SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(15), 1379–1416 (1999)

LETOS – A Lightweight Execution Tool for Operational Semantics

PIETER H. HARTEL*
Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton S017 1BJ, UK
(email: phh@ecs.soton.ac.uk)

SUMMARY

A lightweight tool is proposed to aid in the development of operational semantics. To use LETOS an operational semantics must be expressed in its meta-language, which itself is a superset of Miranda. The LETOS compiler is smaller than comparable tools, yet LETOS is powerful enough to support publication quality rendering using LaTeX, fast enough to provide competitive execution using Haskell, and versatile enough to support browsing of execution traces using Netscape. LETOS can be characterised as an experiment in 'creative laziness', showing how far one can get by gluing existing components together. The major specifications built using LETOS to-date are a smart card version of the Java Virtual Machine, a deterministic version of the π-calculus, and an electronic payment protocol. In addition, we have specified the semantics of many small programming languages and systems, totaling over 9000 lines of formal text. LETOS is unique in that it helps to check that a specification is operationally conservative. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: LETOS; operational semantics

1. INTRODUCTION

To design and specify the semantics of a programming language involves a number of clerical tasks, such as rendering (pretty printing the semantic specification), type-checking (checking that all objects in the specification have a unique type), execution (running sample programs according to the semantics), and execution tracing (producing derivation trees to study the details of executions). Tools help to perform these clerical tasks quickly and accurately, and give the designer increased confidence in the validity of the specification.

The tools that are currently available to assist the practitioner of semantics exhibit considerable variation in their sophistication. On the one hand, powerful systems such as Centaur [1] and ASF+SDF [2] provide parsing, rendering, type-checking, execution, tracing and more. On the other hand, some practitioners use a general purpose programming language to complement, or even as a substitute for, the mathematical notation normally used to specify a semantics. Textbook examples of such approaches include Nielson and Nielson [3], who use Miranda [4]† to implement their semantic specifications, and Stepney [5], who uses Prolog to

*Correspondence to: Pieter H. Hartel, Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton S017 1BJ, UK.
†Miranda is a trademark of Research Software Ltd.

CCC 0038–0644/99/151379–38$17.50

Received 29 January 1999
Revised 7 September 1999
Accepted 7 September 1999

implement her semantics. The RML system [6] covers middle ground in that RML offers only one facility: the compilation of operational semantics specifications into C. Concentrating on one aspect pays off; the RML system executes an operational semantics faster than its competitors.

The three approaches above represent as many points in a spectrum of possibilities. The sophisticated tools provide comprehensive facilities, but not without imposing limitations. The ASF+SDF system, for example, is not polymorphic, and it is also first order. This makes it awkward to work with denotational semantics [7]. The complexity of a system such as ASF+SDF makes it less than straightforward to experiment with, say, a higher order version of the system. The Typol subsystem [8] of Centaur is slow [6]. Recent work on the Minotaur [9] version of Typol has shown that imposing restrictions on the full generality of the Typol inference rules makes speed improvements of a factor of about 10–15 possible. This still leaves the performance of Typol wanting. The RML system does well on one aspect: speed of execution. However, the lack of rendering and execution tracing facilities does not make for a user-friendly system [6].

The systems referred to above have one factor in common: they are not small. For example, in 1988 the Centaur system was reported to consist of 32,000 lines of code [10] and in 1996 the RML compiler and runtime system together comprised 15,000 lines of code [11].

The approach advocated in this paper is to build a lightweight tool that offers the most important facilities at a minimal cost. This implies a small size of the meta-language compiler and minimal runtime support, a quick edit-compile-run cycle and a short learning period. The low cost is achieved by making maximum use of existing components, and by judiciously selecting essential features. The proposed LETOS program is small (2000 lines of lex, yacc and C). It takes as input a superset of Miranda. LETOS is capable of producing a proper declarative program, which can be type-checked and executed by the appropriate language system. Miranda and Haskell are used in the paper, but using Prolog would not pose difficulties. LETOS can also produce a LaTeX script, which when typeset provides the conventional rendering of a semantics. Execution tracing is supported by the ability to produce HTML files representing derivation trees. Small derivations can be rendered nicely by Netscape.

In the literature, several *formats* [12] have been proposed that impose a number of syntactic constraints on the axioms and inference rules defining an operational semantics. These formats can be shown to endow an operational semantics with one or more useful properties, such as operational conservativity [13]. A LETOS specification is an operational semantics that is machine readable. LETOS can thus be used as a vehicle for implementing a checker of the various formats. The current version of LETOS warns if an axiom or inference rule is not *source dependent* [13], (Section 4). This is a necessary condition for operational conservativity.

The paper makes three contributions. First, we show that it is possible to build a useful tool with simple means and a number of sensible engineering choices. Secondly, we demonstrate how using the tool helps to gain insight in the ambiguities of a semantic specification. Thirdly, we show how LETOS helps to build modular specifications.

LETOS can be used for non-deterministic specifications with one restriction: execution will only deliver one out of many possible results. All other features of LETOS, such as type-checking, source dependency checking and rendering, are available for deterministic as well as non-deterministic specifications.

The next section introduces the LETOS architecture. Section 3 describes the input format, using as a running example, the language While [3]. Section 4 presents the semantics of the

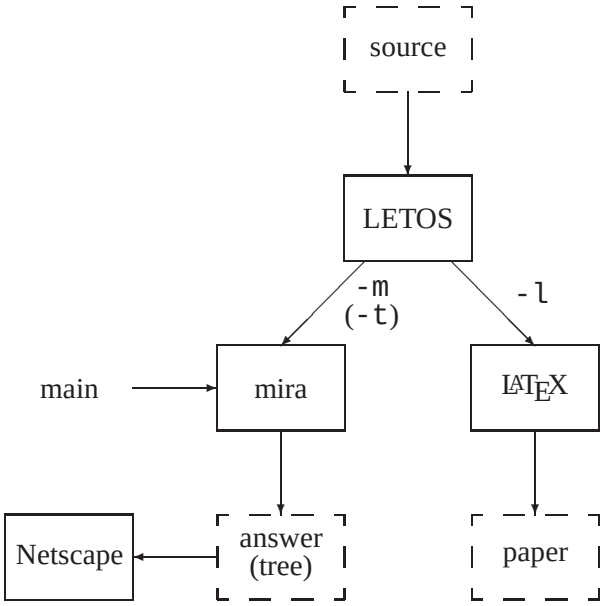


Figure 1. The LETOS *architecture*

LETOS meta-language through a translation from axioms and inference rules to functions. An assessment of the functionality and the performance of the tool is given in Section 5. The limitations are discussed in Section 6. Related work is discussed in Section 7. The last section presents the conclusions.

2. THE LETOS ARCHITECTURE

LETOS links three standard tools (LaTeX, Netscape and Miranda). Figure 1 shows how these components are connected. LETOS has the following properties:

- The LETOS source is a literate script. Sections of the input between delimiters `.MS` and `.ME` are formal text, anything else is LaTeX source.
- The formal text in the input to the program represents a specification written in a superset of Miranda.
- LETOS is capable of translating the specification into LaTeX, such that a publication quality document of the specification can be obtained (`-l` command line option). This paper is an example of the output of the program.
- LETOS is capable of generating a Miranda program for the formal text in its input (`-m` command line option). When the program is compiled, the Miranda system performs strong, polymorphic type checking.
- When executed, the Miranda program generated can be supplied with appropriate input and deliver a result.
- Along with the result, a derivation tree can be generated to show how the behaviour arose (`-t` option).
- The derivation tree is shown in a form that is under the control of the semantic

specification and is not prescribed by LETOS. As an example, we show how a web page is created, which can be browsed by Netscape.

3. THE LETOS INPUT LANGUAGE

The LETOS input language is basically Miranda, augmented with the following constructs:

- A reasonably general notation for expressing an abstract syntax as an algebraic data-type.
- A notation for expressing relations in terms of axioms and inference rules.
- Basic set theoretic expressions, let expressions and conditionals.
- A number of further, minor features such as the provision of various kinds of brackets, and ways of generating arbitrary LaTeX symbols.

LETOS has a limited notion of the semantics of its input language. It 'understands' and translates the extensions listed above into the target language. The tool relies mostly on the target to provide a semantics for the language elements that the target and LETOS have in common. This represents a significant difference between LETOS and other tools, which generally fix the semantics of the entire specification language. On the one hand, this is a disadvantage in that error messages about LETOS specifications are given in terms of the generated LaTeX or Miranda. These errors message are sometimes difficult to relate back to the LETOS input. On the other hand, the LETOS user will have the skills to interpret the error messages as they would also appear during the use of LaTeX or Miranda on their own.

In the sequel, when the distinction is immaterial, we refer to axioms and inference rules as 'rules'.

The best way of introducing the extensions is by discussing a typical example of a language and its semantics: the natural semantics of the language While [3]. This is the subject of the following (sub)sections.

3.1. Abstract syntax

An abstract syntax is represented as a type. Below, the abstract syntax is given of While numerals (n), identifiers (x), arithmetic expressions (a), boolean expressions (b) and statements (S). The example shows how numerals (n) are represented by the primitive Miranda type for numbers. Similarly, an identifier (x) is represented as a string (i.e. a list of characters). It is worth noting that Miranda's primitive number type provides unbounded precision integer arithmetic.

```
n ≡ num;
x ≡ string;
a ≡ n | x | a + a | a * a | a − a;
b ≡ true | false | a = a | a ≤ a | ¬ b | b ∧ b;
S ≡ x := a | skip | S ; S |
    if b then S else S |
    while b do S;
```

The differences between the rendering above and that of Nielson and Nielson [3] are as follows. First, the representation above identifies meta-variables ranging over syntactic categories with the syntactic categories themselves. Separate syntactic categories could be introduced as type synonyms. Secondly, Nielson and Nielson use subscripts on recursive occurrences of a, b and S. The subscripts are not used, and can thus be omitted.

LETOS *input*

In the literate programming convention of LETOS, the abstract syntax is input as a set of algebraic data type declarations. We follow Miranda, in that a prefix binary constructor may be preceded by a dollar sign to turn the prefix constructor into an infix constructor.

```
.MS
n == num ;
x == string ;

a ::= ‘N n | ‘V x |
      a $Add a | a $Mul a | a $Sub a ;

b ::= Btrue | Bfalse | a $Eq a | a $Le a |
      Neg  b | b $And b ;

\stm ::= x $Ass a | Skip | \stm $Comp \stm |
         If b Then \stm Else \stm |
         While b Do \stm ;
.ME
```

The next two sections discuss the details of the LETOS input shown above as part of the presentation of the LaTeX and Miranda output that is generated.

LaTeX *output*

For the purpose of rendering, the algebraic data type definitions shown above should be embedded in a LaTeX array-environment below. The array-environment should have three columns: one for the left-hand side of the definitions, one for the symbols `==` or `::=` and one for the right-hand side of the definitions. The line `\begin{array}{...}` is part of the LETOS input, and can therefore be varied, for example to centre the middle column.

```
\newcommand{\Add}{+}
\newcommand{\stm}{\mathsf{S}}
\begin{array}{@{}lll}
.MS
...
.ME
\end{array}
```

LETOS offers two simple ‘editing’ facilities. First, the backquote prefix (`‘`) causes symbols (`‘N` and `‘V` in the example) to be invisible in the generated LaTeX. This conforms to the common mathematical practice of omitting injection and projection functions when working with disjoint sums (an algebraic data type is a disjoint sum of products).

Like the Z tools ZTC [14] and PiZa [15], LETOS allows an identifier to begin with a backslash (`\`). This tells LaTeX to interpret the identifier as a macro call, so that an appropriate symbol may be used to render the identifier. In addition, LETOS automatically represents constructors as macro calls. The macros `\Add` and `\stm` show how this facility is used.

Miranda output

The LETOS declarations for the syntactic categories n, x and b are already valid Miranda. Dropping the prefixes (') yields valid Miranda for the definition of arithmetic expressions a. The \ prefix of \stm is replaced by the prefix macro. Finally, the type \stm uses dist-fix notation, which is not supported by Miranda. LETOS implements dist-fix constructions by generating a new type for each constructor except the first. The Miranda generated for the statement type is shown below. (The apostrophe ' is a legal character in a Miranda identifier.)

```
macro_stm ::=
          x $Ass a | Skip |
          macro_stm $Comp macro_stm |
          If b then' macro_stm else' macro_stm |
          While b do' macro_stm ;
do'   ::= Do ;
else' ::= Else ;
then' ::= Then ;
```

The combination of invisible symbols, macro prefixes and distfix-constructors makes it possible to render any conceivable abstract syntax. The cost of implementing these three facilities is small, as the effect of each is localised; that is no global information is required. This is a good example of providing an essential facility at minimal cost by making a sensible engineering choice.

3.1.1. Using the abstract syntax

With the above definitions of the abstract syntax of While, it is now possible to write the code for a sample statement in the While language. The factorial function is given below, taking y as argument, and delivering the result in z. The code is written as separate definitions to emphasise the structure.

$$\begin{aligned} \mathsf{fact} &= \mathsf{z} := 1 \,;\, \texttt{while}\ \neg\, \mathsf{y} = 1\ \texttt{do}\ \mathsf{body};\\ \mathsf{body} &= \mathsf{z} := \mathsf{z} \star \mathsf{y} \,;\, \mathsf{y} := \mathsf{y} - 1;\\ \mathsf{y} &= \texttt{"y"};\\ \mathsf{z} &= \texttt{"z"}; \end{aligned}$$

We omit the LETOS input or LaTeX output for this example, as it shows nothing new.

With the abstract syntax and a sample code fragment in place, it is now time to look at the semantics proper.

3.2. Semantic functions

The semantic function $\mathcal{A}$ below defines the value of arithmetic expressions (a). An expression may contain identifiers (x), so the semantic function must know about a mapping from identifiers to values. The definitions below give the type of the value domain $\mathbb{Z}$ and the type of the mapping state.

$$\begin{aligned} \mathbb{Z} &\equiv \mathsf{num};\\ \mathsf{state} &\equiv \{\langle \mathsf{x} \mapsto \mathbb{Z} \rangle\}; \end{aligned}$$

A sample state that specifies an initial value of 3 for the variable y is:

$$\begin{aligned} &s_3 :: \mathsf{state};\\ &s_3 = \{\langle y \mapsto 3\rangle\}; \end{aligned}$$

The semantic function $\mathcal{A}$ performs case analysis on the components of the abstract syntax and defines the value of an arithmetic expression thus:

$$\begin{aligned} \mathcal{A} &:: a \rightarrow \mathsf{state} \rightarrow \mathbb{Z};\\ \mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!];\\ \mathcal{A}[\![x]\!]s &= s(x);\\ \mathcal{A}[\![a_1 + a_2]\!]s &= \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s;\\ \mathcal{A}[\![a_1 \star a_2]\!]s &= \mathcal{A}[\![a_1]\!]s * \mathcal{A}[\![a_2]\!]s;\\ \mathcal{A}[\![a_1 - a_2]\!]s &= \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s; \end{aligned}$$

There is a fundamental difference between an executable specification as given above and a mathematical specification as given by Nielson and Nielson [3]. In mathematics, it is possible to explicitly reason about the partiality of $\mathcal{A}$. This would arise if an expression contains variables that are not represented in the state. For example:

$$\mathcal{A}[\![y]\!]\{\} = \bot;$$

In an executable specification, execution would be terminated when an identifier cannot be mapped onto a value. Another instance of the same difference may be found in Section 3.3.1.

LETOS *input*

The LETOS input for the range of the mapping `state` and the mapping itself are given as follows:

```
\newcommand{\bbZ}{\bbB{Z}}
.MS
\bbZ == num ;
state == { <x|->\bbZ> } ;
.ME
```

The input for the second clause of the function $\mathcal{A}$ is representative for the LETOS notation employed to specify functions:

```
\calA [[‘V x]] s = s{/x/} ;
```

In addition to some minor differences (dealing with the \ prefix and the emphatic brackets), the clause uses the expression `s{/x/}`. This indicates that `s` is an association set, in which `x` should be looked up.

Miranda output

In the Miranda output, the sets used in the LETOS input notation are represented by lists without duplicates. The Miranda output for the definitions of $\mathbb{Z}$ and state are:

```
macro_bbZ == num ;
state     == [(x,macro_bbZ)] ;
```

The Miranda code for $\mathcal{A}$ should be consistent with this use of lists. For example, the code for the second clause is shown below. A `lookup` function is used to search the association list (LETOS has replaced the emphatic brackets by parentheses):

```
macro_calA (V x) s = lookup s x ;
```

The definition of the `lookup` function as well as a number of other utility functions are provided with LETOS as a separate module. This provides for flexibility as the functions may then be redefined, for example to improve their efficiency.

3.2.1. Boolean expressions

The semantic function $\mathcal{B}$ for boolean expressions (b) is not reproduced here, because $\mathcal{B}$ requires no new features. We just give the type of the function:

$$\mathcal{B} :: \mathsf{b} \rightarrow \mathsf{state} \rightarrow \mathbb{B};$$

The boolean type ($\mathbb{B}$) and the relevant constants as used by the semantic function are:

$$\begin{aligned} \mathbb{B} &\equiv \mathsf{bool};\\ \mathsf{tt} &= \mathsf{True};\\ \mathsf{ff} &= \mathsf{False}; \end{aligned}$$

As was done before, for identifiers and numerals, we rely on a built-in data type of Miranda to provide booleans.

3.3. Natural semantics of While

An operational semantics is usually represented by a set of rules. The natural semantics of the statements S of the language While is shown below. On the left of each arrow we find a configuration, which contains a pair of statement and state. The right-hand side is just a state.

$$[\mathsf{ass}_{\mathsf{ns}}] \quad \langle \mathsf{x} := \mathsf{a},\ \mathsf{s}\rangle \xrightarrow{1} \mathsf{s} \oplus \{\mathsf{x} \mapsto \mathcal{A}[\![\mathsf{a}]\!]\mathsf{s}\};$$

$$[\mathsf{skip}_{\mathsf{ns}}] \quad \langle \mathtt{skip},\ \mathsf{s}\rangle \xrightarrow{1} \mathsf{s};$$

$$[\mathsf{comp}_{\mathsf{ns}}] \quad \frac{\langle \mathsf{S}_1,\ \mathsf{s}\rangle \xrightarrow{1} \mathsf{s}',\quad \langle \mathsf{S}_2,\ \mathsf{s}'\rangle \xrightarrow{1} \mathsf{s}''}{\langle \mathsf{S}_1\ ;\ \mathsf{S}_2,\ \mathsf{s}\rangle \xrightarrow{1} \mathsf{s}'';}$$

$$[\mathsf{if}^{\mathsf{tt}}_{\mathsf{ns}}] \quad \frac{\langle \mathsf{S}_1,\ \mathsf{s}\rangle \xrightarrow{1} \mathsf{s}'}{\langle \mathtt{if}\ \mathsf{b}\ \mathtt{then}\ \mathsf{S}_1\ \mathtt{else}\ \mathsf{S}_2,\ \mathsf{s}\rangle \xrightarrow{1} \mathsf{s}',\ \mathbf{if}\ \mathcal{B}[\![\mathsf{b}]\!]\mathsf{s} = \mathsf{tt};}$$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \quad \frac{\langle S_2,\ s\rangle \xrightarrow{1} s'}{\langle \texttt{if b then } S_1 \texttt{ else } S_2,\ s\rangle \xrightarrow{1} s', \quad \textbf{if } \mathcal{B}[\![b]\!]s = \text{ff};}$$

$$[\text{while}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \xrightarrow{1} s', \quad \langle \texttt{while b do } S_1,\ s'\rangle \xrightarrow{1} s''}{\langle \texttt{while b do } S_1,\ s\rangle \xrightarrow{1} s'', \quad \textbf{if } \mathcal{B}[\![b]\!]s = \text{tt};}$$

$$[\text{while}^{\text{ff}}_{\text{ns}}] \quad \langle \texttt{while b do } S_1,\ s\rangle \xrightarrow{1} s, \quad \textbf{if } \mathcal{B}[\![b]\!]s = \text{ff};$$

There are two differences between the representation above and that of Nielson and Nielson [3]. First, the notation above for substitution using curly brackets makes it explicit that a mapping is a set. This particular notation is not standard in the programming language semantics, but it is borrowed from Z.

Secondly, the arrows have been labelled, to be able to distinguish the present relation from other relations that will appear in later sections. This is important because relations have a type:

$$\xrightarrow{1} :: (S,\ \text{state}) \leftrightarrow \text{state};$$

The idea of making the type of a relation unambiguous makes it possible to type check specifications mechanically. Labelling relations also helps to better understand the semantics, as we shall see in Section 3.4.

LETOS *input*

The LETOS input language allows rules to be defined at the top level, together with data type and function definitions. The source for the ass_{ns} axiom is:

```
axiom ass_ns =
|- <x $Ass a, s> =1=> s{/x|->\calA[[a]]s/};
```

The name of the axiom, ass_{ns}, is used for execution tracing purposes (see below). Label (1) on the arrow is used to identify a particular relation. The notation `s{/x|->.../}` denotes substitution.

The LETOS specification for the composition rule is:

```
rule comp_ns =
|- <\stm_1, s> =1=> s',
|- <\stm_2, s'> =1=> s''
-----------------------------------------
|- <\stm_1 $Comp \stm_2, s> =1=> s'' ;
```

As usual, the premises are written above the dashed line, and the conclusion is written below the line.

Some rules have a side condition. In the LETOS language this is indicated as follows:


```
axiom while_nsˆff =
|- <While b Do \stm_1, s> =1=> s,
if \calB [[b]] s = ff ;
```

A rule with a side condition **if** E_b can always be replaced by a rule with one (one more) premise(s). The new premise would then be of the form $E_b \stackrel{id}{\rightarrow} \mathsf{True}$. This will be elaborated in Section 4.

Miranda output

In the Miranda output, a transition system with a label (`l` say) is represented by a set of functional clauses with the name `rule_l`. For each rule a clause is generated that checks the assumptions, the premises (for rules) and the side conditions. When all these checks are successful, a list of states is produced as output.

For symmetry reasons, input is encoded also as a singleton list containing the current configuration. An empty list signals that none of the rules in the transition system apply, otherwise a singleton list results. For example, the generated Miranda for the ass_{ns} axiom above is:

```
rule_1 :: [(macro_stm,state)]->[state] ;
rule_1 [(x $Ass a,s)]
 = [substitute s (x,macro_calA a s)] ;
```

The translation of an inference rule into a function must take the premises into account. Consider the functions generated for the inference rule comp_{ns}. The recursive calls to `rule_1` under the `where` expression below represent the two premises. The guard on the clause checks that the recursive calls do indeed deliver at least one 'success' each, by making sure that the relevant lists are non-empty. The last success in the lists returned by the premises is the chosen value for further computation.

```
rule_1 [(macro_stm_1 $Comp macro_stm_2,s)]
 = [s''],if non_empty t_1 /\ non_empty t_2
   where
   s'  = last t_1 ;
   t_1 = rule_1 [(macro_stm_1,s)] ;
   s'' = last t_2 ;
   t_2 = rule_1 [(macro_stm_2,s')] ;
   ;
```

At the end of all function clauses generated for a labelled transition system, a default clause is appended:

```
rule_1 x = [] ;
```

This default case would apply when all other clauses fail (functional clauses are tried top-down in Miranda). For example, in the natural semantics, omitting the rule for a particular statement would cause the function `rule_1` to fail on a program that contains such a statement.

The method for translating rules into functions will be discussed in more detail in Section 4. For now, we note that the rules in a LETOS specification must be ordered as the order of the generated functional clauses matters.

3.3.1. *Executing and tracing the natural semantics*

With the definition of a suitable abstract syntax, and the specification of the semantic functions and rules, all ingredients necessary to execute the specification for an example (the factorial) are now available. The semantics of a particular program is a function S_{ns}, which when given an initial state returns a list of final states. This will be specified here as follows, using the arrow as a postfix operator in our expression language:

$$S_{ns} :: S \rightarrow state \rightarrow [state];$$
$$S_{ns}[\![S]\!]s = (\langle S,\ s\rangle \xrightarrow{1});$$

Compare this to the mathematical specification of S_{ns} by Nielson and Nielson, which is as follows:

$$S_{ns}\,[\![S]\!]\,s = \begin{cases} s', & \textbf{if } \langle S, s\rangle \xrightarrow{1} s' \\ \bot, & \textbf{otherwise} \end{cases}$$

There is considerable difference between the mathematical and the executable specifications. The reason is that the latter cannot decide whether the derivation is finite (cf. the Halting problem). To acknowledge this, only the essential part of the mathematical specification has been retained in the executable representation. This is the part shown in the shaded areas above.

When applied to the sample factorial program and the sample state s_3 given earlier, the following equality can be proved:

$$S_{ns}\ fact\ s_3 = [\{\langle y \mapsto 1\rangle,\ \langle z \mapsto 6\rangle\}]$$

The equality shows that the behaviour of the program is to decrement the value associated with y from 3 to 1 and to return 3! as the value associated with the variable z.

Miranda output

The Miranda function `macro_sns` generated for S_{ns} relies on the function `rule_1` as shown below. The function `macro_sns` is usually invoked from the main expression supplied to the Miranda interpreter.

```
macro_sns :: macro_stm -> state -> [state] ;
macro_sns macro_stm s = rule_1 [(macro_stm, s)] ;
```

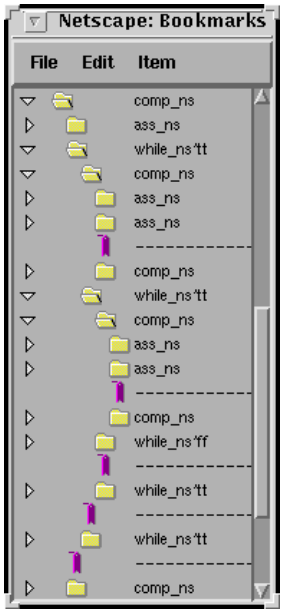
This concludes the presentation of abstract syntax, semantic functions, and rules for the natural semantics of the language While.

3.3.2. *Browsing a derivation tree*

The semantic functions and rules of the natural semantics can be used to prove that the above equality about fact is indeed true. It is technically not difficult to write down such a proof. However, to render the proof nicely as a derivation tree is laborious. The actual proof of the equality above is shown in Figure 2(a). The use of an axiom in the proof is shown in the tree as a single line with one transition. The name of the axiom is used as the label on the transition. When an inference rule is used, the premises are shown, each on a separate line,

$$\dfrac{\dfrac{}{[\mathrm{ass}_{ns}]\ \langle z := 1,\ s_3\rangle \xrightarrow{1} s_{31}} \qquad \dfrac{\dfrac{}{[\mathrm{ass}_{ns}]\ \langle z := (z \star y),\ s_{31}\rangle \xrightarrow{1} s_{33}} \qquad \dfrac{\dfrac{}{[\mathrm{ass}_{ns}]\ \langle y := (y-1),\ s_{33}\rangle \xrightarrow{1} s_{23}} \qquad [\mathrm{comp}_{ns}]\ \langle \mathtt{body},\ s_{31}\rangle \xrightarrow{1} s_{23};}{\dfrac{}{[\mathrm{ass}_{ns}]\ \langle z := (z \star y),\ s_{23}\rangle \xrightarrow{1} s_{26}} \qquad \dfrac{}{[\mathrm{ass}_{ns}]\ \langle y := (y-1),\ s_{26}\rangle \xrightarrow{1} s_{16}} \qquad [\mathrm{comp}_{ns}]\ \langle \mathtt{body},\ s_{23}\rangle \xrightarrow{1} s_{16};}}{\dfrac{}{[\mathrm{while}^{ff}_{ns}]\ \langle \mathtt{while}(\neg(y=1))\mathtt{do\ body},\ s_{16}\rangle \xrightarrow{1} s_{16}} \qquad [\mathrm{while}^{tt}_{ns}]\ \langle \mathtt{while}(\neg(y=1))\mathtt{do\ body},\ s_{23}\rangle \xrightarrow{1} s_{16};}}{[\mathrm{while}^{tt}_{ns}]\ \langle \mathtt{while}(\neg(y=1))\mathtt{do\ body},\ s_{31}\rangle \xrightarrow{1} s_{16};}}{[\mathrm{comp}_{ns}]\ \langle \mathtt{fact},\ s_3\rangle \xrightarrow{1} s_{16};}$$

(a) Conventional proof



(b) Sample LETOS rendering

Figure 2. Two ways of representing the proof associated with computing the factorial of 3. (a) Is isopmorphic to the folded items in (b). The notation s_{jk} *is an abbreviation for* $\{\langle y \mapsto j\rangle, \langle z \mapsto k\rangle\}$ *and* $s_3 = \{\langle y \mapsto 3\rangle\}$

followed by a horizontal line and then a conclusion. The conclusion is labelled by the name of the inference rule.

A proof such as that given in Figure 2(a) is read bottom up. At the bottom of the figure we see a transition labelled comp_{ns}. This is the conclusion of two sub-proofs, shown as sub-trees above the horizontal line at the bottom of the diagram. Both sub-trees are offset to the right. The first sub-tree occupies the first line of the figure. It corresponds to a single transition for the assignment $z := 1$. The second sub-tree occupies the remainder of the diagram. This second sub-tree serves to show that the transition labelled while^{tt}_{ns} at the bottom of the figure is valid.

Driven by a command line option (`-t`), LETOS is able to generate a Miranda program that calculates not just the final state, but also the entire derivation tree that was generated to create the final state. To illustrate this, the generated Miranda for the inference rule comp_{ns} is shown below, with tracing code added automatically. Remembering the derivation, inclusive of all intermediate states, involves a change of type for the function `rule_1` from producing a list of values of type `state` to producing a list of pairs: `(state,string)`.

```
rule_1 :: [(macro_stm,state)]->[(state,string)] ;
rule_1 [(macro_stm_1 $Comp macro_stm_2,s)]
 = [(s'',trace_1 "Comp_ns" ts)],
   if non_empty t_1/\non_empty t_2
   where
   ts  = [map snd t_1,map snd t_2] ;
   s'  = fst(last t_1) ;
```



```
t_1 = rule_1 [(macro_stm_1,s)] ;
s'' = fst(last t_2) ;
t_2 = rule_1 [(macro_stm_2,s')] ;
;
```

The function `trace_1` used above combines the lists of inputs received with the label of the current rule. A function such as this should be supplied as part of the LETOS input. The formatting of the derivation tree is entirely under control of the user, thus providing for maximal flexibility. It is straightforward to produce something like HTML in this way, but other formats such as LaTeX are also possible. Here is a fragment of the HTML that represents the derivation tree for the factorial program, when applied to the initial state s_3:

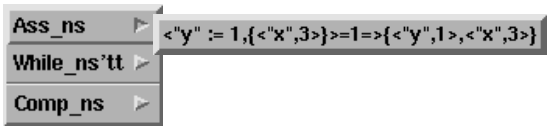
```
<DT><H3>comp_ns</H3>
<DL><p>
    <DT><H3 FOLDED>ass_ns</H3>
        ...
    <DT><H3>while_ns'tt</H3>
    <DL><p>
        <DT><H3>comp_ns</H3>
        <DL><p>
            <DT><H3 FOLDED>ass_ns</H3>
                ...
            <DT><H3 FOLDED>ass_ns</H3>
                ...
            <HR>
            <DT><H3 FOLDED>comp_ns</H3>
                ...
        </DL><p>
```

A convenient way of browsing a HTML derivation tree is by incorporating the tree in the file `bookmarks.html` so that Netscape can render the tree as a normal, quite verbose HTML page, and also in various compact forms. Unfortunately, Netscape does not render arbitrary HTML files in compact forms. The compact representation of the derivation from Figure 2(a) is shown in Figure 2(b) by the bookmarks window. Fine control over the amount of information is provided by clicking on the appropriate triangles, which fold and unfold sub trees.

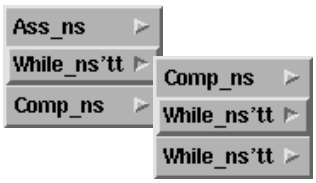
Netscape provides an alternative to browsing the derivation tree by using pull-down menus. Below is the menu corresponding to the conclusion of the derivation. It represents an instance of the $\mathsf{comp}_{\mathsf{ns}}$ rule. The two derivations to be made for the premises are shown above the line. The name of the rule for the root of the derivation tree is given below the line.

Moving the mouse to the field marked `Ass_ns` displays the derivation subtree associated with the first premise, which represents an instance of the $\mathsf{ass}_{\mathsf{ns}}$ axiom. It shows the effect of executing the first assignment statement on the state of the computation. The screen shot

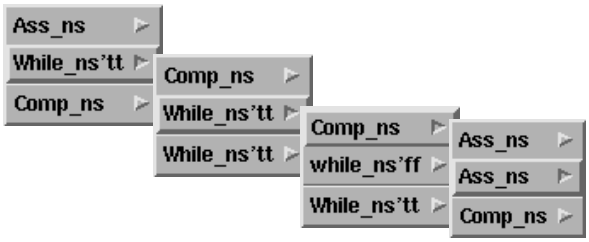
below was actually produced by a slightly more elaborate version of the `trace_1` function, which has access to the configurations, in addition to having access to the label of a rule.



To explore the depth of the derivation tree, move the mouse to the field marked `While_ns'tt`. This displays the root of the derivation subtree for the second premise.



Following the path further down in the tree yields the configuration below. It is now possible to view further assignments, but this will not be done here.



These two compact ways of rendering provide for a flexible and powerful way of browsing derivation trees. Using existing notation and tools (Netscape and HTML), should make the method easy to learn and use, and thus lightweight. Probably inherent to graphical representations of derivation trees, the present method does not scale up to large derivation trees. It does, however, permit browsing trees larger than those typically found in the literature due to the folding properties of the browser.

3.4. Structured operational semantics of While

Like the natural semantics, a Structured Operational Semantics (SOS) is represented by a set of rules. The SOS of the language While, following Nielson and Nielson, is shown below:

$$[\mathsf{ass}_{\mathsf{sos}}]\quad \langle \mathsf{x} := \mathsf{a},\ \mathsf{s}\rangle \overset{2}{\Rightarrow} \mathsf{s} \oplus \{\mathsf{x} \mapsto \mathcal{A}[\![\mathsf{a}]\!]\mathsf{s}\};$$

$$[\mathsf{skip}_{\mathsf{sos}}]\quad \langle \mathtt{skip},\ \mathsf{s}\rangle \overset{2}{\Rightarrow} \mathsf{s};$$

$$[\mathsf{comp}^1_{\mathsf{sos}}]\quad \frac{\langle \mathsf{S}_1,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \mathsf{S}_1',\ \mathsf{s}'\rangle}{\langle \mathsf{S}_1\ ;\ \mathsf{S}_2,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \mathsf{S}_1'\ ;\ \mathsf{S}_2,\ \mathsf{s}'\rangle};$$

$$[\mathsf{comp}^2_{\mathsf{sos}}]\quad \frac{\langle \mathsf{S}_1,\ \mathsf{s}\rangle \overset{2}{\Rightarrow} \mathsf{s}'}{\langle \mathsf{S}_1\ ;\ \mathsf{S}_2,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \mathsf{S}_2,\ \mathsf{s}'\rangle};$$

$$[\mathsf{if}^{\mathsf{tt}}_{\mathsf{sos}}]\quad \langle \texttt{if}\ \mathsf{b}\ \texttt{then}\ \mathsf{S}_1\ \texttt{else}\ \mathsf{S}_2,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \mathsf{S}_1,\ \mathsf{s}\rangle,\quad \textbf{if}\ \mathcal{B}[\![\mathsf{b}]\!]\mathsf{s} = \mathsf{tt};$$

$$[\mathsf{if}^{\mathsf{ff}}_{\mathsf{sos}}]\quad \langle \texttt{if}\ \mathsf{b}\ \texttt{then}\ \mathsf{S}_1\ \texttt{else}\ \mathsf{S}_2,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \mathsf{S}_2,\ \mathsf{s}\rangle,\quad \textbf{if}\ \mathcal{B}[\![\mathsf{b}]\!]\mathsf{s} = \mathsf{ff};$$

$$[\mathsf{while}_{\mathsf{sos}}]\quad \langle \texttt{while}\ \mathsf{b}\ \texttt{do}\ \mathsf{S}_1,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \texttt{if}\ \mathsf{b}\ \texttt{then}\ \mathsf{S}_1\ ;\ \texttt{while}\ \mathsf{b}\ \texttt{do}\ \mathsf{S}_1\ \texttt{else}\ \texttt{skip},\ \mathsf{s}\rangle;$$

This specification consists of two separate relations:

$$\overset{2}{\Rightarrow} :: (\mathsf{S},\ \mathsf{state}) \leftrightarrow \mathsf{state};$$
$$\overset{3}{\Rightarrow} :: (\mathsf{S},\ \mathsf{state}) \leftrightarrow (\mathsf{S},\ \mathsf{state});$$

These relations are different because their types are different. In their specifications, Nielson and Nielson [3] ignore the difference, but introduce it again when the execution of the specification in Miranda is discussed [3]. To be able to type check the specification, it is necessary to make the distinction, and labelled relations provide a tidy way of doing so.

An SOS yields a derivation sequence, whereas a natural semantics delivers a derivation tree. To put the two on a level playing field, it is convenient to add another inference rule to the SOS specification. This new relation $\overset{4}{\Rightarrow}$ given below first computes the transitive closure of the relation $\overset{3}{\Rightarrow}$, and then selects final state:

$$\overset{4}{\Rightarrow} :: (\mathsf{S},\ \mathsf{state}) \leftrightarrow \mathsf{state};$$
$$[\mathsf{run}_{\mathsf{sos}}]\quad \frac{\langle \mathsf{S}_1,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} *\langle \mathsf{S}_1',\ \mathsf{s}'\rangle,\quad \langle \mathsf{S}_1',\ \mathsf{s}'\rangle \overset{2}{\Rightarrow} \mathsf{s}''}{\langle \mathsf{S}_1,\ \mathsf{s}\rangle \overset{4}{\Rightarrow} \mathsf{s}''};$$

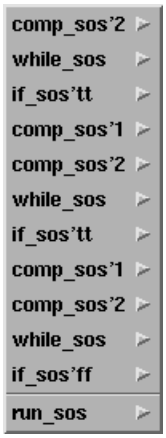
With this preparation, the SOS of While statements can be expressed in the same way as the natural semantics:

$$\mathsf{S}_{\mathsf{sos}} :: \mathsf{S} \rightarrow \mathsf{state} \rightarrow [\mathsf{state}];$$
$$\mathsf{S}_{\mathsf{sos}}[\![\mathsf{S}]\!]\mathsf{s} = (\langle \mathsf{S},\ \mathsf{s}\rangle \overset{4}{\Rightarrow});$$

The translation of the SOS into Miranda and LaTeX is straightforward.

3.4.1. Executing and tracing the SOS

The introduction of the extra inference rule $\mathsf{run}_{\mathsf{sos}}$ makes it possible to render the derivation sequence for the factorial example in the same way as the derivation tree for the natural semantics was rendered. Here is the menu-style result:



This shows that at the root of the 'derivation tree', and as a result of using the $*$ operator, 11 steps have to be taken. The steps that represent the inference rules (`comp_sos'1` and `comp_sos'2`) have subsidiary derivation trees; others will exhibit just a transition on the configuration.

3.5. Alternative approaches

Most books present abstract syntax, axioms and inference rules in the same way as we have done above [3, 16–18]. However, some texts [19] leave out embellishments, and represent an abstract syntax purely as an algebraic data type. LETOS supports both approaches. For example, the LETOS input of the pure abstract syntax of While could be represented as shown below.

```
n  == num ;
x  == string ;

a ::= N n | V x |
      Add a a | Mul a a | Sub a a ;

b ::= Btrue | Bfalse | Eq a a | Le a a |
      Neg b | And b b ;

s ::= Ass x a | Skip | Comp s s |
      If b s s | While b s ;
```

The advantage of using the a pure abstract syntax is that the clutter of the special symbols like `$`, `'` or `\` has disappeared. The disadvantage is that functions and rules defined over a pure abstract syntax cannot be presented in textbook style. Compare the axiom $\mathsf{while}_{\mathsf{sos}}$ from the

previous section with the version using the pure abstract syntax below. The lack of keywords makes the pure version less readable.

$$[\mathsf{while}_{\mathsf{sos}}]\ \langle \texttt{while}\ \mathsf{b}\ \mathsf{S}_1,\ \mathsf{s}\rangle \overset{3}{\Rightarrow} \langle \texttt{if}\ \mathsf{b}(\texttt{comp}\ \mathsf{S}_1(\texttt{while}\ \mathsf{b}\ \mathsf{S}_1))\texttt{skip},\ \mathsf{s}\rangle;$$

Tools such as RML [20], and Centaur/Typol [1] work with a pure abstract syntax. For example, here is a fragment of the Metal specification of arithmetic expressions taken from the Centaur Tutorial [21]. It corresponds exactly to the pure abstract syntax of a above:

```
definition of Exp is
  abstract syntax
    integer -> implemented as INTEGER;
    variable -> implemented as IDENTIFIER;
    add -> EXP EXP;
    mul -> EXP EXP;
    sub -> EXP EXP;
    EXP ::= add mul sub integer variable;
end definition
```

Centaur offers facilities to specify a concrete syntax, the ability to parse input according to the concrete syntax whilst building terms in the abstract syntax, and pretty printing facilities to render the abstract syntax. LETOS does not offer these facilities, but we have shown that the LETOS rendering is of the same high standard as that used in textbooks.

The lack of parsing facilities makes inputting 'abstract syntax' inconvenient. An elegant way to add scanning and parsing facilities would be to use parser combinators [22]. This could be achieved without adding further facilities to LETOS; see, for example, our earlier work [23].

3.6. Abstract machines

Nielson and Nielson [3] describe a low level abstract machine and a 'provably correct implementation' of While. Both the abstract machine and the compiler have been expressed using LETOS and the associated standard tools. Here is the abstract syntax of the instructions of the machine:

$$\begin{aligned}\mathsf{i} \equiv\ & \mathsf{PUSH\ n} \mid \\ & \mathsf{ADD} \mid \mathsf{MULT} \mid \mathsf{SUB} \mid \\ & \mathsf{TRUE} \mid \mathsf{FALSE} \mid \\ & \mathsf{EQ} \mid \mathsf{LE} \mid \\ & \mathsf{AND} \mid \mathsf{NEG} \mid \\ & \mathsf{FETCH\ x} \mid \mathsf{STORE\ x} \mid \\ & \mathsf{NOOP} \mid \\ & \mathsf{BRANCH(c,\ c)} \mid \mathsf{LOOP(c,\ c)}; \\ \mathsf{c} \equiv\ & \epsilon \mid \mathsf{i} : \mathsf{c};\end{aligned}$$

The translation functions for arithmetic expressions (a), boolean expressions (b) and statements (S) do not pose new problems to LETOS. Below are just the types of the translation functions, where $\mathcal{CS}$ represents the compiler from the While language to the abstract machine

code. The compiler is easily written in the conventional recursion equation style using LETOS.

$$\mathcal{CA} :: \mathsf{a} \rightarrow \mathsf{c};$$
$$\mathcal{CB} :: \mathsf{b} \rightarrow \mathsf{c};$$
$$\mathcal{CS} :: \mathsf{S} \rightarrow \mathsf{c};$$

The operational semantics of the abstract machine defines a relation between configurations. This relation is conventionally indicated by the symbol $\triangleright$:

$$\overset{5}{\triangleright} :: (\mathsf{c},\ \mathsf{stack},\ \mathsf{state}) \leftrightarrow (\mathsf{c},\ \mathsf{stack},\ \mathsf{state});$$

The first component (c) of a configuration is a sequence of instructions; the second component (stack) contains boolean and integer values; the third component is the mapping from variables to values (state).

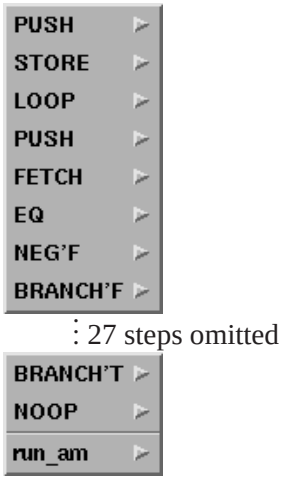
The relation $\overset{5}{\triangleright}$ defines the single step transitions. The full semantics of the abstract machine is given by a new relation $\overset{6}{\triangleright}$, which takes the transitive closure and selects the final state as before, for the SOS:

$$\overset{6}{\triangleright} :: (\mathsf{c},\ \mathsf{stack},\ \mathsf{state}) \leftrightarrow \mathsf{state};$$

The compiler $\mathcal{CS}$ and the abstract machine represented by the relation $\overset{6}{\triangleright}$ together provide for the third specification of the semantics of the While language:

$$\mathsf{S}_{\mathsf{am}} :: \mathsf{S} \rightarrow \mathsf{state} \rightarrow [\mathsf{state}];$$
$$\mathsf{S}_{\mathsf{am}}[\![\mathsf{S}]\!]\mathsf{s} = (\langle \mathcal{CS}[\![\mathsf{S}]\!],\ [],\ \mathsf{s}\rangle \overset{6}{\triangleright});$$

The derivation sequence produced by the computation for 3! on the abstract machine is long. Here are the beginning and the end, omitting 27 intermediate steps:



This time, the derivation sequence is absolutely flat, (cf. Gunter [24]). Moving the mouse to a sub-menu would show the actual transition step.

3.7. Denotational semantics

LETOS can be used to render a conventional denotational semantics. It also supports type checking and execution of a denotational semantics. Unfortunately, LETOS does not allow for derivation trees from denotational semantics to be generated and browsed. The reason is that the structure of each rule corresponds to the structure of a node in a derivation tree. In a denotational semantics, the semantic function is not defined using rules, but as a system of recursion equations. It is possible for a tool to discover where in these equations recursive calls are made to the relevant semantic functions. It is difficult, but presumably not impossible, to decide how such calls should be modified automatically to allow for tracing information to be gathered.

However, even without the tracing and browsing facility, it is illustrative of the pretty printing capabilities of the tool to show the denotational semantics of the While language [3]:

$$
\begin{array}{ll}
\mathsf{S_{ds}} & :: \mathsf{S} \rightarrow \mathsf{state} \rightarrow \mathsf{state}; \\
\mathsf{S_{ds}}[\![\mathsf{x} := \mathsf{a}]\!]\mathsf{s} & = \mathsf{s} \oplus \{\mathsf{x} \mapsto \mathcal{A}[\![\mathsf{a}]\!]\mathsf{s}\}; \\
\mathsf{S_{ds}}[\![\texttt{skip}]\!] & = \mathsf{id}; \\
\mathsf{S_{ds}}[\![\mathsf{S}_1 \texttt{;} \mathsf{S}_2]\!] & = \mathsf{S_{ds}}[\![\mathsf{S}_2]\!].\mathsf{S_{ds}}[\![\mathsf{S}_1]\!]; \\
\mathsf{S_{ds}}[\![\texttt{if b then } \mathsf{S}_1 \texttt{ else } \mathsf{S}_2]\!] & = \mathsf{cond}(\mathcal{B}[\![\mathsf{b}]\!],\ \mathsf{S_{ds}}[\![\mathsf{S}_1]\!],\ \mathsf{S_{ds}}[\![\mathsf{S}_2]\!]); \\
\mathsf{S_{ds}}[\![\texttt{while b do } \mathsf{S}_1]\!] & = \mathsf{FIX}\ \mathsf{F} \\
 & \quad \textbf{where} \\
 & \quad \mathsf{F\ g} = \mathsf{cond}(\mathcal{B}[\![\mathsf{b}]\!],\ \mathsf{g}.\mathsf{S_{ds}}[\![\mathsf{S}_1]\!],\ \mathsf{id}); \\
 & \quad ;
\end{array}
$$

The auxiliary function FIX is the fixed-point combinator, and the cond function is a higher order version of the ordinary conditional:

$$
\begin{array}{ll}
\mathsf{cond} & :: (\alpha \rightarrow \mathsf{bool},\ \alpha \rightarrow \alpha,\ \alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha); \\
\mathsf{cond(p,\ g_1,\ g_2)s} & = \mathsf{g_1\ s}, \quad \textbf{if}\ \mathsf{p\ s} = \mathsf{tt}; \\
 & = \mathsf{g_2\ s}, \quad \textbf{otherwise}; \\
\mathsf{FIX} & :: ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha); \\
\mathsf{FIX\ g} & = \mathsf{g(FIX\ g)};
\end{array}
$$

The function $\mathsf{S_{ds}}$ can be applied to the factorial program and the sample initial state $\mathsf{s_3}$ to yield the final state as follows:

$$\mathsf{S_{ds}\ fact\ s_3} = \{\langle \mathsf{y} \mapsto 1\rangle,\ \langle \mathsf{z} \mapsto 6\rangle\}$$

A continuation semantics [3] is also easy to describe. It does not add much to what has been said before.

4. TRANSLATING RELATIONS

This section gives a transformational semantics to LETOS by describing how rules are translated into functions. The translation is loosely based on Johnsson's method for translating attribute grammars into lazy functional programs [25]. Here an axiom plays the role of a terminal, and an inference rule plays the role of a non-terminal in a grammar. An axiom, or the conclusion of an inference rule receives information through its left-hand side and produces information via its right-hand side. This is comparable to, respectively, inherited and synthesised attributes. The premises of an inference rule inherit information from either the left-hand side of the conclusion, or from other premises. Similarly, they synthesise

information for use by further premises or by the right-hand side of the conclusion and the side condition. Recent work on Typol is also based on this relationship between rules and attribute grammars [9]. Stepney [5] uses a similar method based on definite clause translation grammars [26], which are the logic programming equivalent of attribute grammars.

The following sections discuss the syntax and semantics of the LETOS rules. The definition of the remaining language elements, such as types and function definitions, relies entirely on the target language and is not discussed here.

4.1. Syntax of patterns and expressions

Rules contain patterns (P), and expressions (E). A pattern is a t-tuple (with $t = 0$ or $t > 1$), an n-ary constructor symbol C_n (with $n \geq 0$), or a variable v. Without loss of generality, we ignore here the usual syntactic sugar for lists and infix constructors:

$$\begin{aligned}\mathsf{P} \equiv\ & (\mathsf{P}_1,\ \ldots,\ \mathsf{P}_t) \mid \\ & \mathsf{C}_n\ \mathsf{P}_1\ \ldots\ \mathsf{P}_n \mid \\ & \mathsf{v};\end{aligned}$$

A pattern is *irrefutable* if it contains only tuples and variables. A pattern with constructor symbols is refutable.

An expression is either a t-tuple, the application of an n-ary constructor symbol, the application of an m-ary function symbol F_m (with $m \geq 0$) or a variable.

$$\begin{aligned}\mathsf{E} \equiv\ & (\mathsf{E}_1,\ \ldots,\ \mathsf{E}_t) \mid \\ & \mathsf{C}_n\ \mathsf{E}_1\ \ldots\ \mathsf{E}_n \mid \\ & \mathsf{F}_m\ \mathsf{E}_1\ \ldots\ \mathsf{E}_m \mid \\ & \mathsf{v};\end{aligned}$$

The expressions above represent a first order sub-language of the more general notion of expressions in a higher order functional language. This is somewhat restrictive, but it should be noted that the functions available in addition to the rules still provide the full power of higher order programming to LETOS specifications. The denotational semantics of Section 3.7 illustrate this point.

Function symbols (F) and constructor symbols (C) must be distinct. This is consistent with the use of patterns in most functional languages; languages based on term rewriting (e.g. ASF) take a more liberal view [27].

4.2. Syntax of axioms and inference rules

The syntax of an axiom without a side condition is given by the axiom schema below. The pattern P_0 to the left of the arrow is usually referred to as the subject of the axiom. The expression E_0 will be referred to as the object of the axiom.

$$[\mathsf{schema}_1]\ \mathsf{P}_0 \xrightarrow{R} \mathsf{E}_0;$$

If there is a side condition, which must be a boolean expression (E_b), the syntax is:

$$\begin{aligned}[\mathsf{schema}_2]\ & \mathsf{P}_0 \xrightarrow{R} \mathsf{E}_0, \\ & \textbf{if}\ \mathsf{E}_b;\end{aligned}$$

To cope with inference rules that have a different number of premises a family of rule-schema is used. The syntax of an inference rule with $n \geq 1$ premises, and without a side condition is given by the appropriate member of schema_3 below: Again, the pattern P_0 to the left of the arrow in the conclusion is referred to as the subject of the inference rule. The expression E_0 is the object of the inference rule.

$$\begin{array}{l} \qquad\qquad \mathsf{E}_1 \overset{1}{\rightarrow} \mathsf{P}_1, \\ \qquad\qquad \ldots, \\ \qquad\qquad \underline{\mathsf{E}_n \overset{n}{\rightarrow} \mathsf{P}_n} \\ [\mathsf{schema}_3]\ \mathsf{P}_0 \overset{R}{\rightarrow} \mathsf{E}_0; \end{array}$$

With a side condition, the syntax of an inference rule with n premises must conform to:

$$\begin{array}{l} \qquad\qquad \mathsf{E}_1 \overset{1}{\rightarrow} \mathsf{P}_1, \\ \qquad\qquad \ldots, \\ \qquad\qquad \underline{\mathsf{E}_n \overset{n}{\rightarrow} \mathsf{P}_n} \\ [\mathsf{schema}_4]\ \mathsf{P}_0 \overset{R}{\rightarrow} \mathsf{E}_0, \\ \qquad\qquad \textbf{if}\ \mathsf{E}_\mathsf{b}; \end{array}$$

Each of the arrows in the premises can be adorned with an asterisk to indicate the transitive closure of the relation.

In a subsequent section, the syntactic conditions will be given that have to be satisfied for axioms and inference rules to yield a syntactically correct translation into Miranda. To simplify the presentation, the four different kinds of axioms and inference rules will first be translated into inference rules of the kind schema_3. This is the subject of the following section.

4.3. Simplifying axioms and inference rules

Each axiom of the form schema_2 (i.e. with a side condition **if** E_b) is replaced by an inference rule with a single premise as shown below.

$$\begin{array}{l} \qquad\qquad \underline{\mathsf{E}_\mathsf{b} \overset{id}{\rightarrow} \mathsf{True}} \\ [\mathsf{schema}_3]\ \mathsf{P}_0 \overset{R}{\rightarrow} \mathsf{E}_0; \end{array}$$

The auxiliary relation $\overset{id}{\rightarrow}$ is the (polymorphic) equality relation:

$$[\mathsf{id}]\ \mathsf{x} \overset{id}{\rightarrow} \mathsf{x};$$

Similarly, each inference rule of the form schema_4 (i.e. with a side condition **if** E_b and n premises) is replaced by an inference rule of the form schema_3 with an extra premise $\mathsf{E}_\mathsf{b} \overset{id}{\rightarrow} \mathsf{True}$.

As a second simplification the distinction between axioms and inference rules will be dropped; an axiom will be treated as an inference rule with 0 premises. From now on all rules will thus be of the form schema_3.

4.4. Translating rules to functions

In what follows, let $\mathsf{fv}(\cdot)$ denote the set of free variables of a given pattern or expression. A rule (after the simplification of the previous section) should satisfy the two syntactic constraints below:

pure A rule should not have free variables:

$$\bigcup_{k=0}^{n} \mathsf{fv}(E_k) \subseteq \bigcup_{k=0}^{n} \mathsf{fv}(P_k)$$

linear Patterns must be pairwise disjoint; if $i \neq k$ then:

$$\mathsf{fv}(P_i) \cap \mathsf{fv}(P_k) = \emptyset$$

A set of rules is pure (linear) if all rules are pure (linear).

There are two reasons for insisting on pure rules. The first is that impure rules are not necessarily operationally conservative (see Section 4.5). The second reason is that the semantics of functional programming languages are defined in terms of closed lambda terms. If free variables were admitted, then a translation into Prolog would be more appropriate [8]; the free variables would then be represented as logic variables.

The second condition is a linearity condition, which avoids variables being defined more than once. Without the linearity condition, unification would be required to execute rules.

The linearity requirement does not affect operational conservativity. However, the requirement does make it less convenient to specify a semantics that essentially uses unification, such as a type checker. In such a case, one has to manipulate substitutions explicitly and program the unification process [28]. In the dynamic semantics of the languages that LETOS has been applied to (see Sections 5 and 6), the restriction to linear rules has not posed a problem.

For each pure and linear member the family of schema_3 an appropriate functional clause is created as shown below. This forms the basis of the translation. The Miranda code fragments discussed earlier in the paper provide concrete examples of the translation.

$$\begin{aligned}
\mathsf{rule}_R[P_0] = [E_0], \quad & \textbf{if } \mathsf{non_empty}\ t_1 \wedge \ldots\ \mathsf{non_empty}\ t_n \\
& \textbf{where} \\
& P_1 \ = \mathsf{last}\ t_1; \\
& t_1 \ = \mathsf{rule}_1[E_1]; \\
& \quad \vdots \\
& P_n = \mathsf{last}\ t_n; \\
& t_n \ = \mathsf{rule}_n[E_n]; \\
& ;
\end{aligned}$$

In the next two sections, the generated Miranda is refined to take into account refutable patterns and the reflexive transitive closure of relations. In the section thereafter, the syntactic constraints are discussed again in relation to the notion of operational conservativity.

4.4.1. Translating refutable patterns

Most patterns that occur in the premises of an operational semantics are tuples containing only variables. Such patterns are irrefutable, that is, they will always match. Patterns that

contain constructors are refutable; they can fail to match. The patterns introduced by the simplification to support side conditions are an example of refutable patterns.

LETOS generates code to support refutable patterns as follows. For each refutable pattern P_i, the test non_empty t_i is replaced by the test $\mathsf{match}_i\ t_i$. Furthermore, a new function definition is generated, to decide whether the match succeeds:

$$\begin{aligned}&\mathsf{match}_i(P_i : \_) = \mathsf{True};\\ &\mathsf{match}_i\ \_ \quad\ = \mathsf{False};\end{aligned}$$

As an example, consider the relation $\xrightarrow{gt}$ below, where both x and y range over {0, 1}:

$$\begin{array}{ll} & \xrightarrow{gt} :: (\mathsf{num},\ \mathsf{num}) \leftrightarrow \mathsf{bool};\\ & \quad \mathsf{x} \xrightarrow{id} 1,\\ & \quad \underline{\mathsf{y} \xrightarrow{id} 0}\\ [\mathsf{gt}] & \quad (\mathsf{x},\ \mathsf{y}) \xrightarrow{gt} \mathsf{True};\end{array}$$

The translation of this rule into a functional clause called rule_{gt} is shown below. (This program fragment represents code that has first been generated by LETOS, and then processed again to pretty print the code.)

$$\begin{aligned}&\mathsf{rule}_{gt} \quad\ :: [(\mathsf{num},\ \mathsf{num})] \rightarrow [\mathsf{bool}];\\ &\mathsf{rule}_{gt}[(\mathsf{x},\ \mathsf{y})] = [\mathsf{True}],\ \ \textbf{if}\ \mathsf{match}_1\ t_1 \wedge \mathsf{match}_2\ t_2\\ &\qquad\qquad\qquad\quad \textbf{where}\\ &\qquad\qquad\qquad\quad t_1 = \mathsf{rule}_{id}[\mathsf{x}];\\ &\qquad\qquad\qquad\quad t_2 = \mathsf{rule}_{id}[\mathsf{y}];\\ &\qquad\qquad\qquad\quad ;\\ &\mathsf{rule}_{gt}\ \mathsf{x} \quad\ = [];\end{aligned}$$

According to the general pattern for match_i above, new function definitions are generated to perform the matching as follows:

$$\begin{aligned}&\mathsf{match}_1,\ \mathsf{match}_2 :: [\mathsf{num}] \rightarrow \mathsf{bool};\\ &\mathsf{match}_1(1 : \_) \quad = \mathsf{True};\\ &\mathsf{match}_1\ \_ \quad\ = \mathsf{False};\\ &\mathsf{match}_2(0 : \_) \quad = \mathsf{True};\\ &\mathsf{match}_2\ \_ \quad\ = \mathsf{False};\end{aligned}$$

When rule_{gt} is executed, the first step is to bind the variables (x and y). The second step is to evaluate the first conjunct of the guard ($\mathsf{match}_1\ t_1$). Then there are two possibilities:

False The guard fails and the function rule_{gt} returns the empty list.

True Otherwise, the third step is to evaluate the second conjunct ($\mathsf{match}_2\ t_2$), giving two possibilities again:

False The entire guard fails and the function rule_{gt} returns the empty list.

True Otherwise, the entire guard succeeds, so that the function rule_{gt} will return the singleton list [True].

4.4.2. Translating closures

The premises of a rule can be adorned with an asterisk to indicate that the transitive closure of the relation is desired:

$$\vdash \mathsf{E_i} \xrightarrow{\mathsf{i}} * \mathsf{P_i}$$

The where clause generated in this case relies on the support function closure as follows:

$$\begin{aligned} \mathsf{P}_i &= \mathsf{last\ t_i};\\ \mathsf{t_i} &= \mathsf{closure\ rule_i[E}_i\mathsf{]};\end{aligned}$$

The definition of the function closure is shown below. It repeatedly tries to apply the appropriate function, passed as the argument r, until the latter yields an empty list of successes. A rule using a closure may thus deliver a result list with more than one element. The SOS and abstract machine semantics of While provide examples of use.

$$\begin{aligned} \mathsf{closure\ r\ s} &= \mathsf{s'} : \mathsf{closure\ r[s']}, \quad \textbf{if}\ \mathsf{non\_empty\ ss};\\ &= [], \quad \textbf{otherwise}\\ &\quad \textbf{where}\\ &\quad \mathsf{ss} = \mathsf{r\ s};\\ &\quad \mathsf{s'} = \mathsf{last\ ss};\\ &\quad ;\end{aligned}$$

A transition system is fully defined by a collection of rules that bear the same label (l say). Each of the individual rules is translated into the definition of a function clause as described above. In addition, a default clause is added. The functional clauses together then fully define a function rule_l.

This completes the description of the semantics of LETOS.

4.5. Operational conservativity

The extension of a set of rules is *operationally conservative* if provable transitions in the original system are the same as those in the extended system. Groote en Vaandrager [29] show that the original system must be pure and *well-founded*. LETOS specifications are always pure, but they need not be well founded. Rather than reproducing the definition of well-founded here, we give an example (below) of a legitimate LETOS rule that is not well-founded. The problem is the cyclic dependency between the two premises (the problem is not the infinitary nature of the rule). Rules such as this do occur; Mini-Freja (see Section 5) uses a similar rule to create a (finite) environment for a `letrec` construct.

$$[\mathsf{cycle}]\ \frac{\mathsf{x} : \mathsf{xs} \xrightarrow{id} \mathsf{ys}, \quad \mathsf{y} : \mathsf{ys} \xrightarrow{id} \mathsf{xs}}{\langle \mathsf{x}, \mathsf{y} \rangle \xrightarrow{xyxy} \mathsf{ys};}$$

Well-foundedness is an awkward property to check. Fortunately, Fokkink and Verhoef [13] show that a more liberal condition is sufficient: the original system must be source dependent. A rule is *source dependent* if all variables in the rule are source-dependent. Using the notation for schema_3 from Section 4.2, we define set of source dependent variables inductively as follows:

- All variables in P_0 are source-dependent.
- If all variables in E_i are source-dependent, then the variables in P_i are source-dependent.
- No other variables are source-dependent.

LETOS issues a warning when a rule is not source-dependent. This helps writing operational semantics that can be extended later. It should be noted that an extension to a source-dependent set of rules should itself be source dependent, and that the extension should satisfy a number of further requirements [13] (Theorem 3.21).

Working with an operationally conservative semantics is a technique that allows a language designer to save effort, because a valid property of a language remains valid in an extension of the language. Baeten and Verhoef [30] give a systematic presentation of the technique. They also discuss a host of languages, showing how basic theorems can be reused over and over again. To illustrate this with an example, consider the usual inductive definition of the natural numbers n:

$$n \equiv 0 \mid S(n);$$

The relation $\rightarrow$ given by the two rules below defines addition over the natural numbers. Both rules are pure and source-dependent.

$$[+^0] \quad \langle 0 + p \rangle \rightarrow p;$$

$$[+^1] \quad \frac{\langle p + q \rangle \rightarrow r}{\langle S(p) + q \rangle \rightarrow S(r);}$$

With these two rules it is possible to prove useful properties, for example that addition is commutative. Now suppose that the set of rules is extended with two further rules to define multiplication thus:

$$[\times^0] \quad \langle 0 \times p \rangle \rightarrow 0;$$

$$[\times^1] \quad \frac{\langle p \times q \rangle \rightarrow r, \quad \langle q + r \rangle \rightarrow s}{\langle S(p) \times q \rangle \rightarrow s;}$$

It would be tedious if the proof of the commutativity of addition would have to be repeated for the extended set of rules. Fortunately, the set of four rules $+^0$, $+^1$, $\times^0$ and $\times^1$ is a conservative extension of the initial set ($+^0$, $+^1$), which automatically discharges the proof obligation for the extended system.

The essence of the operational conservativity is the notion of source-dependency. This notion is not only useful in the realm of operational conservativity, but also in conservativity of rewrite systems. For example, Fokkink and Verhoef [31] give a treatment of source-dependency for term rewriting systems with applications to software renovation. Roughly, a component that rewrites legacy code into better code, can also extend it with more functionality. When the extension is conservative, we can be sure that the original functionality of the renovation component is not affected by the extension. Again, it is crucial to check for source-dependency. To illustrate source-dependency, we provide a simple example, also due to Fokkink and Verhoef [13].

Given two constants a and b, consider the inference rule below. With this rule alone it is not possible to prove that $a \rightarrow a$:

$$[a] \quad \frac{x \rightarrow x}{a \rightarrow a;}$$

However, if rule a is extended with axiom b as shown below, it does become possible to give such a proof in the extended system (by instantiating the variable x to the constant b):

$$[b] \quad b \rightarrow b;$$

Rule a is not source-dependent (because the variable x is not source-dependent). Therefore, rules a and b together do not form an operationally conservative extension of rule a. The fact that rule a is not source-dependent warns us of this undesired behaviour.

5. ASSESSMENT

To demonstrate that LETOS is a useful tool, we first characterise the most important specifications that have been built using LETOS. The total portfolio of LETOS specifications is 27 languages and systems, totalling over 9000 lines of formal text. To show that LETOS also offers good performance, we compare it with RML.

5.1. Functional assessment

LETOS has been applied to a number of operational semantics and models from various sources to assess its usefulness:

JSP (183 rules) The Java Secure Processor (JSP) is a byte code interpreter for a smart card implementation of Java [32]. The JSP does not support floating point data types, dynamic linkage, garbage collection or exceptions, but it does support the full object model of Java. The complete SOS of the JSP has been specified as a case study of a realistically sized SOS. Without a tool such as LETOS it would not have been possible to execute and trace any but the most trivial Java programs, because of the large number of byte codes involved. LETOS proved to be an essential tool to validate the semantics by supporting experiments with a suite of small to medium sized Java programs.

π-calculus with application (72 rules) A deterministic operational semantics of the π-calculus [33] has been specified using LETOS [34]. The resulting tool has been used for the exhaustive state space exploration of a simple model of a distributed data base system. The state space exploration has pointed out two problems with the π-calculus model of the data base: first, the model used global names (instead of private names) to communicate the response to a request. This is open to abuse by other processes. Secondly, the model allowed names to escape their scope, which makes it vulnerable to undesired interactions with other agents. Problems such as these commonly occur in π-calculus specifications. They are generally difficult to detect, unless one uses an appropriate tool. This was easily provided by the LETOS execution of the model.

QuickPay (13 rules) is a system for micro-payments aiming to avoid the cost of cryptographic operations during payments [35]. An operational model of the system

has been built to assist in the search for weaknesses in the protocols. The messages exchanged by the QuickPay protocols have been modelled by LETOS rules. The parties and the messages involved in the protocols are shown in Figure 3. The **update** transaction supplies the customer with value tokens, which can be passed on to the merchant during a **sell** transaction, and thence to the broker for clearing. Authentication tokens are provided by the broker to the merchant (**stock**) and subsequently exchanged between broker and merchant during mutual authentication (**auth**$_m$, **auth**$_b$).

As a result, the model building activity, one minor weakness has been found. Another, more serious, weakness has been re-discovered, and a number of solutions are proposed. The strongest solution is proved correct. LETOS executions were instrumental in discovering counter-examples for certain desirable properties, such as the conservation of electronic money.

Scil (80 rules) The secure card instruction language (SCIL) is a threaded code language designed for writing secure and compact smart card operating systems [36]. LETOS has been applied to specify the operational semantics of extended high and low level variants of SCIL, as well as a compiler from the high to the low level language.

Mini-Freja (67 rules) Mini-Freja is a call-by-name pure functional language [6]. The operational semantics for the language is available in Typol and in RML. The LETOS version is a literal translation from the RML version, which itself is a literal translation of the Typol version.

While (40 rules) The language While has been the running example of this paper. The differences between our version of the various styles of semantic specifications (NS, SOS, DS and AM) and the same specifications from the literature are minor.

TTA (19 rules) A Transport Triggered Architecture (TTA) is a novel kind of Very Long Instruction Word (VLIW) architecture. The instruction set and the instruction fetch and execute cycle of this architecture has been described as a parallel, synchronous SOS [37].

Memory manager (9 rules) An operational semantics of the memory manager of a smart card operating system has been specified using LETOS.

Pattern matching compiler (34 functions) A compiler for pattern matching functions in a style similar to that found in Peyton Jones [28] has been specified.

This represents a relatively wide range of operational semantics, though with the exception of the Java Secure Processor, all of a relatively small size.

5.2. Performance assessment

The LETOS+Miranda combination is useful during development because the combination provides fast compilation. When fast execution is desired, the generated Miranda can be converted into Haskell, for which good compilers are available. We have used GHC, the Glasgow Haskell compiler [38]. This conversion can be done largely automatically using a simple `sed` script (it would also be straightforward to extend LETOS to generate Haskell directly).

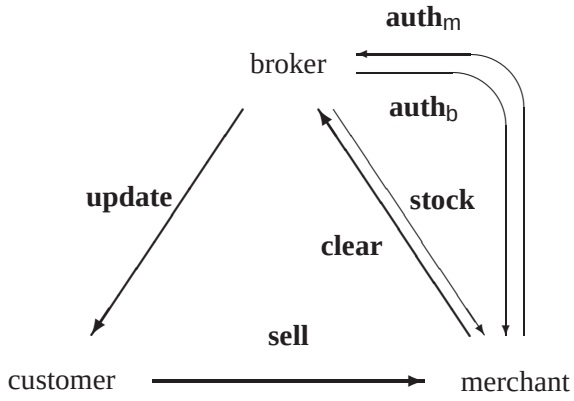


Figure 3. The QuickPay parties and transactions. Thick arrows represent value token transfer and thin arrows represent authentication token transfer

Table I. Using the Mini-Freja operational semantics to compare the performance of RML, Sicstus Prolog, LETOS+GHC and LETOS+Miranda. Seconds represent average user+system time over 10 runs on a SPARCstation-20, 60 MHz, 128 MB, Solaris 2.5. The Prolog version of Mini-Freja ran out of memory when calculating the first 60 primes.

Compiler	RML	Sicstus	LETOS+GHC	LETOS+Miranda
Version	2.0-pl3	2.1 #9	3.02	2.020
Option	`-O2`	fastcode	`-O`	
Time	sec.	sec.	sec.	sec.
Compilation	6.7	2.0	111.3	3.2
Primes 18	0.1	1.9	0.8	42.8
Primes 30	0.4	8.8	3.2	171.1
Primes 60	4.2	—	26.1	1423.5

To assess the performance of the code generated by LETOS, the operational semantics for Mini-Freja has been used to compute the first n primes (for $n = 18$, $n = 30$ and $n = 60$), using the sieve of Eratosthenes.

Experiments have been carried out using four different compilers as shown in Table I. Times reported are in seconds user+system time, measured as an average over 10 runs. The first two columns show the results for repeating the experiments from Table 10.8 in Mikael Pettersson's PhD thesis [6]. The third column reports the results obtained after translating the Miranda code generated by LETOS into Haskell. The last column applies to direct execution of the Miranda code.

The conclusions from this experiment are:

- The best compile time is provided by Sicstus Prolog, closely followed by the LETOS+Miranda combination. The best run time is delivered by RML, followed by the LETOS+GHC combination.
- Using existing technology makes it possible to build a flexible system with limited

effort. Miranda offers fast compilation and slow execution, which would be appropriate most of the time. Haskell offers faster execution, at the cost of longer compilation times.
- Chosing a lazy functional language as a target works well; the runtime results are better than those obtained with a good Prolog compiler.
- Writing a specialised tool that concentrates on one aspect (fast execution) pays off: RML is fast.

6. LIMITATIONS

LETOS has two limitations to its execution capabilities: a semantics should be deterministic, and execution should not rely on unification. These limitations have not caused difficulty whilst specifying extensive collection of deterministic, dynamic semantics reported in the previous section. However, the limitations do pose problems for the specification of concurrent systems and type inference. These applications will be discussed in the following two sections.

6.1. Non-determinism

The disadvantage of using a functional language to execute semantic specifications is the lack of direct support for working with relations. However, the 'list-of-successes' method [39] can be used to create a function that will simulate a relation. When given a relation $\mathsf{R} :: \mathsf{A} \leftrightarrow \mathsf{B}$, this method creates a function F as follows:

$$\begin{aligned} F &:: A \rightarrow \{B\}; \\ F\ \mathsf{a}' &= \{\mathsf{b} \mid \langle \mathsf{a},\ \mathsf{b} \rangle \in R \wedge\ \mathsf{a} = \mathsf{a}'\}; \end{aligned}$$

If a relation is deterministic, the corresponding function either delivers a singleton set to represent success, or an empty set representing failure. For a non-deterministic relation there might be several successes. When LETOS executes a non-deterministic specification, only one result is delivered. This is a safe approximation to the desired set of results. This restriction can be lifted, because lazy evaluation makes it possible to generate all possible successes *in principle*, but in practice only ever to use one or just a few successes. This is the subject of the next section.

6.1.1. Non-deterministic CCS fragment

A separate paper discusses how the complete π-calculus [33] can be specified using LETOS [34]. To illustrate the principle here, a fragment of the basic agent constructions of CCS [40] will be presented in LETOS. The fragment omits restriction, recursion and relabelling. Adding these features would make the example more complicated, but not more informative. The abstract syntax of the fragment is given below. The syntactic category *Act* specifies the actions, where τ is the silent action. The variables α and β range over *Act*. The syntactic category $\mathcal{E}$ names the inactive agent (**0**), it describes the action prefix of an agent expression ($\alpha \cdot \mathrm{E}$), summation ($\mathrm{E} + \mathrm{F}$) and communication ($\mathrm{E} \| \mathrm{F}$). The variables E and F range over $\mathcal{E}$:

$$\begin{aligned} \alpha, \beta \in Act &\equiv \tau \mid \mathrm{a} \mid \bar{\mathrm{a}} \mid \mathrm{b} \mid \bar{\mathrm{b}} \mid \ldots; \\ \mathrm{E}, \mathrm{F} \in \mathcal{E} &\equiv \mathbf{0} \mid \alpha \cdot \mathrm{E} \mid \mathrm{E} + \mathrm{F} \mid \mathrm{E} \parallel \mathrm{F}; \end{aligned}$$

The operational semantics of the fragment of CCS below is based on Milners specification [40] (Section 2.5). The type declaration shows that $\rightarrow$ relates an agent

expression E to a configuration $\langle \alpha, \mathrm{E} \rangle$, where α is the action taken. LETOS does not permit associating an action label with the arrow of the transition. Instead, an action label is associated with the right hand side of the transitions.

$$
\begin{array}{ll}
\rightarrow & :: \mathcal{E} \leftrightarrow \langle Act,\ \mathcal{E} \rangle; \\
[\mathsf{act}] & \alpha \cdot \mathrm{E} \rightarrow \langle \alpha,\ \mathrm{E} \rangle; \\
 & \\
 & \dfrac{\mathrm{E} \rightarrow \langle \alpha,\ \mathrm{E}' \rangle}{\mathrm{E} + \mathrm{F} \rightarrow \langle \alpha,\ \mathrm{E}' \rangle}; \\
 & \\
 & \dfrac{\mathrm{F} \rightarrow \langle \beta,\ \mathrm{F}' \rangle}{\mathrm{E} + \mathrm{F} \rightarrow \langle \beta,\ \mathrm{F}' \rangle}; \\
 & \\
 & \dfrac{\mathrm{F} \rightarrow \langle \alpha,\ \mathrm{E}' \rangle}{\mathrm{E} \parallel \mathrm{F} \rightarrow \langle \alpha,\ \mathrm{E}' \parallel \mathrm{F} \rangle}; \\
 & \\
 & \dfrac{\mathrm{F} \rightarrow \langle \beta,\ \mathrm{F}' \rangle}{\mathrm{E} \parallel \mathrm{F} \rightarrow \langle \beta,\ \mathrm{E} \parallel \mathrm{F}' \rangle}; \\
 & \\
 & \dfrac{\begin{array}{l}\mathrm{E} \rightarrow \langle \alpha,\ \mathrm{E}' \rangle, \\ \mathrm{F} \rightarrow \langle \beta,\ \mathrm{F}' \rangle\end{array}}{\begin{array}{l}\mathrm{E} \parallel \mathrm{F} \rightarrow \langle \tau,\ \mathrm{E}' \parallel \mathrm{F}' \rangle, \\ \textbf{if } \mathsf{match}(\alpha,\ \beta);\end{array}}
\end{array}
$$

An auxiliary function match is used to yield true when presented with a and $\bar{a}$, or with b and $\bar{b}$, etc.

Non-determinism arises because there is a choice between sum_1 and sum_2 for any agent expression of the form $\mathrm{E} + \mathrm{F}$. There are even three choices for an agent expression of the form $\mathrm{E} \| \mathrm{F}$.

The semantics can be made deterministic by a source to source transformation. The idea is to replace each set of rules with a common subject (and arbitrary object) by a new rule. This new rule relates the common subject to the set of all possible objects. The result of applying the transformation to the fragment of CCS is presented in the next section.

6.1.2. Deterministic CCS fragment

The source to source transformation changes the type of the relation $\rightarrow$ such that it relates an agent expression to a set of configurations, instead of a single configuration. The axiom undergoes a similar change, yielding

$$
\begin{array}{ll}
\rightarrow & :: \mathcal{E} \leftrightarrow \{\langle Act,\ \mathcal{E} \rangle\}; \\
[\mathsf{act}] & \alpha \cdot \mathrm{E} \rightarrow \{\langle \alpha,\ \mathrm{E} \rangle\};
\end{array}
$$

The summation rules sum_1 and sum_2 have one premise each. These premises are independent. Both must therefore be carried over to the combined rule. The new premises each relate an agent to a set of configurations $\tilde{\mathrm{E}}$ and $\tilde{\mathrm{F}}$. We therefore take as the object of the

new summation rule the union of the configurations:

$$\frac{\begin{array}{l}E \to \tilde{E},\\ F \to \tilde{F}\end{array}}{[\mathsf{sum}]\ E + F \to \tilde{E} \cup \tilde{F};}$$

Gathering all premises from $\mathsf{com}_1 \ldots \mathsf{com}_3$, and removing the duplicates to avoid name clashes yields the two premises below. The objects of each of the three rules are combined into a single object.

$$\frac{\begin{array}{l}E \to \tilde{E},\\ F \to \tilde{F}\end{array}}{\begin{array}{l}[\mathsf{com}]\ E \parallel F \to\\ \{\langle \alpha,\ E' \parallel F\rangle \mid \langle \alpha,\ E'\rangle \in \tilde{E}\} \cup\\ \{\langle \beta,\ E \parallel F'\rangle \mid \langle \beta,\ F'\rangle \in \tilde{F}\} \cup\\ \{\langle \tau,\ E' \parallel F'\rangle \mid \langle \alpha,\ E'\rangle \in \tilde{E} \wedge \langle \beta,\ F'\rangle \in \tilde{F} \wedge\\ \mathsf{match}(\alpha,\ \beta)\};\end{array}}$$

Finally, if there are agent expressions for which the non-deterministic semantics does not give a transition, these must be related to the empty set by the determinstic semantics:

$$[\mathsf{inaction}]\ \mathbf{0} \to \{\};$$

The correctness of the transformation can be proved by induction on the shape of the derivation trees.

LETOS can now execute the relation $\to$ to generate all possible transitions of an agent expression. For example, the agent expression below would evolve in one step to four different configurations, thus:

$$\begin{array}{rl}(a \cdot E + b \cdot \mathbf{0}) \parallel \bar{a} \cdot F \to \{ & \langle a,\ E \parallel \bar{a} \cdot F\rangle,\\ & \langle b,\ \mathbf{0} \parallel \bar{a} \cdot F\rangle,\\ & \langle \bar{a},\ a \cdot E + b \cdot \mathbf{0} \parallel F\rangle,\\ & \langle \tau,\ E \parallel F\rangle\};\end{array}$$

It would be possible to extend LETOS with the capability to perform the above source-to-source transformation. However, our experience with the π-calculus semantics shows that to control the potentially large number of results poses practical problems. There are two possible approaches. The first is to let the user make the choices interactively. The second approach is to check properties of intermediate and/or final configurations in a model checking fashion. This is how the deterministic semantics of the π-calculus has been validated [34]. The first approach scales up better but cannot be automated. The second approach only works with finite state spaces but it can be automated.

6.2. Unification

The previous section showed how a relation $\mathsf{R} :: \mathsf{A} \leftrightarrow \mathsf{B}$, can be simulated by a function. However, this relies on the assumption that the relation is used such, that when given some

$a \in A$ (a is *ground*), we wish to know which values of $b \in B$ satisfy $a\ R\ b$. The simulation method above would also work if the relation is to be driven backwards, i.e. if $b \in B$ is ground. However the simulation method would fail if neither a nor b are fully ground. Such a situation would arise typically when specifying a type inference system.

With considerable extra work, it would be possible to use LETOS to specify a type inference system. For example Milner's [41] algorithm $\mathcal{W}$ (and Robinson's unification algorithm [42] which it uses) could be written directly using LETOS. However, it would probably be more convenient to extend LETOS so that it would generate Prolog.

However, to show that the execution capabilities of LETOS are not limited to specifying dynamic semantics, the next section present a type checker for the simply typed λ-calculus.

6.2.1. Simply typed λ-calculus

The abstract syntax of Church' version of the simply typed lambda calculus [43] is given below. The type τ defines function types $\tau \to \tau$, a ground type 0 and an error type $\perp$. The latter has been included to cope with terms that cannot be typed in this particular calculus. The syntax of an expression e shows how a bound variable is annotated with its type $v : \tau$. The context Γ maps variables to types.

$$
\begin{aligned}
&v \equiv a \mid b \mid \ldots;\\
&\tau \equiv \tau \to \tau \mid 0 \mid \perp;\\
&e \equiv v \mid \lambda\, v : \tau \cdot e \mid e\ e;\\
&\Gamma \equiv \{\langle v \mapsto \tau \rangle\};
\end{aligned}
$$

The axiom var below shows that the type of the variable v can be inferred from the context Γ, provided the variable actually occurs in the context.

The inference rule lam asserts that the type of the lambda abstraction $\lambda v : \tau \cdot e$ is the function type $\tau \to \sigma$. The premise asserts that the type of the expression e is σ in a context which maps v onto the type τ.

The inference rule app asserts that the expression $f\ e$ has type μ, when the function f has type $\sigma \to \mu$ and when the argument e has type τ, under the condition that σ and τ are the same.

These three rules suffice to specify well typings. They do not specify what should happen in case of an error. Therefore, two extra axioms have been added, which apply when none of the earlier rules apply. The axiom var^{err} applies when v does not occur in the context Γ. The axiom app^{err} applies when either f does not yield a function type, or when the formal and actual argument types differ.

$$
\begin{aligned}
&[\text{var}] \quad \langle w \mapsto \tau \rangle \in \Gamma \wedge\ v = w \vdash v : \tau;\\
&\qquad\qquad \Gamma \oplus \{v \mapsto \tau\} \vdash e : \sigma\\
&[\text{lam}] \quad \Gamma \vdash \lambda\, v : \tau \cdot e : \tau \to \sigma;\\
&\qquad\qquad \Gamma \vdash f : \sigma \to \mu,\\
&\qquad\qquad \Gamma \vdash e : \tau,\\
&\qquad\qquad \sigma = \tau\\
&[\text{app}] \quad \Gamma \vdash f\ e : \mu;
\end{aligned}
$$

$$[\mathrm{var}^{\mathrm{err}}]\quad \Gamma \vdash \mathsf{v} : \bot;$$

$$[\mathrm{app}^{\mathrm{err}}]\quad \Gamma \vdash \mathsf{f}\ \mathsf{e} : \bot;$$

The specification is succinct, close to a textbook representation and directly executable by LETOS. To demonstrate this, consider the following λ-terms:

$$\mathsf{K} = \lambda\ \mathsf{a} : 0 \cdot \lambda\ \mathsf{b} : 0 \cdot \mathsf{a};$$
$$\mathsf{I}\ = \lambda\ \mathsf{a} : 0 \to 0 \to 0 \cdot \mathsf{a};$$

The LETOS execution of the type checker calculates the types of expressions as follows:

$$\mathsf{I} : (0 \to 0 \to 0) \to (0 \to 0 \to 0);$$
$$\mathsf{K} : 0 \to 0 \to 0;$$
$$\mathsf{I}\ \mathsf{K} : 0 \to 0 \to 0;$$
$$\mathsf{K}\ \mathsf{I} : \bot;$$

No unification is necessary, as all type information is explicit in the terms.

7. RELATED WORK

The execution of specifications (not specifically of programming languages) by means of translation, either mechanical or by hand, into declarative languages (Prolog, SML, Miranda and Haskell) has been practised widely and for a long time. See Sherrell and Carver [44] for a survey.

In the domain of programming language specification, considerable effort has been devoted to execution of denotational semantics [5,45], continuation semantics [46], natural semantics [1,47], structured operational semantics [48], and algebraic specifications of various styles of semantics [2]. Publication quality rendering always has a high priority.

Dinesh and Üsküdarlı have used the ASF+SDF system to typeset and execute the various semantics of the While language [7]. They report a different ambiguity in the book of Nielson and Nielson from the two ambiguities reported in this paper. This shows that, depending on the properties of the tool used for executing the semantics, different aspects of the specification are found to be ambiguous.

A number of papers have been written to argue in favour or against the use of executable specifications. Gravell and Henderson provide a recent overview of the discussion [49]. In the context of specifying programming language semantics and translators, the view seems to be somewhat in favour of getting some help from tools to execute specifications.

One report has been found in the literature, where HyperCard stacks are used to support execution [50]. In recent work, Grundy [51] proposes a HTML package called ProofViews to allow window inference style proofs to be browsed comfortably using Netscape. The idea is attributed to Lamport [52].

7.1. Comparing LETOS, RML and Typol

The LETOS, RML and Typol meta-languages are similar in that they all represent an operational semantics by configurations and relations over configurations. In principle, the three systems work in the same fashion. When given an initial configuration (goal) a matching rule is found, whose evaluation (proof) gives rise to subsidiary goals, which are matched in turn. When all goals have been matched and evaluated, the process terminates. The three systems differ in many of the details, as summarised in Table II.

Table II. Differences between Letos, RML and Typol

Property	Letos	RML	Typol
	Target dependency		
Target	Miranda/Haskell	C	Prolog/Functional
Depends on target	a lot	not at all	a little
	Types		
Typing	strong	strong	strong, can be weakened
Polymorphic	yes	yes	no
Overloading	Miranda:no Haskell:yes	yes	no
	Axioms and inference rules		
Labelled relations	yes	yes	yes
Rules order department	yes	no	yes
Premises order department	no	no	Prolog: no Functional: yes.
Assumptions	yes	none	sequent calculus style
Side condition	yes	yes	yes
Higher order	function	no	no
Data flow in/out	fixed	fixed	Prolog: decided by analysis Functional: fixed
	Expressions		
Side effects	no	yes	yes
Sets, comprehensions	yes	no	no
	Evaluation strategy		
Backtracking	no	yes	yes
Unification	no	yes	Prolog: yes Functional: no
	Input		
Abstract syntax	Curried ADT	Uncurried ADT	Uncurried ADT + list arity
Parsing	no	no	yes (Metal)
	Output		
Pretty printing	yes	no	yes (Ppml)
Execution	yes	no	yes
Animation	no	no	yes
Debugging	no	no	Prolog 4-port model
	Intended application area		
Operational semantics	yes	yes	yes
Denotational semantics	yes	no	no
Compilation schemes	yes	no	yes

- **Target dependency** The three meta-languages are compiled into different programming languages. LETOS may be compiled into Miranda and subsequently into Haskell, Typol may be compiled into Prolog or a functional notation based on attribute grammars [9]. RML is compiled into C. The choice of target language influences the properties of the meta-language in the case of LETOS and Typol, but not in the case of RML.
- **Types** All three systems impose a strong typing discipline. Typol requires rules to be accompanied by an explicit type judgement. A judgement is optional for LETOS and RML, which will infer the correct judgement. Typol allows the typing to be weakened by permitting all constructs in a language specification to be identified. LETOS and RML support polymorphism, Typol does not. Overloading is supported by bounded polymorphism in RML. LETOS supports overloading only if the target language is Haskell. Typol does not support overloading.
- **Axioms and inference rules** All three systems permit relations to be named (using different syntax to achieve this). Unlabelled relations are common in textbooks, but they are not usable in the strongly typed meta-languages discussed here. In a proper logical framework, rules and premises within an inference rule should be order independent. This is the case in RML. In LETOS and Typol, the order of rules matters because the order of clauses in the target languages matters. LETOS and Typol support assumptions, RML does not. Assumptions may be encoded as part of the configurations in RML. Side conditions are provided by all three systems. RML and LETOS do not impose restrictions on the use of side conditions. LETOS is higher order, in that it permits functions to be manipulated by functions and by rules; RML and Typol are first order. None of the systems permit rules to be manipulated by rules. LETOS and RML make a specific assumption about the order in which values are bound to variables. Typol is more flexible in this respect, but only if the target language is Prolog.
- **Expressions** LETOS provides the standard notation for sets, including ZF-expressions. It does not provide side-effects, which are supported by RML (ref expressions as in SML) and Typol (global variables).
- **Evaluation strategy** LETOS does not permit backtracking at present, but a trivial modification would allow it. RML and Typol support backtracking. LETOS and the functional version of Typol are based on pure pattern matching, whereas RML and the Prolog version of Typol support unification. The lack of unification means that it is a laborious process to specify type inference in LETOS, because the unification must be made explicit (see Peyton Jones [28], Chapter 8, for an example).
- **Input** RML has the most restricted system for representing an abstract syntax, because it supports SML style uncurried constructors only. LETOS supports curried constructors (which subsume uncurried constructors). Typol also provides list arity constructors. Typol provides a concrete syntax facility and parsing (based on lex and yacc) via the Metal language. LETOS and RML do not provide concrete syntax or parsing facilities.
- **Output** RML provides no support for pretty printing, LETOS provides an effective but inflexible pretty printing facility. Typol provides a powerful pretty printing facility in the form of the Ppml language. Typol provides the standard Prolog 4-port model for debugging, with a versatile user interface. RML and LETOS do not provide debugging facilities apart from the ubiquitous edit-compile-go cycle. RML provides no tracing of executions, LETOS provides a simple off-line rendering facility of derivation trees using Netscape. Typol offers the user interaction with the proof process in a flexible way.

- **Intended application area** Both RML and Typol have been designed specifically to work with an operational semantics. LETOS also supports denotational semantics, although it does not support tracing the execution of a denotational semantics.

8. CONCLUSIONS

LETOS is a small program that makes it possible to enter an operational semantics, to render the specification using LaTeX and to execute and trace the specification using Miranda. Netscape can be used to browse derivation trees. The tool has been applied to a number of languages and systems to demonstrate its usefulness. Owing to its relative simplicity, LETOS should be considered a lightweight alternative to its more powerful brethren. In combination with a state-of-the-art compiler, the execution times of LETOS specifications are competitive. In combination with LaTeX, the rendering facilities are of publication quality.

Applying the tool to the language While as described by Nielson and Nielson [3] has revealed that their semantic specifications are not always well-typed. Invisible constructors have been proposed in this paper as a method to represent disjoint unions without affecting the conventional rendering. Labelled transition systems have been used to represent typed relations. This amounts to introducing a certain amount of notational baggage in the semantic specifications, in order to typecheck the specifications. It has been demonstrated that the publication quality rendering of the same specifications does not have to suffer.

A purely human readable specification may be ambiguous and incomplete. A machine readable specification does not afford such leniency. The successful use of a tool to render, execute and trace a semantic specification requires the user to fully understand the subject matter, in order to be able to resolve ambiguities and fill in missing detail. As such the use of LETOS provides a stimulating method to explore one's understanding of the subject matter.

LETOS issues a warning when a rule is not source-dependent. This helps writing operational semantics that can be extended, without having to redo any of the proofs that were done for the original system. This helps to create modular semantic specifications.

ACKNOWLEDGEMENTS

The help of the anonymous referees, Marjan Alberda, Krzysztof Apt, Marcel Beemster, Michael Butler, Sandro Etalle, Hugh Glaser, Andy Gravell, Wan Fokkink, Kristian Jaldemark, Paul Klint, Hugh McEvoy, Jon Mountjoy, Mikael Pettersson, Chris Verhoef and Eelco Visser is gratefully acknowledged. Chris Verhoef coined the acronym LETOS, and suggested the implementation of the source-dependency check. Mikael Pettersson helped with the benchmarking.

LETOS is available from the web site:
`http://www.ecs.soton.ac.uk/~phh/letos.html`

REFERENCES

1. Centaur team. Centaur tutorial. Research report, INRIA Rocquencourt, France, July 1994.
2. A. van Deursen, J. Heering and P. Klint, *Language Prototyping: An Algebraic Specification Approach*, AMAST* Series in Computing, World Scientific, Singapore, 1996.
3. H. R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*, John Wiley & Sons, Chichester, UK, 1991.

4. D. A. Turner, 'Miranda: A non-strict functional language with polymorphic types', in J.-P. Jouannaud (ed), *2nd Int. Conf. Functional Programming Languages and Computer Architecture, LNCS 201*, Springer-Verlag, Berlin, 1985, pp. 1–16.
5. S. Stepney, *High Integrity Compilation: A Case Study*, Prentice Hall, Hemel Hempstead, UK, 1993.
6. M. Pettersson, 'Compiling natural semantics', PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 1995. (ftp.ida.liu.se/pub/labs/pelab/rml/papers/thesis.ps.gz)
7. T. B. Dinesh and S. Üsküdarlı, 'While semantics in ASF+SDF', in T. B. Dinesh and S. Üsküdarlı (eds), *Using the ASF+SDF Meta-environment for Teaching Computer Science*, CWI, Amsterdam, August 1994, pp. 109–124. (ftp.wins.uva.nl/pub/gipe/drafts/)
8. T. Despeyroux, 'Typol, a formalism to implement natural semantics', Research report 94, INRIA Sophia Antipolis, March 1988.
9. I. Attali and D. Parigot, 'Integrating natural semantics and attribute grammars: the Minotaur system', Research Report 2339, INRIA Sophia Antipolis, September 1994.
10. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual, 'Centaur: the system', *Third Annual Symposium on Software Development Environments (DE3)*, Boston, USA, 1988, pp. 14–24.
11. M. Pettersson, 'A compiler for natural semantics', in T. Gyimothy (ed.), *6th Int. Conf. Compiler Construction (CC), LNCS 1060*, Springer-Verlag, Berlin, April 1996, pp. 177–191. (ftp.ida.liu.se/pub/labs/pelab/papers/cc96rml.ps.gz)
12. C. Verhoef, 'A congruence theorem for structured operational semantics with predicates and negative predicates', *Nordic J. of Computing*, **2**(2), 274–302 (Summer 1995).
13. W. Fokkink and C. Verhoef, 'A conservative look at term deduction systems with variable binding', Logic group preprint series 140, Department of Philosophy, University of Utrecht, September 1995.
14. Xiaoping Jia, 'ZTC: A type checker for Z – user's guide', Department of Computer and Information Science, DePaul University, Chicago, Illinois, May 1995. (ftp.comlab.ox.ac.uk/pub/Zforum/ZTC-1.3/guide.ps.Z)
15. M. A. Hewitt, C. M. O'Halloran and C. T. Sennett, 'Experiences with PiZa, a Z animator', in J. P. Bowen, M. G. Hinchley and D. Till (eds.), *10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, Reading, UK, April 1997, pp. 37–51. (www.cs.reading.ac.uk/zum97/)
16. E. Best, *Semantics of Sequential and Parallel Programs*, Prentice Hall, Hemel Hempstead, UK, 1996.
17. J. C. Mitchell, *Foundations of Programming Languages*, MIT Press, Cambridge, MA, 1996.
18. G. Winksel, *The Formal Semantics of Programming Languages*, MIT Press, Cambridge, MA, 1993.
19. B. Kirkerud, *Programming Language Semantics*, International Thomson Computer Press, London, UK, 1997.
20. A. Pettorossi and M. Proietti, 'Transformation of logic programs', in D. Gabbay, C. J. Hogger and J. A. Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5 Logic Programming*, Oxford University Press, 1996, pp. 200–299.
21. CEN European Committee for Standardization, 'Identification card systems – inter–sector electronic purse', Draft standard prEN 1546, European Committee for Standardization, Brussels, November 1995.
22. G. Hutton, 'Higher-order functions for parsing', *J. Functional Programming*, **2**(3), 323–343 (July 1992).
23. W. G. Vree and P. H. Hartel, 'Communication lifting: fixed point computation for parallelism', *J. Functional Programming*, **5**(4), 549–581 (October 1995). (ftp.wins.uva.nl/pub/computer-systems/functional/reports/JFP_communication_lifting.ps.Z)
24. C. A. Gunter, *Semantics of Programming Languages*, MIT Press, Cambridge, MA, 1992.
25. T. Johnsson, 'Attribute grammars as a functional programming paradigm', in G. Kahn (ed.), *3rd Int. Conf. Functional Programming Languages and Computer Architecture, LNCS 274*, Portland, Oregon, September 1987, pp. 154–173. Springer-Verlag, Berlin.
26. H. Abramson and V. Dahl, *Logic Grammars*, Springer-Verlag, Berlin, 1989.
27. J. F. T. Kamperman, 'Compilation of term rewriting systems', PhD thesis, Faculty of Mathematics, Computer Science, Physics and Astronomy, University of Amsterdam, September 1996.
28. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, Englewood Cliffs, NJ, 1987.
29. J. F. Groote and F. W. Vaandrager, 'Structured operational semantics and bisimulation as a congruence', *Information and Computation*, **100**(2), 202–260 (October 1992).
30. J. C. M. Baeten and C. Verhoef, 'Concrete process algebra', in S. Abramsky, D. M. Gabbay and T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science, Volume IV*, Oxford University Press, 1995, pp. 149–268.

31. W. J. Fokkink and C. Verhoef, 'Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories', in J.-P. Finance (ed.), *2nd Conf. on Fundamental Approaches to Software Engineering, LNCS 1577*, Amsterdam, The Netherlands, 1999, pp. 98–113. Springer-Verlag, Berlin.
32. P. H. Hartel, M. J. Butler and M. Levy, 'The operational semantics of a Java secure processor', in J. Alves-Foss (ed.), *Formal Syntax and Semantics of Java*, *LNCS 1523*, Springer-Verlag, 1999, pp. 313–352. (www.dsse.ecs.soton.ac.uk/techreports/98-1.html)
33. R. Milner, J. Parrow and D. Walker, 'A calculus of mobile processes part 1', *Information and Computation*, **100**(1), 1–40 (September 1992).
34. P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche and B. Walters, 'Questions and answers about ten formal methods', in S. Gnesi and D. Latella (eds.), *4th Int. Workshop on Formal Methods for Industrial Critical Systems, Vol II*, Trento, Italy, July 1999, pp. 179–203. (www.dsse.ecs.soton.ac.uk/techreports/99-1.html)
35. P. Hartel, J. Hill and M. Sims, 'An operational model of QuickPay', in J.-J. Quisquater and B. Schneier (eds.), *3rd Int. Conf. Smart Card Research and Advanced Application (CARDIS 1998)*, *LNCS*, Louvain la Neuve, Belgium, September 1998. Springer-Verlag, Berlin. (www.dsse.ecs.soton.ac.uk/techreports/98-4.html)
36. M. I. Alberda, P. H. Hartel and E. K. de Jong Frz, 'Using formal methods to cultivate trust in smart card operating systems', *Future Generation Computer Systems*, **13**(1), 39–54 (July 1997). (ftp.wins.uva.nl/pub/computer-systems/functional/reports/FGCS_formal_methods.ps.Z)
37. J. Mountjoy, P. H. Hartel and H. Corporaal, 'Modular operational semantic specification of transport triggered architectures', in C. Delgado Kloos and E. Cerny (eds.), *13th IFIP WG 10.5 Conf. on Computer Hardware Description Languages and their Applications*, Toledo, Spain, April 1997, pp. 260–279. (ftp.wins.uva.nl/pub/computer-systems/functional/reports/CHDL97_specification.ps.Z)
38. S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain and P. L. Wadler, 'The Glasgow Haskell compiler: a technical overview', in *Joint Framework for Information Technology (JFIT) Technical Conf.*, Keele, UK, March 1993, pp. 249–257.
39. P. L. Wadler, 'How to replace failure by a list of successes, a method for exception handling, backtracking, and pattern matching in lazy functional languages', in J.-P. Jouannaud (ed), *2nd Int. Conf. Functional Programming Languages and Computer Architecture*, *LNCS 201*, Nancy, France, September 1985, pp. 113–128. Springer-Verlag, Berlin.
40. R. Milner, *Communication and Concurrency*, Prentice Hall, Hemel Hempstead, UK, 1989.
41. R. Milner, 'A theory of type polymorphism in programming', *J. Computer and System Science*, **17**(3), 348–375 (December 1978).
42. J. A. Robinson, 'A machine-oriented logic based on the resolution principle', *JACM*, **12**(1), 23–41 (January 1965).
43. H. P. Barendregt, 'Lambda calculi with types', in S. Abramsky, D. M. Gabbai and T. S. E. Maibaum (eds), *Handbook of Logic in Computer Science, Volume II Background: Computational Structures*, Oxford, 1992, pp. 117–309. Oxford University Press, UK.
44. L. B. Sherrell and D. L. Carver, 'FunZ: An intermediate specification language', *The Computer J.*, **38**(3), 193–206 (1995).
45. D. A. Watt, 'Executable semantic descriptions', *Software—Practice and Experience*, **16**(1), 13–43 (January 1986).
46. K. Slonneger, 'Executing continuation semantics: a comparison', *Software—Practice and Experience*, **23**(12), 1379–1397 (December 1993).
47. R. Bornat and R. Sufrin, 'Jape: A calculator for animating proof-on-paper', in W. McCune (ed.), *14th Int. Conf. on Automated Deduction (CADE)*, *LNCS 1249*, Townsville, North Queensland, Australia, July 1997, pp. 412–415. Springer Verlag, Berlin.
48. M. Dam and F. Jensen, 'Compiler generation from relational semantics', in B. Robinet and R. Wilhelm (eds.), *1st European Symp. on Programming (ESOP)*, *LNCS 213*, Saarbrücken, West Germany, March 1986, pp. 1–29. Springer-Verlag, Berlin.
49. A. M. Gravell and P. Henderson, 'Executing formal specifications need not be harmful', *Software Engineering J.*, **11**(2), 104–110 (March 1996).
50. A. C. Winstanley and D. W. Bustard, 'EXPOSE: an animation tool for process-oriented specifications', *Software Engineering J.*, **6**(6), 463–475 (November 1991).
51. J. Grundy and T. Långbacka, 'Towards a browsable record of HOL proofs', TUCS Technical Reports 7, Turku Center for Computer Science, Turku, Finland, May 1996.
52. L. Lamport, 'How to write a proof', *The American Mathematical Monthly*, **102**(7), 600–608 (August 1995).

