly, generating the same outputs as the interpreter.

13. Summary and Conclusions

In summary, our approach to building a demonstrably correct compiler has the following steps:

1. Specify a denotational semantics for the source language. Many problems and ambiguities arising in the language definition can be resolved at this stage.
2. Write this semantics as a Prolog DCTG. This provides an interpreter, which can be used to provide further validation for the proposed semantics.
3. Specify a denotational semantics for the target language.
4. Specify an operational semantics of the source language as code templates in the target language. Calculate the meaning of these templates, using the target language semantics, to prove that they have the same meaning as the corresponding source language constructs. This proves that the proposed compiler performs a correct translation.
5. Write the operational semantics as a DCTG. This provides a compiler.

Further non-standard semantics can be specified as required, for example, to provide a type-checker and use-checker component to the compiler. Notice that each one, when written as a DCTG, immediately provides the relevant checker – no further translation step is necessary.

Given that we were using a well understood and formal approach (which has had considerable thought expended upon it in the past) and existing supporting tools technology, developing this prototype produced some interesting findings:

- It took three days to specify a dynamic semantics for Tosca and build the interpreter.
- It took another day to specify and build the compiler for the selected target language, Vital.
- Each non-standard semantics – type checking and use checking – took a further one-half day to specify and add to the prototype. This included correcting a small error in the formal specification of the use semantics.
- Once the usual typographical errors were removed, the prototype system ran correctly. It has never produced incorrect output. (Even the typos did not result in incorrect output, rather Prolog's `no` output).

In contrast, the smaller Viper simulator took five days to build using a more traditional approach – design and code using C. Despite being developed using "best practice", it has exhibited a number of minor errors.

The language and compiler described here were developed as a prototype demonstration, not as a commercial product. The timescale for this work was

measured in days; a properly rigorous development of a compiler for a full language would take longer! Since none of the usual commercial quality procedures or rigorous testing were followed, we would not be surprised if mistakes are found in either the semantics or the compiler. Indeed, since one of our aims is to produce a specification and implementation of sufficient clarity to facilitate the discovery of such errors, we are almost hoping they will! Notwithstanding this caveat, we believe that if such a development as outlined above is carried out under the usual quality control and review procedures, a correct, and demonstrably correct, compiler can be produced successfully, and with quantifiable benefits.

Acknowledgements

This work was carried out by Logica as part of a study of implementation techniques for a trustworthy compiler for Viper, commissioned by RSRE, and we would like to thank the staff of RSRE for their input. In particular, detailed technical assistance was provided by John Kershaw, Clive Pygott and Ian Currie, and technical background was provided by Nick Peeling and Roger Smith.

We would also like to thank David Brazier, Tim Hoverd, Mike Flynn and Jon Brumfitt of Logica for helpful discussions.

References

[AbD89] Abramson, H. and Dahl, V.: *Logic Grammars.* Symbolic Computation Series, Springer, 1989.

[All86] Allison, L.: *A Practical Introduction to Denotational Semantics.* Cambridge University Press, 1986.

[Andnd] Andrews, D.: A Formal Definition of Modula-2.

[BC85] Bergeretti, J. F. and Carré, B.: Information Flow and Data Flow Analysis of While Programs. *ACM Transactions on Programming Languages and Systems,* **7,** 37–61 (1985).

[Bra84] Bramson, B. D.: Malvern's Program Analysers. *RSRE Research Review,* 1984.

[Car89] Carré, B.: Reliable Programming in Standard Languages. In *High Integrity Software,* C. Sennett (ed), Pitman, 1989.

[CoC77] Cousot, P. and Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Constuction or Approximation of Fixed Points. In *Proc. Fourth Annual ACM Symposium on the Principles of Programming Languages,* 1977.

[ClM87] Clocksin, W. F. and Mellish, C. S.: *Programming in Prolog.* 3rd edition, Springer, 1987.

[Coh79] Cohn, A.: Machine Assisted Proofs of Recursion Implementation. PhD thesis, University of Edinburgh, 1979.

[Coh87] Cohn, A.: A Proof of Correctness of the Viper Microprocessor: The First Level. Technical Report 104, University of Cambridge, 1987.

[Gor79] Gordon, M. J. C.: *The Denotational Description of Programming Languages – An Introduction.* Springer, 1979.

[HoJ90] Hoare, C. A. R. and He, Jifeng: Refinement Algebra Proves Compiling Specification correct. In *Third BCS-FACS Refinement Workshop,* C. C. Morgan and J. C. P. Woodcock (eds), Workshops in Computing, Springer, 1990.

[Kem88a] Kemp, D. H.: Specification of Viper1 in Z. RSRE Memorandum 4195, Royal Signals and Radar Establishment, 1988.

[Kem88b] Kemp, D. H. Specification of Viper2 in Z. RSRE Memorandum 4217, Royal Signals and Radar Establishment, October 1988.

[Kernd] Kershaw, J.: The Viper Microprocessor. RSRE Memorandum 87014, Royal Signals and Radar Establishment.

[Knu68] Knuth, D. E.: Semantics of Context-Free Languages. *Mathematical Systems Theory,* **2**(2), 127–145 (1968). Correction, **5**(1), 95–96 (1971).
[Lee89] Lee, P.: *Realistic Compiler Generation.* Foundations of Computing series, MIT Press, 1989.
[Mor73] Morris, F. L.: Advice on Structuring Compilers and Proving Them Correct. *Proc. First Annual ACM Symposium on Principles of Programming Languages,* 144–152, 1973.
[Mos75] Mosses, P. D.: Mathematical Semantics and Compiler Generation. PhD thesis, University of Oxford, 1975.
[McP66] McCarthy, J. and Painter, J.: Correctness of a Compiler for Arithmetic Expressions. Technical Report AIM-40, Stanford University, 1966.
[MiW72] Milner, R. and Weyhrauch, R.: Proving Compiler Correctness in a Mechanized Logic. *Machine Intelligence,* **7** (1972).
[Pau81] Paulson, L.: A Compiler Generator for Semantic Grammars. Ph.D. thesis, Stanford University, 1981.
[Pau82] Paulson, L.: A Semantics-Directed Compiler Generator. *Proc. Ninth Annual ACM Symposium on Principles of Programming Languages,* pp. 224–239, 1982.
[Pol81] Polak, W.: *Compiler Specification and Verification.* Volume 124 of *Lecture Notes in Computer Science,* Springer, 1981.
[StL87] Stepney, S. and Lord, S. P.: Formal Specification of an Access Control System. *Software – Practice and Experience,* **17**(9), 575–593 (1987).
[StS86] Sterling, L. and Shapiro, E.: *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.
[Sto77] Stoy, J. E.: *Denotational Semantics and the Scott–Strachey Approach to Programming Language Theory.* MIT Press, 1977.
[Wan84] Wand, M.: A Semantic Prototyping System. *Proc. SIGPLAN 84 Symposium on Compiler Construction. ACM SIGPLAN Notices,* **19**(6), 213–221 (1984).
[War80] Warren, D. H.: Logic Programming and Compiler Writing. *Software – Practice and Experience,* **10,** 97–125 (1980).

Appendix A. Calculation of the Meaning of a Template

As a concrete example of the calculation of the meaning of a code template, described in Section 3, consider Tosca's **if_then_else** statement (Section 9.7.5). It has denotation (meaning):

$$\mathbf{C}[\![\textbf{if } \epsilon \textbf{ then } \gamma_1 \textbf{ else } \gamma_2]\!]\rho\Sigma = (\text{if } \mathbf{E}[\![\epsilon]\!]\rho\Sigma = \mathit{True} \text{ then } \mathbf{C}[\![\gamma_1]\!] \text{ else } \mathbf{C}[\![\gamma_2]\!])\rho\Sigma \tag{145}$$

For the purposes of exposition, the target language chosen is a subset of the small language with `gotos` given in [All86, chapter 7], which has the same semantics as the subset of Vital used in the demonstrator, but a rather more convenient concrete syntax. For reasons of space, the semantics of this language cannot be repeated here. What is of most interest, however, is the form, and length, of the argument below. The code template in this target language is:

$$\begin{aligned}
&\phi_0\texttt{:if } \neg\epsilon \texttt{ then goto } \phi_1 \texttt{ else skip}\\
&\quad \gamma_1\\
&\quad \texttt{goto } \phi_2 \qquad\qquad (146)\\
&\phi_1\texttt{:}\gamma_2\\
&\phi_2\texttt{:skip}
\end{aligned}$$

To prove this is the *correct* template, we need to show that is has the *same* meaning, within the continuation semantics of the language with `gotos`, as the

if statement does within the Tosca semantics. Using the semantics given in [All86], the meaning of this template can be directly calculated:

$$\mathbf{P}[\![\phi_0\colon \texttt{if}\ \neg\epsilon\ \texttt{then goto}\ \phi_1\ \texttt{else skip};\gamma_1;\texttt{goto}\ \phi_2$$
$$\phi_1\colon \gamma_2$$
$$\phi_2\colon \texttt{skip}]\!]\rho\theta\sigma = \theta_0\sigma$$

where

$$\theta_0 = \mathbf{P}[\![\texttt{if}\ \neg\epsilon\ \texttt{then goto}\ \phi_1\ \texttt{else skip};\gamma_1;\texttt{goto}\ \phi_2]\!]\rho_1\theta_1 \tag{148}$$

$$\theta_1 = \mathbf{P}[\![\gamma_2]\!]\rho_1\theta_2 \tag{149}$$

$$\theta_2 = \mathbf{P}[\![\texttt{skip}]\!]\rho_1\theta \tag{150}$$

$$\rho_1 = \rho[\theta_0/\phi_0,\ \theta_1/\phi_1,\ \theta_2/\phi_2] \tag{151}$$

So the meaning of the template is $\theta_0\sigma$. Expanding out the first sequential combination gives:

$$\theta_0\sigma = \mathbf{P}[\![\texttt{if}\ \neg\epsilon\ \texttt{then goto}\ \phi_1\ \texttt{else skip}]\!]\rho_1$$
$$\{\mathbf{P}[\![\gamma_1;\texttt{goto}\ \phi_2]\!]\rho_1\theta_1\}\sigma \tag{152}$$

And expanding out the other sequential combination gives:

$$\theta_0\sigma = \mathbf{P}[\![\texttt{if}\ \neg\epsilon\ \texttt{then goto}\ \phi_1\ \texttt{else skip}]\!]\rho_1$$
$$\{\mathbf{P}[\![\gamma_1]\!]\rho_1\{\mathbf{P}[\![\texttt{goto}\ \phi_2]\!]\rho_1\theta_1\}\}\sigma \tag{153}$$

Substituting for the `goto` gives:

$$\theta_0\sigma = \mathbf{P}[\![\texttt{if}\ \neg\epsilon\ \texttt{then goto}\ \phi_1\ \texttt{else skip}]\!]\rho_1 \tag{154}$$
$$\{\mathbf{P}[\![\gamma_1]\!]\rho_1\{\rho_1[\![\phi_2]\!]\}\}\sigma \tag{155}$$

And substituting for the meaning of the label ϕ_2 gives:

$$\theta_0\sigma = \mathbf{P}[\![\texttt{if}\ \neg\epsilon\ \texttt{then goto}\ \phi_1\ \texttt{else skip}]\!]\rho_1$$
$$\{\mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\}\sigma \tag{156}$$

Expanding out the `if_then_else` gives

$$\theta_0\sigma = (\text{if } \mathbf{E}[\![\neg\epsilon]\!]\sigma \text{ then } \mathbf{P}[\![\texttt{goto}\ \phi_1]\!]$$
$$\text{else } \mathbf{P}[\![\texttt{skip}]\!])\rho_1\{\mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\}\sigma \tag{157}$$

Pulling the continuation and state inside the brackets gives

$$\theta_0\sigma = \text{if } \mathbf{E}[\![\neg\epsilon]\!]\sigma \text{ then } \mathbf{P}[\![\texttt{goto}\ \phi_1]\!]\rho_1\{\mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\}\sigma$$
$$\text{else } \mathbf{P}\ [\![\texttt{skip}]\!]\rho_1\{\mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\}\sigma \tag{158}$$

Then substituting for the `goto` in one branch and the `skip` in the other gives:

$$\theta_0\sigma = \text{if } \mathbf{E}[\![\neg\epsilon]\!]\sigma \text{ then } \rho_1[\![\phi_1]\!]\sigma \text{ else } \mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\sigma \tag{159}$$

And substituting for the meaning of the label ϕ_1 gives:

$$\theta_0\sigma = \text{if } \mathbf{E}[\![\neg\epsilon]\!]\sigma \text{ then } \theta_1\sigma \text{ else } \mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\sigma \tag{160}$$

Then substituting back the meaning of the continuation gives:

$$\theta_0\sigma = \text{if } \mathbf{E}[\![\neg\epsilon]\!]\delta \text{ then } \mathbf{P}[\![\gamma_2]\!]\rho_1\theta_2\sigma \text{ else } \mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2\sigma \tag{161}$$

γ_1 and γ_2 do not mention labels ϕ_0, ϕ_1, ϕ_2, because Tosca does not allow jumps out of a block (there are no `gotos` in Tosca), and the compiler chooses new names for the labels. This result implies that $\mathbf{P}[\![\gamma_1]\!]\rho_1\theta_2 = \mathbf{P}[\![\gamma_1]\!]\rho\theta$, etc., and hence

$$\theta_0\sigma = (\text{if } \mathbf{E}[\![\neg\epsilon]\!] \text{ then } \mathbf{P}[\![\gamma_2]\!] \text{else } \mathbf{P}[\![\gamma_1]\!])\rho\theta\sigma \tag{162}$$

Then expanding the $\neg$ operator, and rearranging, gives

$$\theta_0\sigma = (\text{if } \mathbf{E}[\![\epsilon]\!]\sigma \text{ then } \mathbf{P}[\![\gamma_1]\!] \text{else } \mathbf{P}[\![\gamma_2]\!])\rho\theta\sigma \tag{163}$$

as required.

Appendix B. Tosca's DCTG

The following shows a summary of Tosca's DCTG, as used in the technology demonstration. It has three semantics attached to each node: the two non-standard semantics, typecheck and usecheck, and the dynamic semantics, meaning, that gives the abstract meaning of the node. These provide a type checker, a use checker and an interpreter, respectively.

```
demo(Source) :-
     lexemes(Tokens, _, Source, []),
     !,
     program(Tree, Tokens, []),
     !,
     Tree^^typecheck(Type),
          writel([nl, 'type checking done', nl]),
     (    Type = type_wrong, fatalerror(['type error(s)'])
     ;    Type = type_ok
     ),
     !,
     Tree^^usecheck(UStore),
          writel([nl, 'use checking done', nl]),
     lookup(UStore, X, use_bad),
     (    X = nonebad
     ;    fatalerror(['use error(s)'])
     ),
     Tree^^meaning.
```

```
/*============================================================
*    THE MAIN PROGRAM
*============================================================*/
/*------------------------------------------------------------*
*    start with an empty environment and store
*    <program> ::= <cmd>
*------------------------------------------------------------*/
program ::=
     cmd^^C
<:>
     (typecheck(Type) ::- C^^typecheck([], Type)),
     (usecheck(UStore) ::- C^^usecheck([], [], UStore)),
     (meaning ::- C^^meaning([], [], _)).



/*============================================================
*    COMMAND LIST
*============================================================*/
/*------------------------------------------------------------*
*    <cmdList> ::= <cmd> <cmdCont>
*------------------------------------------------------------*/
cmdList ::=
     cmd^^C, cmdCont^^CL
<:>
     (typecheck(State, Type) ::-
          C^^typecheck(State, Type1),
          (    Type1 = type_ok,
               CL^^typecheck(State, Type)
          ;
               Type1 = type_wrong, Type = type_wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          C^^usecheck(UEnv, PreUStore, MidUStore),
          CL^^usecheck(UEnv, MidUStore, PostUStore)
     ),
     (meaning(Env, PreStore, PostStore) ::-
          C^^meaning(Env, PreStore, MidStore),
          CL^^meaning(Env, MidStore, PostStore)
     ).
```

```
/*-------------------------------------------------------------*
*    <cmdCont> ::= ; <cmdList>
*-------------------------------------------------------------*/
cmdCont ::=
     tSEMICOLON, !,
     cmdList^^CL
<:>
     (typecheck(State, Type) ::-
          CL^^typecheck(State, Type)
     ),
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          CL^^usecheck(UEnv, PreUStore, PostUStore)
     ),
     (meaning(Env, PreStore, PostStore) ::-
          CL^^meaning(Env, PreStore, PostStore)
     ).

/*-------------------------------------------------------------*
*    <cmdCont> ::= []
*-------------------------------------------------------------*/
cmdCont ::=
     []
<:>
     (typecheck(_, type_ok)),
     (usecheck(_, UStore, UStore)),
     (meaning(_, Store, Store)).

/*=============================================================
*    COMMANDS
*=============================================================*/
/*-------------------------------------------------------------*
*    local block (1) :
*-------------------------------------------------------------*/
cmd ::=
     tBEGIN,
     declList^^DL, tENDDECL, cmdList^^CL,
     tEND
<:>
     (typecheck(State, Type) ::-
          DL^^typecheck(State, PostState),
          CL^^typecheck(PostState, Type)
```

```
    ),
    (usecheck(UEnv, PreUStore, PostUStore) ::-
        DL^^usecheck(UEnv, PostUEnv),
        CL^^usecheck(PostUEnv, PreUStore, PostUStore)
    ),
    (meaning(Env, PreStore, PostStore) ::-
        DL^^meaning(Env, PostEnv),
        CL^^meaning(PostEnv, PreStore, PostStore)
    ).

/*-------------------------------------------------------------*
*    local block (2) :
*-------------------------------------------------------------*/
cmd ::=
    tBEGIN,
    cmdList^^CL,
    tEND
<:>
    (typecheck(State, Type) ::-
        CL^^typecheck(State, Type)
    ),
    (usecheck(UEnv, PreUStore, PostUStore) ::-
        CL^^usecheck(UEnv, PreUStore, PostUStore)
    ),
    (meaning(Env, PreStore, PostStore) ::-
        CL^^meaning(Env, PreStore, PostStore)
    ).

/*-------------------------------------------------------------*
*    <cmd> ::= skip
*-------------------------------------------------------------*/
cmd ::=
    tSKIP
<:>
    (typecheck(_, type_ok)),
    (usecheck(_, UStore, UStore)),
    (meaning(_, Store, Store)).

/*-------------------------------------------------------------
*    <cmd> ::= <id> := <expr>
*-------------------------------------------------------------*/
```

```
cmd ::=
     tIDENT^^I, tASSIGN, expr^^E
<:>
     (typecheck(State, Type) ::-
          I^^meaning(Id),
          E^^typecheck(State, TypeE),
          lookup(State, Id, TypeI),
          (    TypeI = int, TypeE = int, Type = type_ok
          ;    Type = type_wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          I^^meaning(Id),
          E^^usecheck(UEnv, PreUStore, MidUStore, _),
          lookup(UEnv, Id, ULocn),
          lookup(MidUStore, ULocn, Use),
          (    Use = bottom,
               update(MidUStore, PostUStore, ULocn, use_ok)
          ;
               PostUStore = MidUStore
          )
     ),
     (meaning(Env, PreStore, PostStore) ::-
          I^^meaning(Id),
          E^^meaning(Env, PreStore, Val),
          lookup(Env, Id, Locn),
          update(PreStore, PostStore, Locn, Val)
     ).

/*-------------------------------------------------------------
*    cmd ::= while <expr> do <cmd>
*-------------------------------------------------------------*/
cmd ::=
     tWHILE, expr^^E,
     tDO, cmd^^C
<:>
     (typecheck(State, Type) ::-
          E^^typecheck(State, TypeE),
          C^^typecheck(State, TypeC),
          (    TypeE = bool, TypeC = type_ok, Type = type_ok
          ;    Type = type_wrong
          )
```

```
    ),
    (usecheck(UEnv, PreUStore, PostUStore) ::-
        E^^usecheck(UEnv, PreUStore, MidUStore, _),
        C^^usecheck(UEnv, MidUStore, PostUStore)
    ),
    (meaning(Env, PreStore, PostStore) ::-
        while(Env, PreStore, PostStore, E, C)
    ).


while(Env, PreStore, PostStore, E, C) :-
    copy_term(E, E1),
    E^^meaning(Env, PreStore, Val),
    (   Val = bTRUE,...', nl]),
        copy_term(C, C1),
        C^^meaning(Env, PreStore, MidStore),
        while(Env, MidStore, PostStore, E1, C1)
    ;
        Val = bFALSE,
        PostStore = PreStore
    ).



/*-------------------------------------------------------------
*    cmd ::= if <expr> then <cmd> else <cmd>
*-------------------------------------------------------------*/
cmd ::=
    tIF, expr^^E,
    tTHEN, cmd^^C1,
    tELSE, cmd^^C2
<:>
    (typecheck(State, Type) ::-
        E^^typecheck(State, TypeE),
        C1^^typecheck(State, TypeC1),
        C2^^typecheck(State, TypeC2),
        (   TypeE = bool, TypeC1 = type_ok,
            TypeC2 = type_ok, Type = type_ok
        ;
            Type = type_wrong
        )
    ),
```

```
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          E^^usecheck(UEnv, PreUStore, MidUStore, _),
          C1^^usecheck(UEnv, MidUStore, UStore1),
          C2^^usecheck(UEnv, MidUStore, UStore2),
          worseof(UStore1, UStore2, PostUStore)
     ),
     (meaning(Env, PreStore, PostStore) ::-
          E^^meaning(Env, PreStore, Val),
          (    Val = bTRUE,
               C1^^meaning(Env, PreStore, PostStore)
          ;
               Val = bFALSE,
               C2^^meaning(Env, PreStore, PostStore)
          )
     ).

worseof(Store1, Store2, StoreJoint) :-
     ...




/*-------------------------------------------------------------
*    procedure call
*    cmd ::= <id>
* -------------------------------------------------------------*/
cmd ::=
     tIDENT^^I
<:>
     (typecheck(State, Type) ::-
          I^^meaning(Id),
          lookup(State, Id, TypeI),
          (    TypeI = proc, Type = type_ok
          ;    Type = type_wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          I^^meaning(Id),
          lookup(UEnv, Id, [UC,DeclUEnv] ),
          UC^^usecheck(DeclUEnv, PreUStore, PostUStore)
     ),
```

```
     (meaning(Env, PreStore, PostStore) ::-
          I^^meaning(Id),
          lookup(Env, Id, [Cmd,DeclEnv] ),
          copy_term(Cmd, C),
          C^^meaning(DeclEnv, PreStore, PostStore)
     ).
```

```
/*-------------------------------------------------------------
*    cmd ::= input <id>
*-------------------------------------------------------------*/
cmd ::=
     tINPUT, tIDENT^^I
<:>
     (typecheck(State, Type) ::-
          I^^meaning(Id),
          lookup(State, Id, TypeI),
          (    TypeI = int, Type = type_ok
          ;    Type = type_wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          I^^meaning(Id),
          lookup(UEnv, Id, ULocn),
          lookup(PreUStore, ULocn, Use),
          (    Use = bottom,
               update(PreUStore, PostUStore, ULocn, use_ok)
          ;
               PostUStore = PreUStore
          )
     ),
     (meaning(Env, PreStore, PostStore) ::-
          I^^meaning(Id),
          writel(['<<< input:  ', Id, '.  <<< :  ']),
          read(Value),
          lookup(Env, Id, Locn),
          update(PreStore, PostStore, Locn, Value)
     ).
```

```
/*-------------------------------------------------------------
* cmd ::= output <expr>
*-------------------------------------------------------------*/
cmd ::=
     tOUTPUT, expr^^E
<:>
     (typecheck(State, Type) ::-
          E^^typecheck(State, TypeE),
          (    TypeE = int, Type = type_ok
          ;    Type = type_wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore) ::-
          E^^usecheck(UEnv, PreUStore, PostUStore, _)
     ),
     (meaning(Env, PreStore, PostStore) ::-
          E^^meaning(Env, PreStore, Value),
          PostStore = PreStore,
          writel(['>>> output >>> :  ', Value, nl])
     ).

/*=============================================================
*    DECLARATION LIST
*============================================================*/
/*-------------------------------------------------------------*
*    <declList> ::= <decl> <declCont>
*-------------------------------------------------------------*/
declList ::=
     decl^^D, declCont^^DL
<:>
     (typecheck(PreState, PostState) ::-
          D^^typecheck(PreState, MidState),
          DL^^typecheck(MidState, PostState)
     ),
     (usecheck(PreUEnv, PostUEnv) ::-
          D^^usecheck(PreUEnv, MidUEnv),
          DL^^usecheck(MidUEnv, PostUEnv)
     ),
     (meaning(PreEnv, PostEnv) ::-
          D^^meaning(PreEnv, MidEnv),
          DL^^meaning(MidEnv, PostEnv)
     ).
```

```
/*-------------------------------------------------------------*
*    <declCont> ::= ; <declList>
*-------------------------------------------------------------*/
declCont ::=
     tSEMICOLON, !,
     declList^^DL
<:>
     (typecheck(PreState, PostState) ::-
          DL^^typecheck(PreState, PostState)),
     (usecheck(PreUEnv, PostUEnv) ::-
          DL^^usecheck(PreUEnv, PostUEnv)),
     (meaning(PreEnv, PostEnv) ::-
          DL^^meaning(PreEnv, PostEnv)).

/*-------------------------------------------------------------*
*    <declCont> ::= []
*-------------------------------------------------------------*/
declCont ::=
     []
<:>
     (typecheck(State, State)),
     (usecheck(UEnv, UEnv)),
     (meaning(Env, Env)).

/*=============================================================
*    DECLARATIONS
*============================================================*/
/*-------------------------------------------------------------*
*    <decl> ::= var <id>
*-------------------------------------------------------------*/
decl ::=
     tVAR, tIDENT^^I
<:>
     (typecheck(PreState, PostState) ::-
          I^^meaning(Id),
          update(PreState, PostState, Id, int)
     ),
     (usecheck(PreUEnv, PostUEnv) ::-
          I^^meaning(Id),
          gensym(uloc, ULocn),
          update(PreUEnv, PostUEnv, Id, ULocn)
```

```
    ),
    (meaning(PreEnv, PostEnv) ::-
        I^^meaning(Id),
        gensym(loc, Locn),
        update(PreEnv, PostEnv, Id, Locn)
    ).


/*-------------------------------------------------------------*
*    <decl> ::= proc <id> = <cmd>
*-------------------------------------------------------------*/
decl ::=
    tPROC, tIDENT^^I, tIS, cmd^^C
<:>
    (typecheck(PreState, PostState) ::-
        I^^meaning(Id),
        C^^typecheck(PreState, Type),
        (    Type = type_ok,
             update(PreState, PostState, Id, proc)
        ;
             Type = type_wrong,
             update(PreState, PostState, Id, wrong)
        )
    ),
    (usecheck(PreUEnv, PostUEnv) ::-
        I^^meaning(Id),
        update(PreUEnv, PostUEnv, Id, [C,PreUEnv])
    ),
    (meaning(PreEnv, PostEnv) ::-
        I^^meaning(Id),
        update(PreEnv, PostEnv, Id, [C,PreEnv])
    ).


/*=============================================================
*    EXPRESSIONS
*=============================================================*/
/*-------------------------------------------------------------
*    <expr> ::= <const>
*-------------------------------------------------------------*/
expr ::= tCONST^^X
<:>
```

```
     (typecheck(_, Type) ::-
          X^^meaning(Value),
          (    (Value = bTRUE ; Value = bFALSE), Type = bool
          ;    Type = int
          )
     ),
     (usecheck(_, _, _, use_ok)
     ),
     (meaning(_, _, Value) ::-
          X^^meaning(Value)
     ).


/*-------------------------------------------------------------
*    <expr> ::= <id>
*-------------------------------------------------------------*/
expr ::= tIDENT^^I
<:>
     (typecheck(State, Type) ::-
          I^^meaning(Id),
          lookup(State, Id, TypeId),
          (    TypeId = int, Type = int
          ;    Type = wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore, Use) ::-
          I^^meaning(Id),
          lookup(UEnv, Id, ULocn),
          lookup(PreUStore, ULocn, Use1),
          (    Use1 = bottom, Use = bad,
               update(PreUStore, PostUStore, ULocn, bad),
          ;
               Use = Use1,
               PostUStore = PostUStore
          )
     ),
     (meaning(Env, Store, Value) ::-
          I^^meaning(Id),
          lookup(Env, Id, Locn),
          lookup(Store, Locn, Value)
     ).
```

```
/*-------------------------------------------------------------
*    <expr> ::= <unyop> <expr>
*-------------------------------------------------------------*/
expr ::= tUNYOP^^O, expr^^E
<:>
     (typecheck(State, Type) ::-
          E^^typecheck(State, TypeE),
          O^^meaning(Op),
          (    Op = sub, TypeE = int, Type = int
          ;    Op = tNOT, TypeE = bool, Type = bool
          ;    Type = wrong
          )
     ),
     (usecheck(UEnv, PreUStore, PostUStore, Use) ::-
          E^^usecheck(UEnv, PreUStore, PostUStore, Use)
     ),
     (meaning(Env, Store, Value) ::-
          E^^meaning(Env, Store, Value1),
          O^^meaning(Op),
          eval(Op, Value1, Value)
     ).


/*-------------------------------------------------------------
*    <expr> ::= ( <expr> <binop> <expr> )
*-------------------------------------------------------------*/
expr ::= tLPAREN, expr^^E1, tBINOP^^O, expr^^E2, tRPAREN
<:>
     (typecheck(State, Type) ::-
          E1^^typecheck(State, Type1),
          E2^^typecheck(State, Type2),
          O^^meaning(Op),
          (    (Op = add ; Op = sub),
               Type1 = int, Type2 = int, Type = int
          ;
               (Op = '>' ; Op = '<' ; Op = '=' ; Op = '>=' ;
                                   Op = '<=' ; Op = ' =='),
               Type1 = int, Type2 = int,
               Type = bool
          ;
               (Op = and ; Op = or),
               Type1 = bool, Type2 = bool, Type = bool
```

```
        ;
            Type = wrong
        )
    ),
    (usecheck(UEnv, PreUStore, PostUStore, Use) ::-
        E1^^usecheck(UEnv, PreUStore, UStore1, Use1),
        E2^^usecheck(UEnv, UStore1, PostUStore, Use2),
        (   Use1 = use_ok, Use2 = use_ok, Use = use_ok
        ;   Use = use_bad
        )
    ),
    (meaning(Env, Store, Value) ::-
        E1^^meaning(Env, Store, Value1),
        E2^^meaning(Env, Store, Value2),
        O^^meaning(Op),
        eval(Op, Value1, Value2, Value)
    ).
```

Received June 1990
Accepted in a revised form in September 1990 *by D. Simpson*