LATOS – A Lightweight Animation Tool for Operational Semantics

Pieter H. Hartel
phh@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-97-1
April 30 1997

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

LATOS – A Lightweight Animation Tool for Operational Semantics

Pieter H. Hartel *

May 1, 1997

Abstract

A simple tool is proposed to aid in the development of operational semantics. The tool supports publication quality rendering using LaTeX, execution and animation using a functional programming system, and derivation tree browsing using Netscape. The tool has been implemented and it has been used on a large subset of the Java Virtual Machine, as well as a number of relatively small languages. The tool helps to check that a specification is operationally conservative.

1 Introduction

To design and specify the semantics of a programming language involves a number of clerical tasks, such as rendering, type-checking, execution and animation. Tools help to perform these clerical tasks quickly and accurately and give the designer increased confidence in the usefulness 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, large and powerful systems such as Centaur [26] and ASF+SDF [27] provide rendering, type-checking, execution, animation 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 [19], who use Miranda [1] to execute their semantics, and Stepney [25], who uses Prolog to specify her semantics. The RML system [20] covers middle ground in that it offers only one facility: the compilation of operational semantics specifications into C. Concentrating on one aspects 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 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 [6] of Centaur is slow [20]. Recent work on the Minotaur [3] version of Typol has shown that imposing restrictions on the full generality of the Typol 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 animation facilities does not make for a user friendly system [20, p. 180].

The systems referred to above have one factor in common: they are all large. For example, in 1988 the Centaur system was reported to consist of 32k lines of code [4] and in 1996 the RML compiler and runtime system together comprised 15k lines of code [21].

The approach advocated in this paper is to build a lightweight tool that offers the most important facilities at a minimal cost. This is achieved by making maximum use of existing components, and by judiciously selecting essential features. The proposed `latos` program is small (2k lines of lex, yacc and C). It takes as input a superset of Miranda. `Latos` is capable of producing a proper declarative program, which can be type-checked and executed by the appropriate language system. We use Miranda and Haskell in the paper, but using Prolog would not pose difficulties. `Latos` can also produce a LaTeX script, that when typeset provides the conventional rendering of a semantics. Support for animation is provided by the ability to produce HTML files representing derivation trees. Such derivations can be rendered nicely by Netscape.

The practical contribution of this paper is to show that it is possible to build a useful tool with simple means and a number of sensible engineering choices. Also we demonstrate how using the tool helps to gain insight in the ambiguities of a semantic specification.

In the literature several *formats* [28] have been proposed that impose a number of syntactic constraints on the operational rules. These formats can be shown to endow an operational semantics with one or more useful properties, such as operational conservativity [8]. A `latos`

*Dept. of Electronics and Computer Science, Univ. of Southampton, S017 1BJ Southampton, UK, Email: phh@ecs.soton.ac.uk

[1] Miranda is a trademark of Research Software Ltd.

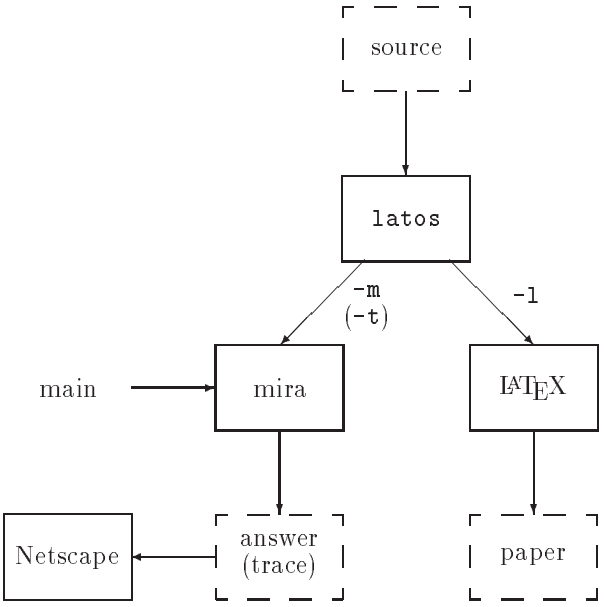


Figure 1: The `latos` architecture.

specification is an operational semantics that is machine readable. `Latos` can thus be used as a vehicle for implementing the various formats. The current version of `latos` warns if an operational rule is not *source dependent*, (See Section 4). This is a necessary condition for operational conservativity.

`Latos` can be used for non-deterministic specifications with one restriction: an animation will only deliver one out of many possible results. All other features of `latos`, such as type checking, source dependency checking and type setting are available for deterministic as well as non-deterministic specifications.

The next section introduces `latos` architecture. Section 3 describes the input format, using as a running example, the language While from the textbook by Nielson and Nielson [19]. The translation from sets of rules and axioms into a functional program is discussed in Section 4. An assessment of the tool is given in Section 5. Related work is discussed in Section 6. The last section presents the conclusions.

2 The `latos` architecture

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

- The `latos` source follows the literate programming convention in the tradition of the UNIX TROFF tools. 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.
- `Latos` 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.
- `Latos` 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 generated Miranda program can be supplied with appropriate input and deliver a behaviour.
- 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 `latos`. As an example we show how a web page is created, which can be browsed interactively by Netscape.

3 The `latos` input language

The `latos` 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 rules of inference.
- Addition of 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.

`Latos` 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 `latos` have in common. This represents a significant difference between `latos` and other tools, which generally fix the semantics of the entire specification language.

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 [19]. This is the subject of the remainder of this section.

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 (ie. a list of characters). It is worth noting that Miranda's primitive number type provides arbitrary precision integer arithmetic.

$$
\begin{array}{ll}
\mathrm{n} \equiv & \mathrm{num}; \\
\mathrm{x} \equiv & \mathrm{string}; \\
\mathrm{a} ::= & \mathrm{n} \mid \mathrm{x} \mid \\
& \mathrm{a} + \mathrm{a} \mid \mathrm{a} \star \mathrm{a} \mid \mathrm{a} - \mathrm{a}; \\
\mathrm{b} ::= & \texttt{true} \mid \texttt{false} \mid \mathrm{a} = \mathrm{a} \mid \mathrm{a} \leq \mathrm{a} \mid \\
& \neg\, \mathrm{b} \mid \mathrm{b} \wedge \mathrm{b}; \\
\mathrm{S} ::= & \mathrm{x} := \mathrm{a} \mid \texttt{skip} \mid \mathrm{S}\ ; \mathrm{S} \mid \\
& \texttt{if}\ \mathrm{b}\ \texttt{then}\mathrm{S}\ \texttt{else}\mathrm{S} \mid \\
& \texttt{while}\ \mathrm{b}\ \texttt{do}\mathrm{S};
\end{array}
$$

The differences between the rendering above and that of Nielson and Nielson [19, Page 7] are as follows. Firstly, 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.

`Latos` input

In the literate programming convention of `latos`, 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 it 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 ;

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

The next two sections discuss the details of the `latos` 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 `latos` input and can therefore be varied, for example to center the middle column.

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

`Latos` offers two simple "editing" facilities. Firstly, 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. Secondly, in the LaTeX output identifiers beginning with `macro_` or `MACRO_` are replaced by LaTeX macro calls. The macros `\Add` and `\S` demonstrate how this facility is used.

Miranda output

The `latos` 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 statement type `macro_S` uses dist-fix notation, which is not supported by Miranda. `Latos` implements dist-fix constructions by generating a new type for each constructor except the first. For example, the Miranda generated for the statement type is shown below. (The apostrophe ’ is a legal character in a Miranda identifier).

```
macro_S ::= x $Ass a | Skip |
            macro_S $Comp macro_S |
            If b then’ macro_S else’ macro_S |
            While b do’ macro_S ;
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 two separate definitions to emphasise the structure.

```
fact  :: S;
fact  = ("z":= 1);(while(¬("y"= 1))do body);
body = ("z":=("z"⋆"y"));("y":=("y"− 1));
```

The differences between the rendering above and that of Nielson and Nielson [19, Page 8] are twofold. Firstly, extra parentheses have been inserted to make the expression unambiguous. Secondly, the variables above are represented as strings (cf. type x), and therefore shown adorned by double quotes. If this becomes unwieldy, it would be possible to represent identifiers as in Nielson and Nielson, for example:

```
fact  = (z := 1);(while(¬(y = 1))dobody);
body = (z :=(z ⋆ y));(y :=(y − 1));
y     = "y";
z     = "z";
```

3.1.2 Alternative approaches

`Latos` allows for an abstract syntax to be defined that quite closely resembles a concrete syntax. There are two alternatives to this approach. The first is exemplified by the Syntax Definition Formalism (SDF) of the ASF+SDF system [27]. This system permits arbitrary context free productions to be defined, and the integrated Algebraic Specification Formalism (ASF) allows equations to be written over these syntactic constructs. SDF also supports scanning and parsing of programs, for which `latos` provides no support.

The other alternative is exemplified by Stepney's work [25, Page 168]. She uses the free type notation of Z to define the abstract syntax of languages.

Our approach is a compromise in the sense that input facilities are primitive, but the rendering is the same as that used in text books on the subject. 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 [13]. This could be achieved without adding further facilities to `latos`, see for example our earlier work [29].

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.

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

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

$$\begin{aligned}&\text{s}_3 :: \text{state};\\ &\text{s}_3 = \{\langle \text{y}\mapsto 3\rangle\};\end{aligned}$$

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

$$\begin{aligned}\mathcal{A} &:: \text{a}\to\text{state}\to\mathbb{Z};\\ \mathcal{A}[\![\text{n}]\!]\text{s} &= \mathcal{N}[\![\text{n}]\!];\\ \mathcal{A}[\![\text{x}]\!]\text{s} &= \text{s}\{\text{x}\};\\ \mathcal{A}[\![\text{a}_1 + \text{a}_2]\!]\text{s} &= \mathcal{A}[\![\text{a}_1]\!]\text{s} + \mathcal{A}[\![\text{a}_2]\!]\text{s};\\ \mathcal{A}[\![\text{a}_1 \star \text{a}_2]\!]\text{s} &= \mathcal{A}[\![\text{a}_1]\!]\text{s} * \mathcal{A}[\![\text{a}_2]\!]\text{s};\\ \mathcal{A}[\![\text{a}_1 - \text{a}_2]\!]\text{s} &= \mathcal{A}[\![\text{a}_1]\!]\text{s} - \mathcal{A}[\![\text{a}_2]\!]\text{s};\end{aligned}$$

There is a fundamental difference between an executable specification as given above and a mathematical specification as given by Nielson and Nielson [19, Page 13]. 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:

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

In an executable specification, execution would simply be terminated when an identifier cannot be mapped onto a value. We will see another instance of the same difference in Section 3.3.1.

`Latos` input

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

```
\newcommand{\bbz}{\bbB{B}}
.MS
macro_bbZ == num ;
state == { <x|->macro_bbZ> } ;
.ME
```

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

```
macro_calA [[‘V x]] s = s{/x/} ;
```

In addition to some minor differences (dealing with the `macro_` 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 `latos` 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 (`latos` 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 `latos` 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 it requires no new features. We just give the type of the function:

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

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

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

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 axioms and rules of inference. 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.

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

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

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

$$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}}]\ \frac{\vdash \langle \mathrm{s}_1,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s'}}{\begin{array}{l}\vdash \langle \mathtt{if}\ \mathrm{b}\ \mathtt{then}\ \mathrm{s}_1\ \mathtt{else}\ \mathrm{s}_2,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s'},\\ \textbf{if}\ \mathcal{B}[\![\mathrm{b}]\!]\mathrm{s} = \mathrm{tt};\end{array}}$$

$$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}]\ \frac{\vdash \langle \mathrm{s}_2,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s'}}{\begin{array}{l}\vdash \langle \mathtt{if}\ \mathrm{b}\ \mathtt{then}\ \mathrm{s}_1\ \mathtt{else}\ \mathrm{s}_2,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s'},\\ \textbf{if}\ \mathcal{B}[\![\mathrm{b}]\!]\mathrm{s} = \mathrm{ff};\end{array}}$$

$$[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}]\ \frac{\vdash \langle \mathrm{s}_1,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s'}, \vdash \langle \mathtt{while}\ \mathrm{b}\ \mathtt{do}\ \mathrm{s}_1,\ \mathrm{s'}\rangle \xrightarrow{1} \mathrm{s''}}{\begin{array}{l}\vdash \langle \mathtt{while}\ \mathrm{b}\ \mathtt{do}\ \mathrm{s}_1,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s''},\\ \textbf{if}\ \mathcal{B}[\![\mathrm{b}]\!]\mathrm{s} = \mathrm{tt};\end{array}}$$

$[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}]\quad \vdash \langle \mathtt{while}\ \mathrm{b}\ \mathtt{do}\ \mathrm{s}_1,\ \mathrm{s}\rangle \xrightarrow{1} \mathrm{s},$
$\textbf{if}\ \mathcal{B}[\![\mathrm{b}]\!]\mathrm{s} = \mathrm{ff};$

There are two differences between the representation above and that of Nielson and Nielson[19, Page 20]. Firstly the notation above for substitution using curly brackets makes it explicit that a mapping is a set. This also overloads the curly brackets in that $s\{x\}$ represents lookup but $s\{x \mapsto v\}$ represents substitution. Secondly, the arrows have been labeled, 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:

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

Making the type of a relation unambiguous is necessary for the specification to be type checked mechanically.

`Latos` input

The `latos` input language allows axioms and rules to be defined at the top level, together with data type and function definitions. Here is the source language representation for the $\mathrm{ass}_{\mathrm{ns}}$ axiom.

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

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

Here is the `latos` specification for the composition rule:

```
rule comp_ns =
|- <s_1, s> =1=> s', |- <s_2, s'> =1=> s''
-----------------------------------------
|- <s_1 $Comp s_2, s> =1=> s'' ;
```

As usual, the premises are written above the dashed line, and the conclusion is written below it. It is possible to precede the premises and the conclusion by a list of assumptions. The turnstile symbol separates the two. None of the rules in this paper need assumptions; they will not be discussed here. `Latos` can handle assumptions appropriately.

Some axioms and rules have a side condition. In the `latos` language this is indicated as follows:

```
axiom while_ns^ff =
|- <While b Do s_1, s> =1=> s,
if macro_calB [[b]] s = ff ;
```

An axiom or rule with a side condition **if** E_b can always be replaced with a rule with one (one more) premise(s). The new premise would then be of the form $\text{E}_\text{b} \stackrel{\text{id}}{\rightarrow} \text{True}$. We will elaborate this in Section 4.

Miranda output

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 [30] can be used to simulate a relation. When given a relation $\text{R} :: \text{A} \leftrightarrow \text{B}$, this method creates a function F as follows:

$$F \quad :: A \rightarrow \{B\};$$
$$F\ \text{a'} = \{\text{b} \mid \langle \text{a} \mapsto \text{b} \rangle \leftarrow R \wedge\ \text{a} = \text{a'}\};$$

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 semantics there might be several successes. An animation produced by `latos` for a non-deterministic specification will only deliver one result. This restriction could easily 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. However, the need has not arisen to lift this restriction as yet. (See also Section 5).

In the Miranda output, a transition system with a label (`1` say) is represented by a set of functional clauses with the name `rule_1`. For each rule or axiom 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 or axioms in the transition system apply, otherwise a singleton list results.

The generated Miranda for the ass_{ns} axiom above is:

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

The translation of a rule into a function must take the premises into account. Consider the rule comp_{ns} below. 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 [(s_1 $Comp s_2,s)]
 = [s''],if non_empty t_1 /\ non_empty t_2
   where
   s'  = last t_1 ;
   t_1 = rule_1 [(s_1,s)] ;
   s'' = last t_2 ;
   t_2 = rule_1 [(s_2,s')] ;
   ;
```

At the end of the 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 and axioms into functions will be discussed in more detail in Section 4.

3.3.1 Animating the natural semantics

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

$$\text{S}_{\text{ns}} \quad :: \text{S} \rightarrow \text{state} \rightarrow [\text{state}];$$
$$\text{S}_{\text{ns}}[\![\text{S}]\!]\text{s} = (\langle \text{S},\ \text{s} \rangle \stackrel{1}{\rightarrow});$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{[\mathrm{ass}_{\mathrm{ns}}] \vdash \langle \mathrm{z} := 1,\ \mathrm{s}_3\rangle \xrightarrow{1} \mathrm{s}_{31}}
\qquad
\dfrac{}{[\mathrm{ass}_{\mathrm{ns}}] \vdash \langle \mathrm{z} := (\mathrm{z} \star \mathrm{y}),\ \mathrm{s}_{31}\rangle \xrightarrow{1} \mathrm{s}_{33}}
\quad
\dfrac{}{[\mathrm{ass}_{\mathrm{ns}}] \vdash \langle \mathrm{y} := (\mathrm{y} - 1),\ \mathrm{s}_{33}\rangle \xrightarrow{1} \mathrm{s}_{23}}
}{[\mathrm{comp}_{\mathrm{ns}}] \vdash \langle \mathrm{body},\ \mathrm{s}_{31}\rangle \xrightarrow{1} \mathrm{s}_{23};}
\qquad
\dfrac{}{[\mathrm{ass}_{\mathrm{ns}}] \vdash \langle \mathrm{z} := (\mathrm{z} \star \mathrm{y}),\ \mathrm{s}_{23}\rangle \xrightarrow{1} \mathrm{s}_{26}}
\quad
\dfrac{}{[\mathrm{ass}_{\mathrm{ns}}] \vdash \langle \mathrm{y} := (\mathrm{y} - 1),\ \mathrm{s}_{26}\rangle \xrightarrow{1} \mathrm{s}_{16}}
}{[\mathrm{comp}_{\mathrm{ns}}] \vdash \langle \mathrm{body},\ \mathrm{s}_{23}\rangle \xrightarrow{1} \mathrm{s}_{16};}
\qquad
\dfrac{}{[\mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}] \vdash \langle \mathtt{while}(\neg(\mathrm{y} = 1))\mathtt{do}\ \mathrm{body},\ \mathrm{s}_{16}\rangle \xrightarrow{1} \mathrm{s}_{16}}
}{[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}] \vdash \langle \mathtt{while}(\neg(\mathrm{y} = 1))\mathtt{do}\ \mathrm{body},\ \mathrm{s}_{23}\rangle \xrightarrow{1} \mathrm{s}_{16};}
}{[\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}] \vdash \langle \mathtt{while}(\neg(\mathrm{y} = 1))\mathtt{do}\ \mathrm{body},\ \mathrm{s}_{31}\rangle \xrightarrow{1} \mathrm{s}_{16};}
}{[\mathrm{comp}_{\mathrm{ns}}] \vdash \langle \mathrm{fact},\ \mathrm{s}_3\rangle \xrightarrow{1} \mathrm{s}_{16};}
$$

Figure 2: Proof associated with computing the factorial of 3. The notation s_{jk} is an abbreviation for $\{\langle \mathrm{y} \mapsto \mathrm{j}\rangle, \langle \mathrm{z} \mapsto \mathrm{k}\rangle\}$ and $\mathrm{s}_3 = \{\langle \mathrm{y} \mapsto 3\rangle\}$.

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

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

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 a sample state s_3 given earlier, the following equality is proved:

$$
\mathrm{S}_{\mathrm{ns}}\ \mathrm{fact}\ \mathrm{s}_3 = [\{\langle \mathrm{y} \mapsto 1\rangle,\ \langle \mathrm{z} \mapsto 6\rangle\}]
$$

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

Miranda output

The Miranda generated for function `macro_Sns` 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_S -> state -> [state] ;
macro_Sns macro_S s = rule_1 [(macro_S, s)] ;
```

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

3.3.2 Browsing a derivation tree

The semantic functions and the axioms and rules of the natural semantics can be used to prove that the above equality 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. 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 a rule is used, the premises are shown, each on a separate line, to be followed by a horizontal line and then a conclusion. The conclusion is labeled by the name of the rule.

A proof such as that given in Figure 2 is read bottom up. At the bottom of the figure we see a transition labelled $\mathrm{comp}_{\mathrm{ns}}$. 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 $\mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}}$ at the bottom of the figure is valid.

Driven by a command line option (`-t`), `latos` 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 rule $\mathrm{comp}_{\mathrm{ns}}$ is shown below,

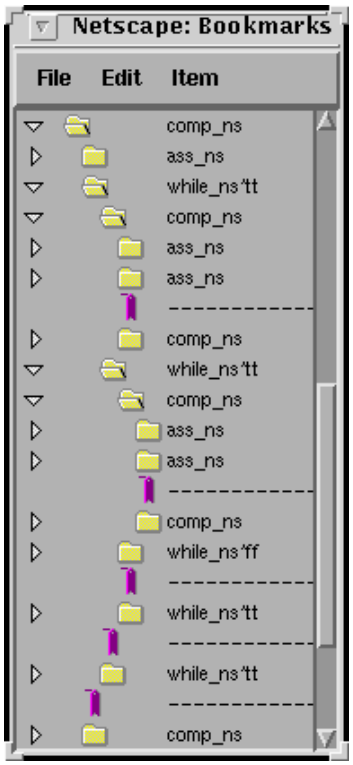
with tracing code added automatically. Remembering the derivation, inclusive of all intermediate states, involves a change of type for the function `rule_1` from producing a list of values of type `state` to producing a list of pairs: `(state,string)`.

```
rule_1 :: [(macro_S,state)]->[(state,string)] ;
rule_1 [(s_1 $Comp s_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 [(s_1,s)] ;
   s'' = fst(last t_2) ;
   t_2 = rule_1 [(s_2,s')] ;
   ;
```

The function `trace_1` used above combines the lists of inputs it receives with the label of the current rule. A function such as this should be supplied as part of the `latos` input. The format in which it will produce the results is under control of the user, thus providing for maximal flexibility. It is straightforward to produce something like HTML in this way, but other formats 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:

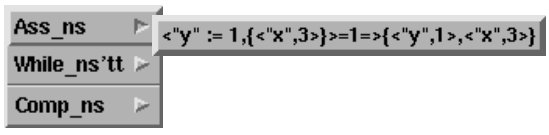
```
<DT><H3>comp_ns</H3>
<DL><p>
    <DT><H3 FOLDED>ass_ns</H3>
        ...
    <DT><H3>while_ns'tt</H3>
    <DL><p>
        <DT><H3>comp_ns</H3>
        <DL><p>
            <DT><H3 FOLDED>ass_ns</H3>
                ...
            <DT><H3 FOLDED>ass_ns</H3>
                ...
            <HR>
   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

A convenient way of browsing this derivation tree is by incorporating it in the file `bookmarks.html` so that Netscape will render it as a normal, quite verbose HTML page and also in various compact forms. Unfortunately, Netscape will not render arbitrary HTML files in compact forms. Below is the compact representation of the derivation tree as shown 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.

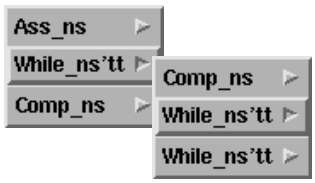


Netscape provides an alternative to browsing the derivation tree by using pull-down menus. Below is the menu corresponding to the conclusion of the derivation. It represents an instance of the comp_{ns} rule. The two derivations to be made for the premises are shown above the line. The name of the rule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

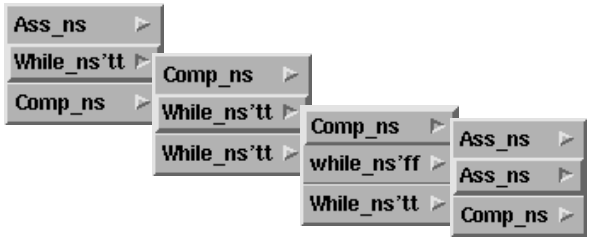
Moving the mouse to the field marked `Ass_ns` displays the derivation subtree associated with the first premise, which represents an instance of the ass_{ns} rule. It shows the effect of executing the first assignment statement on the state of the computation. The screen shot below was actually produced 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 axioms and rules. The SOS of the language While, following Nielson and Nielson, is shown below:

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

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

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

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

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

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

$$[\mathrm{while}_{\mathrm{sos}}] \quad \vdash \langle \mathtt{while}\ \mathrm{b}\ \mathtt{do}\ \mathrm{s}_1,\ \mathrm{s}\rangle \overset{3}{\Rightarrow} \langle \mathtt{if}\ \mathrm{b}\ \mathtt{then}(\mathrm{s}_1\ ;(\mathtt{while}\ \mathrm{b}\ \mathtt{do}\ \mathrm{s}_1))\mathtt{else}\ \mathtt{skip},\ \mathrm{s}\rangle;$$

This specification consists of two separate relations:

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

These relations are different because their types are different. In their specifications, Nielson and Nielson [19, Page 33] ignore the difference, but introduce it again when the animation of the specification in Miranda is discussed [19, Page 217]. 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 rule to the SOS specification. This new relation $\overset{4}{\Rightarrow}$ given below computes the transitive closure of the relation $\overset{3}{\Rightarrow}$, and selects final state:

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

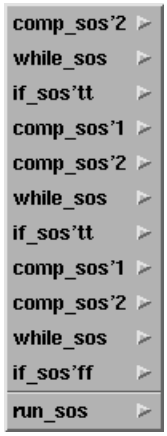
With this preparation, the SOS of While statements can be expressed in the same way as the natural semantics:

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

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

3.4.1 Animating the SOS

The introduction of the extra rule $\overset{4}{\Rightarrow}$ 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 rules (`comp_sos'1` and `comp_sos'2`) have subsidiary derivation trees; others will exhibit just a transition on the configuration.

3.5 Abstract machines

Nielson and Nielson describe a low level abstract machine and a "provably correct implementation" of While [19, Chapter 3]. Both the abstract machine and the compiler have been expressed using the `latos` tool and the associated standard tools. Here is the abstract syntax of the instructions of the machine:

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

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

$$\begin{aligned} &\mathcal{CA} :: \mathrm{a} \rightarrow \mathrm{c}; \\ &\mathcal{CB} :: \mathrm{b} \rightarrow \mathrm{c}; \\ &\mathcal{CS} :: \mathrm{S} \rightarrow \mathrm{c}; \end{aligned}$$

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

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

The first component (c) of a configuration is a sequence of instructions (i); 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:

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

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:

$$\begin{aligned} &\mathrm{S_{am}} :: \mathrm{S} \rightarrow \mathrm{state} \rightarrow [\mathrm{state}]; \\ &\mathrm{S_{am}}[\![\mathrm{S}]\!]\mathrm{s} = (\langle \mathcal{CS}[\![\mathrm{S}]\!],\ [],\ \mathrm{s} \rangle \overset{6}{\triangleright}); \end{aligned}$$

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

⋮ 27 steps omitted

This time the derivation sequence is absolutely flat, cf [12, Page 27]. Moving the mouse to a sub-menu would show the actual transition step.

3.6 Denotational semantics

`Latos` can be used to render a conventional denotational semantics. It also supports the execution of a denotational

semantics. Unfortunately, it does not allow for derivation trees to be generated and browsed. The reason is as follows. With inference rules, 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 inference 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 semantic function. 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 [19, Page 86]:

$$
\begin{array}{ll}
S_{ds} & :: S \rightarrow state \rightarrow state; \\
S_{ds}[\![x := a]\!]s & = s\{x \mapsto \mathcal{A}[\![a]\!]s\}; \\
S_{ds}[\![\texttt{skip}]\!] & = id; \\
S_{ds}[\![s_1 \ ; \ s_2]\!] & = S_{ds}[\![s_2]\!].S_{ds}[\![s_1]\!]; \\
S_{ds}[\![\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!] & = cond(\mathcal{B}[\![b]\!],\ S_{ds}[\![s_1]\!],\ S_{ds}[\![s_2]\!]); \\
S_{ds}[\![\texttt{while}\ b\ \texttt{do}\ s_1]\!] & = FIX\ F \\
 & \quad \textbf{where} \\
 & \quad F\ g = cond(\mathcal{B}[\![b]\!],\ g.S_{ds}[\![s_1]\!],\ id) \\
 & \quad ;
\end{array}
$$

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

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

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

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

A continuation semantics [19, Page 128] is also easy to describe. It does not add to what has been said before.

4 Translating relations

`Latos` translates axioms and rules into functions. The translation is loosely based on Johnsson's method for translating attribute grammars into lazy functional programs [14]. In the present work an axiom plays the role of a terminal, and a rule plays the role of a non-terminal in a grammar. The conclusion of an axiom or 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 a 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 [3]. Stepney [25] uses a similar method based on definite clause translation grammars [1, Chapter 9], which are the logic programming equivalent of attribute grammars.

The following sections discuss the syntax and semantics of the rules and axioms of `latos`. 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 and axioms contain patterns, indicated by the letter P, and expressions, indicated by the letter 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 restricting generality, we ignore here the usual syntactic sugar for lists and infix constructors:

$$
\begin{array}{rl}
P ::= & (P_1,\ \ldots,\ P_t) \mid \\
 & C_n P_1 \ldots P_n \mid \\
 & v;
\end{array}
$$

An expression is 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.

$$
\begin{array}{rl}
E ::= & (E_1,\ \ldots,\ E_t) \mid \\
 & C_n E_1 \ldots E_n \mid \\
 & F_m E_1 \ldots E_m \mid \\
 & v;
\end{array}
$$

The expressions above represent a first order sublanguage 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 and axioms still provide the full power of higher order programming to `latos` specifications. The denotational semantics of Section 3.6 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 take a more liberal view [15].

4.2 Syntax of an axioms and rules

The syntax of an axiom without a side condition is given by the axiom schema below:

$$[\text{schema}_1] \vdash \mathrm{P}_0 \xrightarrow{R} \mathrm{E}_0;$$

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

$$[\text{schema}_2] \vdash \mathrm{P}_0 \xrightarrow{R} \mathrm{E}_0, \quad \textbf{if } \mathrm{E_b};$$

To cope with rules that have a different number of premises a family of rule-schema is used. The syntax of a rule with $n \geq 1$ premises, and without a side condition is given by the appropriate member of the schema$_3$ below:

$$[\text{schema}_3] \frac{\begin{array}{l}\vdash \mathrm{E}_1 \xrightarrow{1} \mathrm{P}_1, \\ \vdots \\ \vdash \mathrm{E}_n \xrightarrow{n} \mathrm{P}_n\end{array}}{\vdash \mathrm{P}_0 \xrightarrow{R} \mathrm{E}_0;}$$

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

$$[\text{schema}_4] \frac{\begin{array}{l}\vdash \mathrm{E}_1 \xrightarrow{1} \mathrm{P}_1, \\ \vdots \\ \vdash \mathrm{E}_n \xrightarrow{n} \mathrm{P}_n\end{array}}{\vdash \mathrm{P}_0 \xrightarrow{R} \mathrm{E}_0, \quad \textbf{if } \mathrm{E_b};}$$

In a subsequent section the syntactic conditions will be given that have to be satisfied for axioms and rules to yield a syntactically correct translation into Miranda. To simplify the presentation the four different kinds of axioms and rules will first be translated into rules of the kind schema$_3$. This is the subject of the following section.

4.3 Simplifying axioms and rules

Each axiom of the form schema$_2$ (ie. with a side condition **if** E$_\mathrm{b}$) is replaced by a rule with a single premise as shown below.

$$[\text{schema}_3] \frac{\vdash \mathrm{E_b} \xrightarrow{id} \mathrm{True}}{\vdash \mathrm{P}_0 \xrightarrow{R} \mathrm{E}_0;}$$

The auxiliary relation $\xrightarrow{id}$ is the (polymorphic) identity relation:

$$[\mathrm{id}] \vdash \mathrm{x} \xrightarrow{id} \mathrm{x};$$

Similarly, each rule of the form schema$_4$ (ie. with a side condition **if** E$_\mathrm{b}$ and n premises) is replaced by a rule of the form schema$_3$ with an extra premise $\mathrm{E_b} \xrightarrow{id} \mathrm{True}$.

As a second simplification the distinction between axioms and rules will be dropped; an axiom will be treated as a 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 fv$(\cdot)$ denote the set of free variables of a given pattern or expression. A rule (after simplification) should satisfy the two syntactic constraints below:

pure A rule should not have free variables:

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

linear No two patterns in a rule should use the same variable, that is if $\mathrm{i} \neq \mathrm{k}$ then:

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

A set of rules and axioms is pure(linear) if all rules and axioms 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 [6]; 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 operational rules. Note that the linearity condition does not require an individual pattern to be linear (Miranda supports such non-linear patterns; most other lazy functional languages do not).

The linearity requirement does not affect operational conservativity. However, it does make it less convenient to specify an operational semantics that essentially uses unification, such as a type checker. In such a case one has to manipulate substitutions explictly and program the unification process [22, Ch. 9]. In the dynamic semantics of the languages that `latos` has been applied to (See Section 5), 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.

$$
\begin{array}{l}
\text{rule}_{\text{R}}[\text{P}_0] = [\text{E}_0], \\
\qquad \textbf{if } \text{non\_empty } \text{t}_1 \wedge \ldots \text{non\_empty } \text{t}_\text{n} \\
\qquad \textbf{where} \\
\qquad \text{P}_1 = \text{last } \text{t}_1; \\
\qquad \text{t}_1 = \text{rule}_1[\text{E}_1]; \\
\qquad \quad \vdots \\
\qquad \text{P}_n = \text{last } \text{t}_\text{n}; \\
\qquad \text{t}_\text{n} = \text{rule}_\text{n}[\text{E}_n]; \\
\qquad ;
\end{array}
$$

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 theoretical work.

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.

`Latos` generates code to support refutable patterns as follows. For each refutable pattern P_i,the test $\text{non\_empty } \text{t}_\text{i}$ is replaced by the test $\text{match}_\text{i } \text{t}_\text{i}$. Furthermore, a new function definition is generated, to decide whether the match succeeds:

$$
\begin{array}{l}
\text{match}_\text{i}[\text{P}_i] \quad = \text{True}; \\
\text{match}_\text{i } \text{other} = \text{False};
\end{array}
$$

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

$$
\begin{array}{ll}
\stackrel{gt}{\rightarrow} & :: ((\text{num},\ \text{num}) \leftrightarrow \text{bool}); \\
& \underline{\vdash \text{x} \stackrel{id}{\rightarrow} 1, \vdash \text{y} \stackrel{id}{\rightarrow} 0} \\
[\text{gt}] & \vdash (\text{x},\ \text{y}) \stackrel{gt}{\rightarrow} \text{True};
\end{array}
$$

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 `latos`, and then processed again to pretty print the code).

$$
\begin{array}{ll}
\text{rule}_{\text{gt}} & :: [(\text{num},\ \text{num})] \rightarrow [\text{bool}]; \\
\text{rule}_{\text{gt}}[(\text{x},\ \text{y})] & = [\text{True}], \\
& \quad \textbf{if } \text{match}_1\ \text{t}_1 \wedge \text{match}_2\ \text{t}_2 \\
& \quad \textbf{where} \\
& \quad \text{t}_1 = \text{rule}_{\text{id}}[\text{x}]; \\
& \quad \text{t}_2 = \text{rule}_{\text{id}}[\text{y}]; \\
& \quad ; \\
\text{rule}_{\text{gt}}\ \text{x} & = [];
\end{array}
$$

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

$$
\begin{array}{ll}
\text{match}_1,\ \text{match}_2 & :: [\text{num}] \rightarrow \text{bool}; \\
\text{match}_1[1] & = \text{True}; \\
\text{match}_1\ \text{other} & = \text{False}; \\
\text{match}_2[0] & = \text{True}; \\
\text{match}_2\ \text{other} & = \text{False};
\end{array}
$$

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 ($\text{match}_1\ \text{t}_1$). 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 ($\text{match}_2\ \text{t}_2$), 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:

$$\vdash \text{E}_\text{i} \stackrel{\text{i}}{\rightarrow} *\text{P}_\text{i}$$

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

$$
\begin{array}{l}
\text{P}_i = \text{last } \text{t}_\text{i}; \\
\text{t}_\text{i} \ = \text{closure } \text{rule}_\text{i}[\text{E}_i];
\end{array}
$$

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.

$$
\begin{array}{l}
\text{closure r s} = \text{s' : closure r[s']}, \\
\qquad \textbf{if } \text{non\_empty ss}; \\
\qquad = [], \\
\qquad \textbf{otherwise} \\
\qquad \textbf{where} \\
\qquad \text{ss} = \text{r s}; \\
\qquad \text{s'} = \text{last ss}; \\
\qquad ;
\end{array}
$$

A transition system is fully defined by a collection of rules and axioms that bear the same label (0 say). Each of the individual rules and axioms is translated into the definition of a function clause as described above. In addition a final function clause is added. The functional clauses together then fully define a function rule_0.

This completes the description of the actions of `latos` to translate a set of axioms and rules into a function definition.

4.5 Operational conservativity

The extension of a set of axioms and rules is *operationally conservative* if provable transitions in the orginal system are the same as those in the extended system. Groote en Vaandrager show that the original system must be pure and *well-founded* [10]. `Latos` specifications are always pure, but they need not be well founded. Rather than reproducing the definition of well-founded here, we give an example of a perfectly legitimate `latos` rule that is not well-founded. The problem is the cyclic dependency between the two hypotheses (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.

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

Well-foundedness is an awkward property to check. Fortunately, Fokkink and Verhoef [8] 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 in P_i are source dependent.
- No other variables are source dependent.

`Latos` 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 [8, Theorem 3.21].

4.6 Proof outline

The partial correctness of the translation from rules to functions can be expressed as follows. Given two configurations l_0 and r_0:

$$\text{rule}_0\ [\text{l}_0] = [\text{r}_0] \quad \Longleftrightarrow \quad (\text{l}_0 \xrightarrow{0} \text{r}_0)$$

The proof is by induction on the structure of the derivation tree.

A rigorous proof would require a formal semantics of the target language Miranda, which is not available. For the specially designed specification notations [27, 26, 5] correctness proofs do exist.

5 Assessment

To demonstrate that `latos` is a useful tool, we first list the specifications that have been built using `latos`. This is followed by a comparison of the performance of the tool with that of RML.

5.1 Functional assessment

`Latos` has been applied to a number of operational semantics specification from various sources to assess its usefulness:

JVM subset (131 axioms and rules, 19 functions) A subset of the Java Virtual Machine specification [17] has been specified as a case study of a realistically sized SOS. The subset omits floating point data types only.

Scil (80 axioms and rules, 14 functions) The secure card instruction language (SCIL) is a threaded code language designed for writing secure and compact smart card operating systems. The core of the scil semantics may be found in previous work [2]. `Latos` 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 axioms and rules, 3 functions) Mini-Freja is a call-by-name pure functional language [20]. The operational semantics for the language is available in Typol and in RML. The `latos` version is a literal translation from the RML version, which itself is a literal translation of the Typol version.

While (40 axioms and rules, 28 functions) The language While has been the running example of this paper. The differences between our version of the various styles of semantic specifications and the same specifications from the literature are minor.

TTA (19 axioms and rules, 3 functions) A transport Triggered architecture (TTA) is a novel kind of

system	Sun 10/40, SuperSPARC, 64MB, Solaris 2.4			Sun 10/41, SUNW, 128MB, Solaris 2.5		
compiler	RML	Sicstus	latos+HBC	latos+HBC	latos+GHC	latos+Miranda
version	1	2.1	0.999.1	0.999.3	2.02	2.020
option	-O2		-O	-O	-O	
time	sec.	sec.	sec.	sec.	sec.	sec.
primes 18	0.22	2.20	3.1	3.4	1.5	58
primes 30	0.87	11.20	12.4	12.9	4.2	266

Table 1: Using the Mini-Freya operational semantics to compare the performance of Typol, RML and latos. Seconds represent user+system time.

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 [18].

Memory manager (9 axioms and rules, 4 functions) An operational semantics of the memory manager of a smart card operating system has been specified using latos.

Pattern matching compiler (34 functions) A compiler for pattern matching functions in a style similar to that found in Peyton Jones [22, Ch. 5] has been specified.

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

5.2 Performance assessment

The latos+Miranda combination is useful during development because it provides fast type checking, and compilation. When fast execution is desired, the generated Miranda can be converted into Haskell, for which good compilers are available. This conversion can be done largely automatically using a simple sed script (it would also be straightforward to extend latos so that it will generate Haskell directly).

To assess the performance of the code generated by latos, the operational semantics for Mini-Freja has been used to compute the first 18 and 30 primes, using the sieve of Eratosthenes. Experiments have been carried out using a number of different compilers and two similar machines; times reported are in seconds user+system time. The results are shown in Table 1. The first three columns originate from Table 10.8 in Mikael Pettersson's thesis [20]. The next three columns report the results obtained after translating the Miranda code generated by latos into Haskell. The last column applies to direct execution of the Miranda code.

The three experiments on the Sun 10/40 were carried out by Mikael Pettersson. The execution times for the Haskell code generated (indirectly) by Latos are comparable to the times obtained with Sicstus Prolog. These times were obtained using the HBC haskell compiler from Chalmers (version 0.999.1, with the -O option selected). RML is about a factor 10 faster than Latos+HBC.

The three experiments on the Sun 10/41 show execution times for the HBC compiler, the latest Glasgow Haskell compiler (GHC, version 2.02) and Miranda (version 2.020). The results show that for this program, the GHC compiler generates code that is over two times faster than the HBC compiler. Miranda executes the code about 20 times slower.

The conclusions from this experiment are:

- The best performance is obtained by the highly optimising RML compiler, which has specifically been designed for executing natural semantics specifications.
- 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 fast execution, at the cost of long compilation times.
- Chosing a lazy functional language as a target works well; the results are comparable to those obtained with a good Prolog compiler.

6 Related work

Animating 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 [23] for a recent survey.

In the domain of programming language specification, considerable effort has been devoted to animation of denotational semantics [31, 25], continuation semantics [24], natural semantics [26], structured operational semantics [5], and algebraic specifications of various styles of semantics [27]. Publication quality rendering always has a high priority.

Dinesh and Üsküdarlı have used the ASF+SDF system to typeset and animate 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 system 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, prototyping and animation of specifications. Gravell and Henderson provide a recent overview of the discussion [9]. 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 animate behaviours.

One report has been found in the literature where HyperCard stacks are used to support animation [32]. In recent work Grundy [11] proposes a HTML package called ProofViews to allow window inference style proofs to be browsed comfortably using Netscape. The idea is attributed to Lamport [16].

7 Conclusions

`Latos` is a small program that makes it possible to enter an operational semantics, to render the specification using LaTeX and to animate 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. Due to its relative simplicity it should be considered a light-weight alternative to its more powerful brethren. In combination with a state-of-the-art compiler the execution times of `latos` specifications are competitive.

Applying the tool to the language While as described by Nielson and Nielson 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 and animate 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 and animate 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 `latos` provides a stimulating method to explore ones understanding of the subject matter.

`Latos` 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 Marjan Alberda, Marcel Beemster, Michael Butler, 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 `latos` and Mikael Pettersson performed the `Latos`+HBC benchmark on his machine.

References

[1] H. Abramson and V. Dahl. *Logic Grammars*. Springer-Verlag, Berlin, 1989.

[2] 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), Jun 1997.

[3] I. Attali and D. Parigot. Integrating natural semantics and attribute grammars: the minotaur system. Research Report 2339, INRIA Sophia Antipolis, Sep 1994.

[4] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*, pages 14–24, Boston, USA, 1988. ACM, New York.

[5] M. Dam and F. Jensen. Compiler generation from relational semantics. In B. Robinet and R. Wilhelm, editors, *1st European symp. on programming (ESOP 86), LNCS 213*, pages 1–29, Saarbrücken, West Germany, Mar 1986. Springer-Verlag, Berlin.

[6] T. Despeyroux. Typol, a formalism to implement natural semantics. Research report 94, INRIA Sophia Antipolis, Mar 1988.

[7] T. B. Dinesh and S. Üsküdarlı. While semantics in ASF+SDF. In T. B. Dinesh and S. Üsküdarlı, editors, *Using the ASF+SDF meta-environment for teaching computer science*, pages 109–124. CWI, Amsterdam, Aug 1994.

[8] W. Fokkink and C. Verhoef. A conservative look at term deduction systems with variable binding. Logic group preprint series 140, Department of Philosophy, Univ. of Utrecht, Sep 1995.

[9] A. M. Gravell and P. Henderson. Executing formal specifications need not be harmful. *Software engineering journal*, 11:104–110, Mar 1996.

[10] J. F. Groote and F. W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and computation*, 100(2):202–260, Oct 1992.

[11] J. Grundy and T. Langbåcka. Towards a browsable record of HOL proofs. TUCS Technical Reports 7, Turku Center for Computer Science, Turku, Finland, May 1996.

[12] C. A. Gunter. *Semantics of programming languages.* MIT Press, Cambridge, Massachusetts, 1992.

[13] G. Hutton. Higher-order functions for parsing. *J. functional programming*, 2(3):323–343, Jul 1992.

[14] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *3rd Functional programming languages and computer architecture, LNCS 274*, pages 154–173, Portland, Oregon, Sep 1987. Springer-Verlag, Berlin.

[15] J. F. T. Kamperman. *Compilation of term rewriting systems.* PhD thesis, Fac. of Math., Comp. Science, Physics and Astronomy. Univ. of Amsterdam, Sep 1996.

[16] L. Lamport. How to write a proof. *The American mathematical monthly*, 102(7):600–608, Aug 1995.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, Reading, Massachusetts, 1996.

[18] J. Mountjoy, P. H. Hartel, and H. Corporaal. Modular operational semantic specification of transport triggered architectures. In *13th IFIP WG 10.5 Conf. on Computer Hardware Description Languages and Their Applications*, Toledo, Spain, Apr 1997. Chapman and Hall.

[19] H. R. Nielson and F. Nielson. *Semantics with applications: A formal introduction.* John Wiley & Sons, Chichester, England, 1991.

[20] M. Pettersson. *Compiling Natural Semantics.* PhD thesis, Dept. of Computer and Information Science, Linköping University, Sweden, 1995.

[21] M. Pettersson. A compiler for natural semantics. In T. Gyimothy, editor, *6th Compiler Construction (CC), LNCS 1060*, pages 177–191. Springer-Verlag, Apr 1996.

[22] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[23] L. B. Sherrell and D. L. Carver. FunZ: An intermediate specification language. *The computer journal*, 38(3):193–206, 1995.

[24] K. Slonneger. Executing continuation semantics: a comparison. *Software—practice and experience*, 23(12):1379–1397, Dec 1993.

[25] S. Stepney. *High integrity compilation: A case study.* Prentice Hall, Hemel Hempstead, England, 1993.

[26] Centaur team. Centaur tutorial. Research report, INRIA Rocquencourt, France, Jul 1994.

[27] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach.* AMAST* Series in Computing, World Scientific, Singapore, 1996.

[28] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative predicates. *Nordic J. of Computing*, 2:274–302, 1995.

[29] W. G. Vree and P. H. Hartel. Communication lifting: fixed point computation for parallelism. *J. functional programming*, 5(4):549–581, Oct 1995.

[30] 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, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 113–128, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

[31] D. A. Watt. Executable semantic descriptions. *Software—practice and experience*, 16(1):13–43, Jan 1986.

[32] A. C. Winstanley and D. W. Bustard. EXPOSE: an animation tool for process-oriented specifications. *Software engineering journal*, 6(6):463–475, Nov 1991.