The $\mathcal{L}$ 1.0 Primer

Christoph Walther
Markus Aderhold
Andreas Schlosser

Fachgebiet Programmiermethodik
Fachbereich Informatik
Technische Universität Darmstadt

http://verifun.org
verifun@informatik.tu-darmstadt.de

Technical Report VFR 06/01
April 4, 2006 (Update June 10, 2006)
Abstract

This document defines syntax and semantics of the functional programming language $\mathcal{L}$ (version 1.0) which is used in $\texttt{VeriFun3.0}$ [Ver] for writing programs and stating lemmas about them. The use of $\mathcal{L}$ is illustrated by several examples and a small case study.
## Contents

1 Preface  

2 Syntax of $\mathcal{L}$  
   2.1 Data Structures  
      2.1.1 Definitions  
      2.1.2 Types and Terms  
      2.1.3 Fixity Declarations  
   2.2 Procedures  
      2.2.1 Definitions  
      2.2.2 Let-Expressions  
      2.2.3 Incompletely Defined Procedures  
   2.3 Lemmas  
      2.3.1 Definitions  
      2.3.2 Connectives  

3 Semantics of $\mathcal{L}$  
   3.1 Semantics of Data Structures and Procedures  
      3.1.1 The Exception Guard  
      3.1.2 The Computation Calculus  
      3.1.3 The Interpreter $eval_P$  
      3.1.4 Termination of $\mathcal{L}$-Programs  
   3.2 Semantics of Lemmas  
   3.3 Domain Procedures  

4 The Tautology Checker - A Case Study  
   4.1 Preliminaries  
   4.2 The Tautology Checking Procedure  
   4.3 Soundness of the Tautology Checking Procedure  
   4.4 Completeness of the Tautology Checking Procedure  

5 Appendix  
   5.1 Character Set  
   5.2 Name Spaces  
   5.3 Naming Conventions  
   5.4 Reserved Words and Symbols  
   5.5 Separation of Symbols  
   5.6 Predefined and Implicitly Defined Procedures and Lemmas  
   5.7 Aliases  


<table>
<thead>
<tr>
<th>Index</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>35</td>
</tr>
</tbody>
</table>
1. Preface

This document describes the functional programming language $\mathcal{L}$ (version 1.0) which is used in the verification tool $\text{VeriFun}3.0$ [Ver] for writing programs and stating lemmas about them. The development of $\mathcal{L}$ is not meant as an attempt to invent just another programming language. There are no features in $\mathcal{L}$ which cannot be found in other languages, and worse, many useful feature of these languages are (still) missing.

In fact, the intention at the beginning of the development was not to define a programming language. It rather was the need for a simple notation to state propositions about data structures and procedures in order to investigate reasoning problems arising when proving these propositions by machine. Since the reasoning problems had been the focus of attention, not much care was given to the notation, which grew up rather rampantly taking the notation used in the former INKA system [BHHW86] as a starting point.

The development of $\text{VeriFun}$ started as a small student project but grew up more successfully than anticipated. New language features were needed to write up more ambitious case studies in order to challenge the verifier. It was a rather obvious choice to incorporate features well known from programming languages. But when we noticed that we were defining our own functional language, it was too late to switch over to a “real” programming language. Simply too much time had been spent to develop the verifier and to build a lot of software. This makes it impossible in an academic environment to restart from scratch to reimplement the whole system for another language. This might apologize why we came up with a further programming language and also answers the question “Why don’t you use a ‘real’ functional language” sometimes heard.

But there is another point: Since our concern is verification, it does not help to use a “real” language containing features the verifier does not support. This would rather mean to define subsets of the “real” language by restricting it to language features the verifier can cope with. But it takes time to set up a verifier to support all these beautiful language features coming with “real” functional languages. So this answers another question sometimes heard: “Why does your language not support mutual recursion, higher-order functions, a more flexible type system, etc.?” This is simply because we first have to develop the reasoning mechanisms for a feature before we can integrate it into the language. And the development of those mechanisms requires numerous experiments and tests, unfortunately always taking much more time than expected.

Having extended our “notation” from time to time with the development of the $\text{VeriFun}$ system, we felt that the time was ripe to give it a name and—more importantly—to present the formal definition for it. So in the sequel, we define syntax and semantics of $\mathcal{L}$, trying to be as brief as possible but as precise as necessary. We also illustrate the use of $\mathcal{L}$ by several examples and present a small case study which—as we hope—helps to become quickly familiar with the language.
2. Syntax of $\mathcal{L}$

2.1. Data Structures

2.1.1. Definitions

A data structure is defined by an $\mathcal{L}$-expression of form

\[
\text{structure } \text{struct}[@V_1, \ldots, @V_l] \triangleq \text{cons}_1(\text{sel}_{1,1} : \text{type}_{1,1}, \ldots, \text{sel}_{1,n_1} : \text{type}_{1,n_1}) \\
\quad \vdots \\
\quad \text{cons}_k(\text{sel}_{k,1} : \text{type}_{k,1}, \ldots, \text{sel}_{k,n_k} : \text{type}_{k,n_k})
\]

(2.1)

where $l \geq 0$, $k \geq 1$ and $n_i \geq 0$ for each $i \in \{1, \ldots, k\}$. The symbol \text{struct} is called a type constructor of arity $l$ and the symbols $@V_1, \ldots, @V_l$ are called type variables.

The $\mathcal{L}$-expression (2.1) defines a type $\text{struct}[@V_1, \ldots, @V_l]$ and must satisfy for each $i \in \{1, \ldots, k\}$ and each $h \in \{1, \ldots, n_i\}$:

- $\text{type}_{i,h}$ is a type built by the type variables $@V_1, \ldots, @V_l$, type $\text{struct}[@V_1, \ldots, @V_l]$ and the type constructors (different from the predefined type bool—see Example 1) defined previously by other data structures,
- the symbols \text{struct}, \text{cons}_i and $\text{sel}_{i,h}$ are pairwise different as well as different from all identifiers that have been previously introduced by data structure, procedure and lemma definitions.

In addition,

- each type variable $@V_j$ is used in one of the types $\text{type}_{i,h}$ different from $\text{struct}[@V_1, \ldots, @V_l]$ for some $i \in \{1, \ldots, k\}$ and some $h \in \{1, \ldots, n_i\}$ at least, and
- $\text{type}_{i,h} \neq \text{struct}[@V_1, \ldots, @V_l]$ must hold for some $i \in \{1, \ldots, k\}$ and each $h \in \{1, \ldots, n_i\}$.

One writes simply $\text{cons}_i$ instead of $\text{cons}_i()$ if $n_i = 0$ as well as $\text{structure } \text{struct} <<= \ldots$ instead of $\text{structure } \text{struct}[@V_1, \ldots, @V_l] <<= \ldots$ if $l = 0$. \text{struct} is called a monomorphic type in such a case, and $\text{struct}[@V_1, \ldots, @V_l]$ is a polymorphic type otherwise.

Example 1.

1. The monomorphic type bool is defined by the $\mathcal{L}$-expression

\[
\text{structure } \text{bool} <<= \text{true}, \text{false}
\]

This data structure is predefined in $\mathcal{L}$. 

4
2. The monomorphic type \( \mathbb{N} \) is defined by the \( \mathcal{L} \)-expression

\[
\text{structure } \mathbb{N} <= 0, + (\cdot : \mathbb{N}) .
\]

This data structure—defining the natural numbers \( \mathbb{N} \)—is predefined in \( \mathcal{L} \). The constructor function symbol \(+\) (denoting the successor function) can be alternatively written as \( \text{succ} \), the selector function symbol \(-\) (denoting the predecessor function) can be alternatively written as \( \text{pred} \) and the type \( \mathbb{N} \) can be alternatively written as \( \text{nat} \).

3. The polymorphic type \( \text{SEXPR}[\@\text{ITEM}] \) is defined by the \( \mathcal{L} \)-expression

\[
\text{structure } \text{SEXPR}[\@\text{ITEM}] <=
\begin{align*}
\text{NIL}, \\
\text{ATOM}(\text{DATA} : \@\text{ITEM}), \\
\text{CONS}(\text{CAR} : \text{SEXPR}[\@\text{ITEM}], \text{CDR} : \text{SEXPR}[\@\text{ITEM}])
\end{align*}
\]

This data structure (related to the s-expressions of Lisp) defines binary trees which hold their data in the leaves.

4. The polymorphic type \( \text{LIST}[\@\text{ITEM}] \) is defined by the \( \mathcal{L} \)-expression

\[
\text{structure } \text{LIST}[\@\text{ITEM}] <=
\begin{align*}
\text{EMPTY}, \\
\text{ADD}(\text{HD} : \@\text{ITEM}, \text{TL} : \text{LIST}[\@\text{ITEM}])
\end{align*}
\]

This data structure defines linear lists, where \( \text{EMPTY} \) stands for the empty list and \( \text{ADD} \) builds a new list \( k \) by placing some new list element \( \text{HD}(k) \) in front of another list \( \text{TL}(k) \).

5. The polymorphic type \( \text{tree}[\@\text{ITEM}] \) is defined by the \( \mathcal{L} \)-expression

\[
\text{structure } \text{tree}[\@\text{ITEM}] <=
\begin{align*}
\text{leaf}(\text{data} : \@\text{ITEM}), \\
\text{node}(\text{descendants} : \text{LIST}[\text{tree}[\@\text{ITEM}]])
\end{align*}
\]

This data structure defines finitely branching trees the inner nodes of which may have different outdegrees and the leaves hold the data.

### 2.1.2. Types and Terms

**Types** are built with type constructors and type variables, where the arity of the type constructors has to be respected. A type containing type variables is called *polymorphic*, and otherwise the type is *monomorphic*. By replacing type variables in a polymorphic type by types, further types are obtained. So, for the data structure definitions of Example 1, \( \text{SEXPR}[\mathbb{N}], \text{LIST}[\mathbb{N}], \text{LIST}[\text{SEXPR}[\mathbb{N}]] \) and \( \text{SEXPR}[\text{SEXPR}[\@\text{ITEM}]] \) are monomorphic types, and the types \( \text{SEXPR}[\text{SEXPR}[\@\text{ITEM}]] \) and \( \text{LIST}[\text{LIST}[\@\text{ITEM}]] \) are polymorphic. However, when building further types, type variables must not be replaced by the type \( \text{bool} \), so e.g. \( \text{SEXPR}[\text{bool}] \) and \( \text{LIST}[\text{bool}] \) are no legal types.\(^2\)

Each data structure definition \( DS \) as given in (2.1) defines a *signature* in terms of types for the function symbols introduced by \( DS \): For each \( i \in \{1, \ldots, k\} \) and each \( h \in \{1, \ldots, n_i\} \):

\(^1\) Note that \( \cdot \) always has to be enclosed in blanks in a type declaration of a selector.

\(^2\) Type \( \text{bool} \) receives special treatment as function symbols \( p : \tau_1 \times \ldots \times \tau_n \to \text{bool} \) adopt the rôle of predicate symbols in \( \mathcal{L} \).
• cons\(_i\) : \text{type}_{i,1} \times \ldots \times \text{type}_{i,n_i} \rightarrow \text{struct}[\@V_1, \ldots, \@V_l]$, called a constructor function symbol,

• \(\ell\) cons\(_i\) : \text{struct}[\@V_1, \ldots, \@V_l] \rightarrow \text{bool}$, called a structure predicate symbol, and

• sel\(_i,n_h\) : \text{struct}[\@V_1, \ldots, \@V_l] \rightarrow \text{type}_{i,n_h}$, called a selector function symbol.

Using this signature, terms \(t\) can be defined and assigned a type \(\tau\) (denoted \(t : \tau\) for short) as usual (see e.g. [CW85]). For example,

• \text{true} and \text{false} are terms of type \text{bool},

• 0, \(+(0)\), \(+(+0)\), \((-0)\), \(+(−0)\), \(+(+0)\) are terms of type \(\mathbb{N}\),

• \text{CONS}(\text{NIL}, \text{NIL})$, \text{CONS}(\text{NIL}, \text{CONS}(\text{NIL}, \text{NIL}))$ are terms of type \text{SEXPR}[\@ITEM],

• \text{CONS}(\text{NIL}, \text{ATOM}(+(0)))$, \text{CONS}(\text{CONS}(\text{NIL}, \text{ATOM}(−0))), \text{NIL})$ are terms of type \text{SEXPR}[\@N].

One may also use numerals to write terms of type \(\mathbb{N}\), where \(n\) stands for \(n\) applications of \(+\) to 0. So e.g., 1 stands for \(+(0)\), 2 for \(+(+(0))\), 3 for \(+(+(+(0)))\) etc.

For building terms, the following function symbols are available in addition for each type \(\tau\) different from a type variable:

• \(\text{=} : \tau \times \tau \rightarrow \text{bool}$, where \(\tau \neq \text{bool}$,

• if\( : \text{bool} \times \tau \times \tau \rightarrow \tau$ and

• case\( : \tau' \times \tau \times \ldots \times \tau \rightarrow \tau$, where \(\tau'$ is some type defined by a data structure with \(k\) constructors.

The symbol \(\text{=}\) denotes equality and is written in infix notation.

Boolean conditionals (also called if-expressions) of type \(\tau\) are written in functional or in procedural notation, viz.

• if\(\{a, b, c\}$, or

• if \(a\) then \(b\) else \(c\) \text{end_if}.

Structural conditionals (also called case-expressions) of type \(\tau\) defined by a data structure with constructors \(\text{cons}_1, \ldots, \text{cons}_k\) are written in functional or in procedural notation, viz.

• case\{\(t; \text{cons}_1' : t_1, \ldots, \text{cons}_k' : t_k\}\},

• case\{\(t; \text{cons}_1' : t_1, \ldots, \text{cons}_h' : t_h, \text{other: } t'\}\},

• case \(t\) of \(\text{cons}_1' : t_1, \ldots, \text{cons}_k' : t_k\) \text{end_case}, or

• case \(t\) of \(\text{cons}_1' : t_1, \ldots, \text{cons}_h' : t_h, \text{other: } t'\) \text{end_case,} \(^3\)

where \{\(\text{cons}_1, \ldots, \text{cons}_k\}\} \subseteq \{\(\text{cons}_1, \ldots, \text{cons}_h\}\}$.

Instead of end_if or end_case, end as well as pairs of parentheses instead of pairs of curly brackets may be used when writing conditional expressions in functional notation.

\(^3\) Note that “;” always has to be enclosed in blanks in a case expression.
2.1.3. Fixity Declarations

Each function symbol in a data structure definition of form (2.1) may be preceded by a fixity declaration of form

- \([\text{prefix}]\)
- \([\text{postfix}]\)
- \([\text{outfix}]\)
- \([\text{infix}]\)
- \([\text{infixl},N]\)
- \([\text{infixr},N]\)

where \(N\), called the precedence of the function symbol, is a numeral different from 0.

**prefix and postfix**

Fixity \(\text{prefix}\) is assumed by default if a fixity declaration is missing and fixity \(\text{postfix}\) places a function symbol behind the list of arguments, i.e. \((\ldots)!\) for a \(\text{postfix}\) function symbol \(!\) as compared to \(!(\ldots)\) if \(!\) is declared \(\text{prefix}\) or no fixity is declared for \(!\) at all.

Constant symbols cannot have a fixity other than \(\text{prefix}\). Also structure predicates \(?\text{cons}\) have always fixity \(\text{prefix}\), except if the constructor \(?\text{cons}\) has fixity \(\text{outfix}\) (see below).

**outfix**

If fixity \(\text{outfix}\) is declared, 2 function symbols (not defined elsewhere) separated by a colon (enclosed in blanks) may (but need not) be used. For example, one may write

\[
\text{structure bintree [@ITEM]} <= \begin{align*}
\bot, \\
.AudioState \less \> (\text{left : bintree [@ITEM]}, \text{key : @ITEM}, \text{right : bintree [@ITEM]})
\end{align*}
\]

and terms of type \(\text{bintree}[\text{N}]\) then are written as

- \(\less \bot, 0, \bot \\rangle\),
- \(\less \less \bot, 0, \bot \\rangle, 1, \bot \\rangle\),
- \(\less \less \bot, 5, \bot \\rangle, 1, \less \bot, 3, \bot \rangle \rangle \rangle \text{ etc.}^4\)

A fixity declaration of form \(\text{[outfix] symbol}\) is treated like \(\text{[outfix] symbol : symbol}\). So if one replaces \(\text{[outfix] \less \rangle}\) by \(\text{[outfix] \rangle}\) in the above definition, terms of type \(\text{bintree}[\text{N}]\) are written as

- \(\| \bot, 0, \bot \|\),
- \(\| \| \bot, 0, \bot \|, 1, \bot \|\),

\(\text{Note that the argument list of an outfix function symbol always has to be enclosed in blanks.}\)
The application of a structure predicate of a constructor \("\text{[outfix]}\) \text{ left : right}\) to an argument \(t\) is written as \("\text{[outfix]} t \text{ right}\). So one writes e.g. \(\text{[outfix]} t \left\langle \right\rangle\) for the constructor of \(\text{bintree}\). In \(\text{case}\)-expressions, one writes \("\text{[outfix]} t \text{ left : right}\) for those constructors, for example

\[
\text{case } t \text{ of } \perp : \ldots , \left\langle \right\rangle : \ldots \text{ end case}
\]

\textbf{infix, infixr and infixl}

These fixity declarations are reserved for \textit{binary} function symbols. One may define, for example,

\[
\text{structure } \text{SEXPR} [\@\text{ITEM}] <= \\
\quad \text{NIL}, \\
\quad \text{ATOM}(\text{DATA} : @\text{ITEM}), \\
\quad \text{[infix]} \circ (\text{CAR} : \text{SEXPR}[@\text{ITEM}], \text{CDR} : \text{SEXPR}[@\text{ITEM}])
\]

and terms of type \(\text{SEXPR}[N]\) then are written as

\[
\bullet \ (\text{NIL} \circ \text{NIL}), \\
\bullet \ (\text{NIL} \circ \text{ATOM}(1)), \\
\bullet \ ((\text{NIL} \circ \text{ATOM}(1)) \circ \text{NIL}) \text{ etc.},
\]

where \("\circ\) has to be enclosed in blanks. If one replaces \([\text{infix]}\) by \([\text{infixl}, 2]\) in the above definition, parentheses may be omitted in left-nested \(\circ\)-terms. For instance, terms of type \(\text{SEXPR}[N]\) now can (but need not) be written as

\[
\bullet \ \text{NIL} \circ \text{NIL}, \\
\bullet \ \text{NIL} \circ \text{ATOM}(1), \\
\bullet \ \text{NIL} \circ \text{ATOM}(1) \circ \text{NIL}
\]

where the latter term is the same as

\[
\bullet \ ((\text{NIL} \circ \text{ATOM}(1)) \circ \text{NIL})
\]

hence different to

\[
\bullet \ (\text{NIL} \circ (\text{ATOM}(1) \circ \text{NIL})).
\]

The precedence of a function symbol, e.g. \(2\) for the constructor \(\circ\) in the example, is used to resolve ambiguities. A term \(a \Diamond b \Box c\) stands for \((a \Diamond b) \Box c\), if the \(\text{infixl}\) function symbol \(\Diamond\) has a greater precedence than the \(\text{infixl}\) function symbol \(\Box\), and it stands for \(a \Diamond (b \Box c)\) if the precedence of \(\Box\) is greater than the precedence of \(\Diamond\). If both function symbols have same precedence, \(a \Diamond b \Box c\) is not a syntactically correct term, hence parentheses must be used to resolve ambiguities explicitly.

Similarly, \(\text{infixr}\) is used to omit parentheses in terms which then are treated as right-nested. Further examples:
Example 2.

1. The polymorphic type pair [@ITEM1, @ITEM2] is defined by the \( \mathcal{L} \)-expression

\[
\text{structure \ pair \ } [@ITEM1, \ @ITEM2] \ <= \\
\quad \text{[infix]} \bullet (\text{[postfix]}_1 : @ITEM1, \ [\text{postfix}]_2 : @ITEM2)
\]

and

\(-\ (2 \bullet 3) \text{ is a term of type } \left[\mathbb{N}, \mathbb{N}\right],\)
\(-\ (1 \bullet (2 \bullet 3)) \text{ is a term of type } \left[\mathbb{N}, \text{pair } \left[\mathbb{N}, \mathbb{N}\right]\right],\)
\(-\ ((1 \bullet 2) \bullet 3) \text{ is a term of type } \left[\text{pair } \left[\mathbb{N}, \mathbb{N}\right], \mathbb{N}\right],\)
\(-\ (1 \bullet (2 \bullet 3))_1 \text{ is a term of type } \mathbb{N}, \text{ and}\)
\(-\ (1 \bullet (2 \bullet 3))_2 \text{ is a term of type pair } \left[\mathbb{N}, \mathbb{N}\right].\)

2. The polymorphic type list [@ITEM] is defined by the \( \mathcal{L} \)-expression

\[
\text{structure \ list \ } [@ITEM] \ <= \\
\quad \emptyset, \ [\text{infix}_r, 100] :: (\text{hd} : @ITEM, \ \text{tl} : \text{list[@ITEM]})
\]

and

\(-\ 1 :: 2 :: 3 :: 4 :: \emptyset \text{ is a term of type } \left[\mathbb{N}\right],\)
\(-\ (1 \bullet 2) :: (3 \bullet 4) :: \emptyset \text{ is a term of type } \left[\text{pair } \left[\mathbb{N}, \mathbb{N}\right]\right], \text{ and}\)
\(-\ (1 :: 2 :: \emptyset) :: (3 :: 4 :: \emptyset) :: \emptyset \text{ is a term of type } \left[\text{list } \left[\mathbb{N}\right]\right].\)

2.2. Procedures

2.2.1. Definitions

A procedure is defined by an \( \mathcal{L} \)-expression of form

\[
\text{function \ proc}(x_1 : \text{type}_1, \ldots, x_k : \text{type}_k) : \text{type} \ <= \ \text{body}_{\text{proc}}
\]  

(2.2)

where \( k \geq 1 \) and for each \( i, j \in \{1, \ldots, k\} : \)

- \( \text{proc} \), called the name of the procedure, is an identifier different from all identifiers that have been previously introduced by data structure, procedure and lemma definitions,
- \( x_i \) is a symbol different from all type constructors and function symbols of the data structure and procedure definitions introduced so far,
- \( x_i \neq x_j \), if \( i \neq j \),
- \( \text{type}_i \) is a type variable or a type built with type constructors different from bool defined previously by some data structures,
- \( \text{type} \) is a type variable or a type built with type constructors defined previously by some data structures, and
each type variable in type also appears in some of the types type

A procedure definition as given in (2.2) extends the signature by a

procedure function symbol proc : type₁ × ⋯ × typeₖ → type.

The symbols xᵢ in (2.2) are called the formal parameters of procedure proc and are used as variable symbols of type typeᵢ. The expression bodyproc is called the body of procedure proc and it is demanded that

- bodyproc is a term of type type using only
  - function symbols of the data structure and procedure definitions introduced so far,
  - the procedure function symbol proc, and
  - the formal parameters xᵢ, such that
- the type of each actual parameter tᵢ is identical to typeᵢ for each occurrence of proc(t₁, ⋯ , tₖ) in bodyproc, and
- b is free of if- and case-expressions for each subterm if{b,...} or case{b;...} of bodyproc.

Like for constructor and selector function symbols in a data structure definition, the procedure function symbol may be preceded by a fixity declaration in the head of the procedure definition, cf. Section 2.1.3. Also formal parameters sharing a common type may be written as x₁, ⋯ , xₖ : type instead of x₁ : type, ⋯ , xₖ : type in the procedure head.

Example 3.

1. The greater-than relation on natural numbers can be defined by the procedure

   function [infixl,1] >(x : N, y : N) : bool =
   if ?0(x) then false else if ?0(y) then true else −(x) > −(y) end if end if

   Procedure > is predefined in L.5

2. Addition of natural numbers can be defined by the procedure

   function [infixr,10] +(x, y : N) : N =
   if ?0(x) then y else +(−(x) + y) end

3. Multiplication of natural numbers can be defined by the procedure

   function [infixr,20] ∗(x, y : N) : N =
   if ?0(x) then 0 else ∗(−(x) ∗ y + y) end

4. The factorial function can be defined by the procedure

   function [postfix] !(x : N) : N =
   if ?0(x) then 1 else (∗(−(x)))! ∗ x end

5 Note that ":" always has to be enclosed in blanks in a type declaration of a formal parameter.
5. List concatenation can be defined by the procedure

\[
\text{function } [\text{infixr,10}] \text{<>}(k, l : \text{list[@ITEM]}) : \text{list[@ITEM]} \text{<>} = \\
\text{if } ?\circ(k) \text{ then } l \text{ else } \text{hd}(k) :: (\text{tl}(k) \text{<>} l) \text{ end}.
\]

6. The length of a list can be computed by the procedure

\[
\text{function } [\text{outfix}] |(k : \text{list[@ITEM]}) : \mathbb{N} \text{<>} = \\
\text{if } ?\circ(k) \text{ then } 0 \text{ else } +(|\text{tl}(k)|) \text{ end}.
\]

7. Deletion of list elements can be defined by the procedure

\[
\text{function } [\text{infixr,100}] \setminus(k : \text{list[@ITEM]}, i : \text{ITEM}) : \text{list[@ITEM]} \text{<>} = \\
\text{if } ?\circ(k) \text{ then } \emptyset \text{ else if } i = \text{hd}(k) \text{ then } \text{tl}(k) \text{ else } \text{hd}(k) :: (\text{tl}(k) \setminus i) \text{ end}.
\]

8. The product of the members of a list of natural numbers can be defined by the procedure

\[
\text{function } \Pi(k : \text{list}[\mathbb{N}]) : \mathbb{N} \text{<>} = \\
\text{if } ?\circ(k) \text{ then } 1 \text{ else } \text{hd}(k) \ast \Pi(\text{tl}(k)) \text{ end}.
\]

2.2. Let-Expressions

Procedure bodies may contain so-called let-expressions, i.e. expressions of the form

\[\text{let } \text{var} := t \text{ in } r \text{ end let} \]

where

- \text{var} is some identifier—called a \textit{local variable}—which is different from the formal parameters of the procedure and from all type constructors and function symbols of the data structure and procedure definitions introduced so far, and

- \text{r} is a term which may also contain \text{var} and other local variables introduced by further let-expressions surrounding the let-expression in the procedure body.

The local variable \text{var} of a let-expression is assigned the type of term \text{t} and the whole let-expression is assigned the type of term \text{r}. Instead of \text{procedural notation}, let-expressions may also be written in \text{functional notation}, viz.

\text{let}\{\text{var} := t; \ r\}.

Instead of \text{end let}, \text{end} as well as \text{pairs of parentheses} instead of pairs of curly brackets may be used when writing let-expressions in functional notation.
Example 4.

1. The depth of binary trees represented by terms of type \texttt{SEXPR[@ITEM]} can be computed by the procedure

\begin{verbatim}
function depth(x : SEXPR[@ITEM]) : N <=
case x of
  CONS : let CAR-depth := depth(CAR(x)) in
    let CDR-depth := depth(CDR(x)) in
    if CAR-depth > CDR-depth then \(+\)(CAR-depth) else \(+\)(CDR-depth) end
  end_let
  other : 0
end_case.
\end{verbatim}

2. The minimal element of a list of natural numbers can be defined by the procedure

\begin{verbatim}
function minimum(k : list[\*N]) : N <=
  if \(?\)\textcircled{\texttt{o}}(k)
    then 0
  else if \(?\)\textcircled{\texttt{o}}(tl(k))
    then hd(k)
  else let min-tl := minimum(tl(k)) in
    if hd(k) > min-tl then min-tl else hd(k) end_if
  end_let
end_if
end_if.\end{verbatim}

\subsection*{2.2.3. Incompletely Defined Procedures}

For some procedures, a meaningful result cannot be defined for certain inputs or it is impossible to stipulate a result for certain inputs at all. For example, it is not meaningful to define the minimum of an empty list as done by procedure \texttt{minimum} of Example 4. And worse, one cannot give a procedure for computing the last element of a polymorphic list as no result can be defined for the empty list.

As a remedy, the wildcard symbol \texttt{*} may be used to denote an unspecified result in a procedure body by writing \texttt{*} in the alternatives of a conditional, yielding a \textit{incompletely defined} procedure. Incompletely defined procedures (also known as \textit{loose specifications} or \textit{underspecifications} in the literature) are in particular useful to avoid artificial results using “default” values in case of invalid inputs, see [WS05b] for further details. For instance, \texttt{0} may be replaced in procedure \texttt{minimum} of Example 4 by \texttt{*}. Further examples:
Example 5.

1. Subtraction of natural numbers can be defined by the procedure

\[
\text{function } [\text{infixl,10}] - (x, y : \mathbb{N}) : \mathbb{N} <= \\
\quad \text{if } ?0(y) \text{ then } x \text{ else if } ?0(x) \text{ then } * \text{ else } -(x) - (y) \text{ end end}
\]

2. Truncated division of natural numbers can be defined by the procedure

\[
\text{function } [\text{infixl,20}] / (x, y : \mathbb{N}) : \mathbb{N} <= \\
\quad \text{if } ?0(y) \text{ then } * \text{ else if } y > x \text{ then } 0 \text{ else } +((x - y) / y) \text{ end end}
\]

3. The remainder function for natural numbers can be defined by the procedure

\[
\text{function } [\text{infixr,20}] \text{ mod}(x, y : \mathbb{N}) : \mathbb{N} <= \\
\quad \text{if } ?0(y) \text{ then } * \text{ else if } y > x \text{ then } x \text{ else } (x - y) \text{ mod } y \text{ end end}
\]

4. The last element of a non-empty polymorphic list can be defined by the procedure

\[
\text{function } \text{last}(k : \text{list } [\text{ITEM}]) : \text{ITEM} <= \\
\quad \text{if } ?\emptyset(k) \text{ then } * \text{ else if } ?\emptyset(tl(k)) \text{ then hd(k) else last(tl(k)) end end}
\]

2.3. Lemmas

2.3.1. Definitions

A lemma is defined by an \(\mathcal{L}\)-expression of form

\[
\text{lemma } \text{lem} <= \quad \forall x_1 : \text{type}_1, \ldots, x_k : \text{type}_k \quad \text{body}_\text{lem}
\]  

(2.3)

where \(k \geq 0\) and for each \(i, j \in \{1, \ldots, k\}\):

- \(\text{lem}\)—called the name of the lemma—is an identifier different from all identifiers that have been previously introduced by data structure, procedure and lemma definitions,

- \(x_i\) is an identifier—called a variable of the lemma—different from all identifiers denoting type constructors and function symbols of the data structure and procedure definitions introduced so far,

- \(x_i \neq x_j\), if \(i \neq j\),

- \(\text{type}_i\) is an identifier denoting a type variable or a type built with type constructors different from \text{bool} defined previously by some data structures, and
• **body**—called the **body** of the lemma—is a term of type `bool` using only
  
  the lemma variables `x_i` of type `type_i`, and
  
  the function symbols of the data structure and procedure definitions introduced so far.

Like in procedure definitions, the variables of a lemma sharing a common type may be written as `x_1, \ldots, x_k : type` instead of `x_1 : type, \ldots, x_k : type` in a lemma definition. Also let-expressions may be used in the lemma body, cf. Section 2.2, where, however, the local variables must be different from the variables of the lemma. The universal quantifier “∀” may be written alternatively as “all” and is omitted if `k = 0`.

### 2.3.2. Connectives

The following table illustrates how the usual connectives are represented by the constants `true` and `false` and the boolean conditional `if`, where `a` and `b` stand for boolean terms in the table:

<table>
<thead>
<tr>
<th>Connective</th>
<th>Boolean Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬a</td>
<td>{a, false, true}</td>
</tr>
<tr>
<td>a ∨ b</td>
<td>{a, true, b}</td>
</tr>
<tr>
<td>a ∧ b</td>
<td>{a, b, false}</td>
</tr>
<tr>
<td>a → b</td>
<td>{a, b, true}</td>
</tr>
<tr>
<td>a ↔ b</td>
<td>{a, b, {b, false, true}}</td>
</tr>
<tr>
<td>¬a ∧ b</td>
<td>{a, false, b}</td>
</tr>
</tbody>
</table>

Instead of writing `if \{a, false, true\}` to denote the negated boolean term `a`, the negation operators `¬` or `not` may be used, i.e. `¬ a` or `not a` may be written alternatively.

**Example 6.**

1. The following lemma states facts about the minimal element a list of natural numbers:

   ```
   lemma minsort_sorts-lemma#1 <= ∀ k : list[N], n : N
   if \{?ø(k),
   true,
   if \{n > hd(k),
   let \{min-k := minimum(k);
   if \{min-k = n,
   true,
   if \{min-k = hd(k),
   ¬ min-k > minimum(n :: tl(k)),
   ¬ min-k > minimum(hd(k) :: (tl(k) \ min-k)))\}
   let \{min-k := minimum(n :: tl(k));
   if \{min-k = n, ¬ min-k > minimum(k), true\}\}
   .
   ```

2. Associativity of `+` is stated by the lemma

   ```
   lemma + associative <= ∀ x, y, z : N (x + y) + z = x + y + z .
   ```
3. Distributivity of $+$ and $*$ is stated by the lemma

\[ \text{lemma } * \text{ distributes over } + \implies \forall x, y, z : \mathbb{N} \]

\[ \text{if } \{ (x + y) * z = x * z + y * z, x * (y + z) = x * y + x * z, \text{false} \} \].
3. Semantics of $\mathcal{L}$

Here we present a brief account on the semantics of $\mathcal{L}$ and refer to [Sch06] for details.

3.1. Semantics of Data Structures and Procedures

We define an operational semantics for $\mathcal{L}$, thus providing a meaning for the expressions of the language: Given an $\mathcal{L}$-program $P$ as a list of data structure and procedure definitions, we write $\mathcal{G}(P)_\tau$ for the set of ground terms of a monomorphic type $\tau$ in $P$ built with the function symbols introduced by the data structure and procedure definitions of $P$, and we let $\mathcal{C}(P)_\tau$ denote the set of constructor ground terms of a monomorphic type $\tau$ in $P$, i.e. those terms of $\mathcal{G}(P)_\tau$ which contain constructor function symbols only.

The elements of $\mathcal{C}(P)_\vdash S \tau \mathcal{C}(P)_\tau$ denote the set of all values given for the program $P$ and the elements of $\mathcal{G}(P)_\vdash S \tau \mathcal{G}(P)_\tau$ denote the set of all expressions for which values shall be computed by the procedures of $P$. Such computations are formally defined by a partial mapping $\text{eval}_P : \mathcal{G}(P) \rightarrow \mathcal{G}(P)$ which assigns the result $\text{eval}_P(t) \in \mathcal{G}(P)$ computed for a term $t \in \mathcal{G}(P)$ to this term. The mapping $\text{eval}_P$ is called the interpreter of the $\mathcal{L}$-program $P$.

Since $P$ may contain non-terminating procedures, cf. Section 3.1.4, $\text{eval}_P$ is a partial mapping only, which means that $\text{eval}_P(t)$ may not exist for some program $P$ and some term $t \in \mathcal{G}(P)$. We say in such a case, that the computation of $t$ diverges and write $\text{eval}_P(t) = \perp$ for denoting divergence. Otherwise, either $\text{eval}_P(t) \in \mathcal{C}(P)$, called a succeeding computation of $t$, because a value is obtained for $t$, or $\text{eval}_P(t) \in \mathcal{G}(P) \setminus \mathcal{C}(P)$ else, which means that the computation of $t$ has failed. The result $\text{eval}_P(t)$ is called a stuck computation of $t$ in such a case.

3.1.1. The Exception Guard

To care for the computation of calls of incompletely defined procedures, cf. Section 2.2.3, a so-called exception guard $\text{except}_{\text{proc}}$ is associated with each procedure $\text{proc}$ as defined in (2.2) of Section 2.2: For each occurrence $\pi$ of the wildcard symbol $*$ in the body of procedure $\text{proc}$, all conditions leading to this occurrence are collected in a set $C_\pi$. The exception guard $\text{except}_{\text{proc}}$ then is defined as a boolean term representing the formula $\vee_{\pi \in \Pi^*} \land_{c \in C_\pi} c$, where $\Pi^*$ is the set of all occurrences of $*$ in the body of procedure $\text{proc}$.

One may think of the exception guard as a requirement such that $\text{except}_{\text{proc}}[x_1/q_1, \ldots, x_n/q_n]$ is satisfied for $q_1, \ldots, q_n \in \mathcal{C}(P)$ iff the computation of the procedure call $\text{proc}(q_1, \ldots, q_n)$ raises an exception, see [WS05b] for further details.\footnote{1 We write $t[x_1/t_1, \ldots, x_n/t_n]$ for the term obtained from term $t$ by replacing each variable symbol $x_i$ of $t$ by the term $t_i$.}
Example 7.

1. For all procedures of Example 3, the exception guards are given as false.

2. For procedure “−” from Example 5, except_− is given as if {?0(y), false, ?0(x)}.

3. For the procedures “/” and “mod” from Example 5, except_/ and except_mod both are given as ?0(y).

4. For the procedure

\[
\text{function } \log_2(x : \mathbb{N}) : \mathbb{N} \leq \begin{align*}
\text{if } ?0(x) \\
\text{then } * \\
\text{else if } ?0(^{-}(x)) \\
\text{then } 0 \\
\text{else if } ?0(x \text{ mod } 2) \text{ then } ^{+}(\log_2(x / 2)) \text{ else } * \text{ end} \\
\text{end} \\
\end{align*}
\]

(computing the binary logarithm of powers of 2), the exception guard except_log_2 is given as if {?0(x), true, if {?0(^{-}(x)), false, ¬ ?0(x mod 2)}}.

3.1.2. The Computation Calculus

The interpreter \(\text{eval}_P\) is defined by the so-called computation calculus which stipulates how \(\text{eval}_P(t)\) is computed for an expression \(t \in G(P)\). This calculus consists of so-called computation rules of form

\[
t \xrightarrow{r} p, \text{ if } \text{cond}(t, r)
\]

where \(t, r \in G(P)\) and \(\text{cond}(t, r)\) is a side condition controlling the application of the computation rule.

We write \(t_1 \Rightarrow_P t_2\) and call the replacement of \(t_1\) by \(t_2\) a computation step, if some computation rule applies for \(t_1\) and \(t_2\), i.e. \(\text{cond}(t_1, t_2)\) is satisfied for the pair of terms \(t_1, t_2\). As usual, \(\Rightarrow^*_P\) denotes the transitive closure of \(\Rightarrow_P\) and \(\Rightarrow_P\) is the reflexive closure of \(\Rightarrow^*_P\). We write \(t \Rightarrow^*_P t'\) iff \(t \Rightarrow^*_P t'\) for some \(t' \in G(P)\) and \(t' \Rightarrow_P t''\) for each \(t'' \in G(P)\). \(\Rightarrow_P\) is defined as \(t'\) iff exactly one \(t'\) such that \(t \Rightarrow^*_P t'\) exists.

Subsequently, we treat ¬ \(b\) as an abbreviation for if(\(b, \text{false}, \text{true}\)), and we assume that data structure and procedure definitions are given in the same format as in (2.1) and (2.2) respectively.

Rules for Constructors, Selectors and Equality

\[
\begin{align*}
\text{cons}_j(\text{cons}_i(q_1, \ldots, q_n)) & \text{, if } q_1, \ldots, q_n \in C(P) \quad (3.1) \\
\text{cons}_j(\text{cons}_i(q_1, \ldots, q_n)) & \text{, if } q_1, \ldots, q_n \in C(P) \text{ and } j \neq i \quad (3.2)
\end{align*}
\]
\[
\text{sel}_{i,h}(\text{cons}(q_1,\ldots,q_n)) \quad \text{if } q_1,\ldots,q_n \in \mathcal{C}(P) \quad (3.3)
\]

\[
q_1 = q_2 \quad \text{true}, \quad \text{if } q_1, q_2 \in \mathcal{C}(P) \text{ and } q_1 = q_2 \quad (3.4)
\]

\[
q_1 = q_2 \quad \text{false}, \quad \text{if } q_1, q_2 \in \mathcal{C}(P) \text{ and } q_1 \neq q_2 \quad (3.5)
\]

**Rules for Conditionals**

\[
\frac{\text{if } \{b,t_1,t_2\}}{\text{if } \{b',t_1,t_2\}}, \quad \text{if } b \Rightarrow b' \quad (3.6)
\]

\[
\frac{\text{if } \{\text{true},t_1,t_2\}}{t_1} \quad (3.7)
\]

\[
\frac{\text{if } \{\text{false},t_1,t_2\}}{t_2} \quad (3.8)
\]

\[
\text{case}\{c; \text{cons}_1 : t_1,\ldots,\text{cons}_k : t_k\} = \text{case}\{c' ; \text{cons}_1 : t_1,\ldots,\text{cons}_k : t_k\}, \quad \text{if } c \Rightarrow c' \quad (3.9)
\]

\[
\text{case}\{\text{cons}_1(q_1,\ldots,q_n); \text{cons}_1 : t_1,\ldots,\text{cons}_k : t_k\} = \frac{\text{cons}_1 : t_1,\ldots,\text{cons}_k : t_k}{t_i}, \quad \text{if } q_1,\ldots,q_n \in \mathcal{C}(P) \quad (3.10)
\]

\[
\text{case}\{\text{cons}_1(q_1,\ldots,q_n); \text{cons}_1 : t_{\pi(1)},\ldots,\text{cons}_h : t_{\pi(h)}, \text{other} : t\} = \text{case}\{\text{cons}_1 : t_{\pi(1)},\ldots,\text{cons}_h : t_{\pi(h)}, \text{other} : t\}, \quad \text{if } c \Rightarrow c', \text{ where } \pi : \{1,\ldots,k\} \rightarrow \{1,\ldots,k\} \text{ is a bijective mapping} \quad (3.11)
\]

\[
\text{case}\{\text{cons}_1(q_1,\ldots,q_n); \text{cons}_{\pi(1)} : t_{\pi(1)},\ldots,\text{cons}_{\pi(h)} : t_{\pi(h)}, \text{other} : t\} = \frac{\text{cons}_{\pi(1)} : t_{\pi(1)},\ldots,\text{cons}_{\pi(h)} : t_{\pi(h)}, \text{other} : t}{t_i}, \quad \text{if } q_1,\ldots,q_n \in \mathcal{C}(P) \text{ and } i = \pi(j) \text{ for some } j \leq h, \quad \text{where } \pi : \{1,\ldots,k\} \rightarrow \{1,\ldots,k\} \text{ is a bijective mapping} \quad (3.12)
\]
\[
\text{case}\{\text{cons}_i(q_1, \ldots, q_n) ; \ \text{cons}_{\pi(1)}(t_{\pi(1)}, \ldots, t_{\pi(h)}) ; \ \text{other} : t\} \\
\]

if \(q_1, \ldots, q_n \in \mathcal{C}(P)\) and \(i \neq \pi(j)\) for each \(j \leq h\),
where \(\pi : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\}\) is a bijective mapping

**Rules for let-Expressions**

\[
\begin{align*}
\text{let}\{\text{var} := t \} & \Rightarrow_\text{P} \text{let}\{\text{var} := t' \} \\
\text{let}\{\text{var} := t ; r \} & \Rightarrow_\text{P} \text{let}\{\text{var} := t ; r \} \\
\text{let}\{\text{var} := t ; r \} & \Rightarrow_\text{P} r \text{[var/}t]\ 
\end{align*}
\]

**Rules for Function Applications**

\[
\begin{align*}
\frac{f(t_1, \ldots, t_i, \ldots, t_n)}{f(t_1, \ldots, t'_i, \ldots, t_n)}, & \text{ if } f \notin \{\text{if}, \text{case}\} \text{ and } t_i \Rightarrow_\text{P} t'_i \\
\text{proc}(q_1, \ldots, q_k) & \Rightarrow_\text{P} \text{body}_{\text{proc}}[x_1/q_1, \ldots, x_k/q_k], \text{ if } q_1, \ldots, q_k \in \mathcal{C}(P) \text{ and } \text{except}_{\text{proc}}[x_1/q_1, \ldots, x_k/q_k] = \text{false} 
\end{align*}
\]

3.1.3. The Interpreter \(\text{eval}_P\)

The computation relation \(\Rightarrow_\text{P}\) neither is *noetherian* (as \(P\) may contain non-terminating procedures, cf. Section 3.1.4) nor *deterministic* (by presence of computation rule (3.16)). However, \(\Rightarrow_\text{P}\) is *confluent*, i.e. for all \(t, t_1, t_2 \in \mathcal{G}(P)\) such that \(t_1 \Rightarrow_\text{P} r, t_2 \Rightarrow_\text{P} s\) some \(r \in \mathcal{G}(P)\) satisfying \(t_1 \Rightarrow_\text{P} r, r \Rightarrow_\text{P} s = t_2\) exists.

Each constructor ground term is \(\Rightarrow_\text{P}-\text{minimal}\), i.e. \(q \Rightarrow_\text{P} q\) for each \(q \in \mathcal{C}(P)\), as no computation rule can be applied to a term \(q \in \mathcal{C}(P)\). Each term \(t \in \mathcal{G}(P) \setminus \mathcal{C}(P)\) satisfying \(t \Rightarrow_\text{P} t\) is called a *stuck computation*. Those terms stem from

- the application of a selector to a constructor it does not belong to, as e.g. \(-0\), \(\text{CAR(NIL)}\), \(\text{DATA(CONS(\ldots))}\), \(\text{CDR(ATOM(\ldots))}\) for the data structures of Example 1,
- the call of a procedure with arguments for which the exception guard is computed to a term \(\neq \text{false}\), e.g. \(1/0\), \(\text{mod}(1/-0)\), \(\text{log}_2(3)\) for the procedures of Examples 5 and 7,
- the computation of a conditional having a stuck computation as first argument, e.g. \(\text{if}((\text{mod}(\log_2(3)), \ldots, \ldots), \text{case}((\text{CAR(NIL)}), \ldots))\), and
- the application of a function symbol different from \(\text{if}\) and \(\text{case}\) to one stuck argument at least, e.g. \(\text{CDR(CAR(NIL))}\), \(\text{CONS(NIL, CDR(NIL))}\), \(\text{mod}(1/0, \log_2(3))/1 = 5\), \(+(-0))\).
By the confluence of $\Rightarrow_P$, for each $t \in G(P)$ at most one $r \in G(P)$ satisfying $t \Rightarrow_P r$ exists, hence $r = t \Rightarrow_P$ in such a case. Consequently, $\text{eval}_P$ is well-defined by stipulating

$$\text{eval}_P(t) := \begin{cases} t \Rightarrow_P, & \text{if } t \Rightarrow_P r \text{ for some } r \in G(P) \\ \bot, & \text{otherwise.} \end{cases}$$

### 3.1.4. Termination of \(L\)-Programs

#### Completely Defined Programs without Polymorphic Types

By presence of the wildcard symbol $\ast$ and the stuck computations caused by selectors applied to constructors they do not belong to, \(L\)-programs are inherently incompletely defined. We therefore call an \(L\)-program completely defined [WS05b] iff

- the wildcard symbol $\ast$ is not used in any procedure body, and
- witness terms $\omega_{\text{sel}_{i,h}}$ containing only a variable $x$ of type `struct`—given as in (2.1)—at most are provided for each selector $\text{sel}_{i,h}$ for defining the result of a selector applied to a constructor it does not belong to.\(^2\)

A procedure

\[
\text{function } \text{proc}(x_1: \text{type}_1, \ldots, x_k: \text{type}_k) : \text{type} \quad <= \quad \text{body}_\text{proc}
\]

of a completely defined \(L\)-program \(P\) without polymorphic types terminates iff

$$\text{eval}_P(\text{proc}(q_1, \ldots, q_k)) \in C(P)_{\text{type}}$$

- for each $q_1 \in C(P)_{\text{type}_1}, \ldots, q_k \in C(P)_{\text{type}_k}$.\(^3\)

A completely defined \(L\)-program \(P\) without polymorphic types terminates iff

1. each procedure \(\text{proc}\) of \(P\) terminates, and
2. $\text{eval}_P(\omega_{\text{sel}_{i,h}} [x/q]) \in C(P)_{\text{type}_{i,h}}$ for each $q \in C(P)_{\text{type}}$.

#### Incompletely Defined Programs without Polymorphic Types

For defining termination of incompletely defined \(L\)-programs \(P\) without polymorphic types, the notion of a fair completion of program \(P\) is used: A completely defined program \(P'\) is a fair completion of program \(P\) iff the body of each procedure

\[
\text{function } \text{proc}(x_1: \text{type}_1, \ldots, x_k: \text{type}_k) : \text{type} \quad <= \quad \text{body}_\text{proc}
\]

in \(P\) coincides with the body of some procedure

\[
\text{function } \text{proc}(x_1: \text{type}_1, \ldots, x_k: \text{type}_k) : \text{type} \quad <= \quad \text{body}_\text{proc}'
\]

\(^2\) There is no feature in \(L\) allowing to assign witness terms. We abstain from extending \(L\) by such a feature as their is no practical use for those programs.

\(^3\) $\text{eval}_P(t) \in C(P)$ is equivalent to $\text{eval}_P(t) \neq \bot$ as $\text{eval}_P(t) \in G(P) \setminus C(P)$ is impossible by the requirements for completely defined programs with monomorphic types.
in $P'$ except for the unspecified $*$-cases in $\text{body}_{\text{proc}}$. Almost any result may be stipulated for those cases in $\text{body}'_{\text{proc}}$, however the fairness requirement demand that (i) termination of procedure $\text{proc}$ in $P'$ not be spoiled just because procedure $\text{proc}$ was completed by a non-terminating result in a $*$-case or (ii) a non-terminating witness term was assigned to a selector $\text{sel}_{i,h}$ used in $\text{body}'_{\text{proc}}$.

A procedure

$$\text{function proc}(x_1: \text{type}_1, \ldots, x_k: \text{type}_k): \text{type} \quad \text{<= body}_{\text{proc}}$$

of an incompletely defined $\mathcal{L}$-program $P$ without polymorphic types terminates iff

$$\text{eval}_{P'}(\text{proc}(q_1, \ldots, q_k)) \in \mathcal{C}(P)_{\text{type}}$$

- for each fair completion $P'$ of $P$, and
- for each $q_1 \in \mathcal{C}(P)_{\text{type}_1}, \ldots, q_k \in \mathcal{C}(P)_{\text{type}_k}$.

An incompletely defined $\mathcal{L}$-program $P$ without polymorphic types terminates iff each procedure $\text{proc}$ of $P$ terminates, see [WS05b] for further details.

Incompletely Defined Programs with Polymorphic Types

For defining termination of general $\mathcal{L}$-programs $P$, we consider additional data structure definitions in a fair completion $P'$ of program $P$ too.

A procedure

$$\text{function proc}(x_1: \text{type}_1, \ldots, x_k: \text{type}_k): \text{type} \quad \text{<= body}_{\text{proc}}$$

of an $\mathcal{L}$-program $P$ terminates iff

$$\text{eval}_{P'}(\text{proc}(q_1, \ldots, q_k)) \in \mathcal{C}(P)_{\theta(\text{type})}$$

- for each fair completion $P'$ of $P$,
- for each type substitution $\theta$ which replaces the type variables in $\text{type}_1, \ldots, \text{type}_k, \text{type}$ by monomorphic types of $P'$ yielding the monomorphic types $\theta(\text{type}_1), \ldots, \theta(\text{type}_k), \theta(\text{type})$, and
- for each $q_1 \in \mathcal{C}(P)_{\theta(\text{type}_1)}, \ldots, q_k \in \mathcal{C}(P)_{\theta(\text{type}_k)}$, for each $q_i \in \mathcal{C}(P)_{\text{type}_i}$.

An $\mathcal{L}$-program $P$ terminates iff each procedure $\text{proc}$ of $P$ terminates.

It should be noted that the existence of a diverging computation of a procedure call $\text{proc}(q_1, \ldots, q_k)$ with arguments $q_i \in \mathcal{C}(P)$ entails non-termination of procedure $\text{proc}$. However, the converse does not hold as there are non-terminating procedures causing no divergent computations.
3.2. Semantics of Lemmas

A lemma

\[ \text{lemma } \text{lem} \quad \Leftarrow \quad \forall x_1 : \text{type}_1, \ldots, x_k : \text{type}_k \quad \text{body}_{\text{lem}} \]

of a terminating \( \mathcal{L} \)-program \( P \) is defined to be true iff

\[ \text{eval}^{P'}(\text{body}_{\text{lem}}[x_1/q_1, \ldots, x_k/q_k]) = \text{true} \]

- for each fair completion \( P' \) of \( P \),
- for each type substitution \( \theta \) which replaces the type variables in \( \text{type}_1, \ldots, \text{type}_k \) by monomorphic types \( \theta(\text{type}_1), \ldots, \theta(\text{type}_k) \), and
- for each \( q_1 \in \mathcal{C}(P)_{\theta(\text{type}_1)}, \ldots, q_k \in \mathcal{C}(P)_{\theta(\text{type}_k)} \).

3.3. Domain Procedures

Partial definitions are a means to model undefinedness, cf. the procedures of Example 5 and procedure \( \text{log}_2 \) of Example 7. However, formally a procedure call like \( \text{log}_2(0) \) denotes an unknown value causing a failed computation only, whereas a procedure call like \( \text{loop}(0) \), cf. Example 8, causes a diverging computation and does not denote a value at all.
By the semantics of $\mathcal{L}$-lemmas, cf. Section 3.2, a lemma like

$$\text{lemma lem1} \iff \forall x : \mathbb{N} x/0 = x \mod 0$$

is false, because a fair completion $P'$ containing the fairly completed procedures “/” and “mod” exists such that e.g. $\text{eval}_{P'}(0/0) = 0$ but $\text{eval}_{P'}(0 \mod 0) = 1$. However, a lemma like

$$\text{lemma lem2} \iff \forall x : \mathbb{N} x/0 = x/0$$

is true, because $\text{eval}_{P'}(q/0 = q/0) = true$ for each $q \in C(P)\mathbb{N}$ and for any fair completion $P'$ containing the fairly completed procedure “/”, although $\text{eval}_{P}$ computes a stuck computation as $\text{eval}_{P}(q/0 = q/0) = q/0 = q/0$.

In order to recognize such unwanted truths caused by the differences between the

- semantics of data structures and procedures of an $\mathcal{L}$-program $P$ given by the interpreter $\text{eval}_P$ and the
- semantics of lemmas given by the interpreters $\text{eval}_{P'}$ of all fairly completed programs $P'$,

so-called domain procedures [WS05b] may be used: For any function symbol $f : \text{type}_1 \times \ldots \times \text{type}_k \to \text{type}$ of an $\mathcal{L}$-program with $f \notin \{\text{if, case}\}$, it is demanded that this $\mathcal{L}$-program contains a domain procedure

$$\text{function } \nabla f(x_1 : \text{type}_1, \ldots, x_k : \text{type}_k) : \text{bool} \iff \text{body}_{\nabla f}$$

satisfying for each type substitution $\theta$ which replaces the type variables in $\text{type}_1, \ldots, \text{type}_k$ by monomorphic types of $P$—yielding the monomorphic types $\theta(\text{type}_1), \ldots, \theta(\text{type}_k)$—and for each $q_1 \in C(P)_{\theta(\text{type}_1)}, \ldots, q_k \in C(P)_{\theta(\text{type}_k)}$

1. $\text{eval}_P(\nabla f(q_1, \ldots, q_k)) \in G(P) \iff \text{eval}_P(f(q_1, \ldots, q_k)) \in G(P)$,
2. $\text{eval}_P(\nabla f(q_1, \ldots, q_k)) \in G(P) \iff \text{eval}_P(\text{eval}_P(\nabla f(q_1, \ldots, q_k)) \in G(P)$, and
3. $\text{eval}_P(\nabla f(q_1, \ldots, q_k)) = true \iff \text{eval}_P(f(q_1, \ldots, q_k)) \in C(P).$

By requirement (1), the computation of $\nabla f(q_1, \ldots, q_k)$ diverges iff the computation of $f(q_1, \ldots, q_k)$ diverges. By requirement (2), stuck computations are never obtained by calling domain procedures. Finally by requirement (3), a domain procedure $\nabla f$ decides for each list $q_1, \ldots, q_k$ of actual parameters whether a non-diverging computation of $f(q_1, \ldots, q_k)$ fails or succeeds. Section 4 provides examples for the use of domain procedures.

Example 9.

1. The domain procedure for procedure “+” from Example 3 is given as

$$\text{function } \nabla +(x, y : \mathbb{N}) : \text{bool} <= true.$$

---

In $\texttt{VeriFun}$, domain procedures (with fixity $\texttt{prefix}$) are automatically synthesized for each function symbol different from a conditional.
2. The domain procedure for procedure \textit{mod} from Example 5 is given as

\[ \text{function } \nabla \text{mod}(x, y : \mathbb{N}) : \text{bool} <= \neg \exists 0(y). \]

3. The domain procedure for procedure \textit{log} from Example 7 is given as

\[ \text{function } \nabla \text{log}_2(x : \mathbb{N}) : \text{bool} <= \]

\[ \text{if } \exists 0(x) \]
\[ \text{then false} \]
\[ \text{else if } \exists 0(-x) \]
\[ \text{then true} \]
\[ \text{else if } \exists 0(x \mod 2) \]
\[ \nabla \text{log}_2(x / 2) \]
\[ \text{else false end} \]
\[ \text{end}. \]

\[ ^5 \text{Note that } \text{eval}_P(\text{except}_{\text{proc}}[x_1/q_1, \ldots, x_k/q_k]) = \text{true} \text{ entails } \text{eval}_P(\nabla_{\text{proc}}(q_1, \ldots, q_k)) = \text{false} \text{ but } \text{eval}_P(\nabla_{\text{proc}}(q_1, \ldots, q_k)) = \text{false} \text{ does not entail } \text{eval}_P(\text{except}_{\text{proc}}[x_1/q_1, \ldots, x_k/q_k]) = \text{true}. \text{ For example, } \text{eval}_P(\nabla_{\text{log}_2}(3)) = \text{false} \text{, but } \text{eval}_P(\text{except}_{\text{log}_2}[x/3]) = \text{false} \text{ as well.} \]
4. The Tautology Checker - A Case Study

We illustrate the use of \( \mathcal{L} \) by a small case study taken from [BM79]. In this example, syntax and semantics of a propositional language are defined, a tautology checker for formulas of this language is given and lemmas stating soundness and completeness of this procedure are provided. We also give some of the auxiliary lemmas which are required to prove both main statements in \texttt{veriFun}.

4.1. Preliminaries

The propositional language is defined by the data structure

\[
\text{structure IF.Expr } <= \\
T, \\
F, \\
PROP(index : N), \\
IF(test : IF.Expr, left : IF.Expr, right : IF.Expr)
\]

where the constructor constants \( T \) and \( F \) are intended to denote truth and falsity, propositional variables are given as \( PROP(0) \), \( PROP(1) \), \( PROP(2) \), ... and \( IF \) is the only connective of the language intended to denote \((a \rightarrow b) \land (\neg a \rightarrow c)\) by a term \( IF(a, b, c) \).

As usual, the meaning of a propositional formula is given by a valuation, i.e. a mapping which assigns truth values to propositional variables. As each propositional formula consists of finitely many propositional variables only, a mapping assigning truth values to finitely many propositional variables suffices. The graph of such a partial valuation can be represented by a finite list of pairs \((i, tv)\), where \( i \) denotes the index of the propositional variable \( PROP(i) \) to which the truth value \( tv \) is assigned.

Truth values, pairs and lists are defined by the data structures

\[
\text{structure TruthValue } <= \text{TRUE, FALSE} \\
\text{structure pair[@ITEM1,@ITEM2] } <= \\
[\text{infixr,100}] \bullet ([\text{postfix} \ 1 : @ITEM1, [\text{postfix} \ 2 : @ITEM2}) \\
\text{structure list[@ITEM] } <= \varnothing, [\text{infixr,100}] ::(\text{hd : @ITEM, tl : list[@ITEM])}
\]

so that partial valuations can be represented by terms of type \( \text{list [pair [N, TruthValue]]} \). This type is an instance of the type \( \text{list [pair [@ITEM1, @ITEM2]]} \) the terms of which represent so-called association lists. For those lists, the data associated with a certain key can be computed by the procedure
function \texttt{infixl, 20} \circ (\texttt{alist : list[pair[@ITEM1, @ITEM2]], key : @ITEM1}) : @ITEM2 <=
if ?o(\texttt{alist})
then *
else let head := hd(\texttt{alist}) in
if key = (head)
then (head)
else tl(\texttt{alist}) \circ key end if
end if.

Since no data can be retrieved by procedure \texttt{circ} if a key cannot be found in the association list, the result of \texttt{circ} is indetermined in such a case. Consequently, the domain procedure of \texttt{circ} is given as

\begin{verbatim}
function \texttt{infixr, 10} \triangledown \circ (\texttt{alist : list[pair[@ITEM1, @ITEM2]], key : @ITEM1}) : bool <=
if ?o(\texttt{alist})
then false
else if key = (hd(\texttt{alist})); then true else \texttt{infixr, 10} \triangledown \circ (tl(\texttt{alist}), key) end if
end if.
\end{verbatim}

Using these definitions, the semantics of propositional formulas is defined by a procedure

\begin{verbatim}
function \texttt{infixr, 10} \vdash (\sigma : list[pair[N, TruthValue]], x : IF.Expr) : bool <=
case x of
T : true,
F : false,
PROP : \texttt{TRUE}(\sigma \circ index(x)) ,
IF : if \sigma \vdash test(x) then \sigma \vdash left(x) else \sigma \vdash right(x) end if
end case.
\end{verbatim}

Procedure \texttt{\vdash} implements the satisfaction relation yielding \texttt{true} if a partial valuation \(\sigma\) satisfies a propositional formula \(x\), and yielding \texttt{false} if \(\sigma\) falsifies \(x\). However, since it is not guaranteed that each propositional variable of formula \(x\) considered by procedure \texttt{\vdash} is mentioned in the partial valuation \(\sigma\), computation of \(\sigma \vdash x\) might fail with a stuck computation.

### 4.2. The Tautology Checking Procedure

The base idea of the tautology checking algorithm is to decide for a propositional formula \(x\) whether a given partial valuation \(\sigma\) can be extended to a valuation which falsifies \(x\). If so, the extended valuation is returned, and otherwise the algorithm reports failure about computing a falsifying extension of \(\sigma\). Now to check whether \(x\) is a tautology, the algorithm is called with the empty valuation \(\sigma\). If failure is reported, no falsifying valuation for \(x\) exists—hence each valuation must satisfy \(x\)—and so \(x\) is a tautology. Otherwise there is some valuation falsifying \(x\), and then \(x\) cannot be a tautology.

The result type of the tautology checker is given by the type of valuations expanded by some failure indication. We therefore introduce the data structure

\begin{verbatim}
structure yields[@ITEM] <= \bot, return(result : @ITEM))
\end{verbatim}

\footnote{By abuse of notation, “\texttt{\circ}” stands for “content”.
}
to be used for defining the result type of the tautology checker. This algorithm now can implemented by the procedure

```
function \( \triangledown (\sigma : \text{list}[\text{pair}[:\mathbb{N}, \text{TruthValue}]], x : \text{IF.Expr}) : \text{yields}[:\text{list}[\text{pair}[:\mathbb{N}, \text{TruthValue}]]] \)
```

case \( x \) of

\( T : \bot, \)

\( F : \text{return}(\sigma), \)

\( \text{PROP} : \text{let} \ prop := \text{index}(x) \) in

\( \text{if} \ \nabla c°(\sigma, prop) \)

\( \text{then} \ if \ ?FALSE(\sigma \odot prop) \text{then return}(\sigma) \text{else } \bot \text{endif} \)

\( \text{else return}(\text{prop} \bullet \text{FALSE} :: \sigma) \)

\( \text{endif} \)

\( \text{endlet}, \)

\( \text{IF} : \text{case test}(x) \) of

\( T : \sigma \nabla \leftarrow(x) , \)

\( F : \sigma \nabla \rightarrow(x) , \)

\( \text{PROP} : \text{let} \ prop := \text{index}(\text{test}(x)) \) in

\( \text{if} \ \nabla c°(\sigma, prop) \)

\( \text{then} \ if \ ?TRUE(\sigma \odot prop) \text{then } \sigma \nabla \leftarrow(x) \text{else } \sigma \nabla \rightarrow(x) \text{endif} \)

\( \text{else let } \rho := (\text{prop} \bullet \text{TRUE}) :: \sigma \nabla \leftarrow(x) \text{in} \)

\( \text{if } ?\bot (\rho) \text{then } (\text{prop} \bullet \text{FALSE}) :: \sigma \nabla \rightarrow(x) \text{else } \rho \text{endif} \)

\( \text{endif} \)

\( \text{endlet}, \)

\( \text{IF} : * \)

\( \text{end_case} \).

Procedure \( \triangledown \) recursively explores a propositional formula \( x \): If \( x = T \), failure \( \bot \) is reported as \( T \) cannot be falsified, and the input valuation \( \sigma \) is returned for \( x = F \), as \( \sigma \) (vacuously) falsifies \( F \). If \( x \) is a propositional variable, domain procedure \( \nabla c° \) is used to decide whether \( x \) is assigned a truth value by the partial valuation \( \sigma \): If \( x \) is assigned \( FALSE \), the input valuation \( \sigma \) is returned as \( \sigma \) falsifies \( x \). If \( x \) is assigned \( TRUE \), failure \( \bot \) is reported as \( x \) cannot be falsified by any extension of \( \sigma \). If, however, \( x \) is not assigned a truth value, \( x \) can be falsified by assigning \( FALSE \) to it, and this extension of \( \sigma \) is returned as result.

If \( x \) is built with the \( \text{IF}-\)constructor, the result of \( \triangledown \) is computed depending on the propositional formula \( \text{test}(x) \) in the test of \( x \): If \( \text{test}(x) \) is \( T \) or \( F \), the respective alternatives have to be explored recursively by \( \triangledown \). If \( \text{test}(x) \) is a propositional variable, domain procedure \( \nabla c° \) is used again to decide whether \( \text{test}(x) \) is assigned a truth value by \( \sigma \). If so, the respective alternatives recursively have to be explored by \( \triangledown \). Otherwise the partial valuation \( \sigma \) has to be extended: First it is tried to falsify the left branch of the \( \text{IF}-\)conditional under the additional assumption that \( \text{test}(x) \) is \( TRUE \). If this computation succeeds, the computed extension of \( \sigma \) is returned. Otherwise it is tried to falsify the right branch of the \( \text{IF}-\)conditional under the additional assumption that \( \text{test}(x) \) is \( FALSE \), and the result of this computation defines the final result of \( \triangledown \).

---

2 Data structure \text{yields} is generally useful to define the result type of procedures which—like the tautology checker—\textit{decide} whether some problem is solvable and \textit{return} some solution if possible. Further examples for those kinds of procedures are a unification algorithm or a parser.
The result of $\forall$ when applied to a propositional formula of form $IF(IF(\ldots),\ldots)$ is left unspecified by procedure $\forall$. This means that computations succeed only if $\forall$ is called with propositional formulas free of $IF$'s in the tests. We say that such formulas are in $IF$-normal form, and one may verify that each propositional formula can be transformed into an equivalent formula in $IF$-normal form.

The $IF$-normal form of a propositional formula is computed by procedure

```
function [outfix] ||(x : IF.Expr) : IF.Expr <=
  if IF(x)
    then || IF(test(test(x)), IF(left(test(x)), left(x), right(x)), IF(right(test(x)), left(x), right(x))) ||
    else IF(test(x), || left(x) ||, || right(x) ||)
  end
else x
end_if
```

and

```
lemma normalization is sound <= \forall \sigma : list[pair[N, TruthValue]], x : IF.Expr
if {\sigma \models || x ||, \sigma \models x, \neg \sigma \models x}
```

states that normalization of a propositional formula $x$ by procedure $||$ computes a formula equivalent to $x$ in fact. The fact that computations of $\forall$ always succeed for formulas in $IF$-normal form is expressed by

```
lemma normalization $\forall$ determination <=
\forall \sigma : list[pair[N, TruthValue]], x : IF.Expr $\nabla \forall (\sigma, || x ||)$.3
```

### 4.3. Soundness of the Tautology Checking Procedure

Soundness of the tautology checker is stated by

```
lemma $\forall$ is sound <= \forall \sigma : list[pair[N, TruthValue]], x : IF.Expr
if {? \perp (\sigma $\forall$ || x ||), \sigma \models x, true}
```

asserting that each partial valuation $\sigma$ satisfies a propositional formula $x$ whenever the tautology checker—called with the normal form of $x$ and the empty valuation $\perp$—reports failure $\perp$. To prove this theorem, two auxiliary lemmas are needed:

```
lemma $\forall$ ignores assignments <= \forall x : IF.Expr, \sigma : list[pair[N, TruthValue]], n : N
if {((n \bullet (\sigma \odot n)) :: $\sigma \models x, \sigma \models x, \neg \sigma \models x} ,
```

states that duplicated assignments for propositional variables do not affect the satisfaction relation, and

```
lemma $\forall$ soundness-lemma <= \forall x : IF.Expr, \sigma, \theta : list[pair[N, TruthValue]]
if {\forall \perp (\theta $\forall$ || x ||), \theta \models x, true} ,
```

3 Lemmas normalization is sound and normalization $\forall$ determination are not required to verify soundness and completeness of the tautology checker.
states that each partial valuation $\theta < > \sigma$ satisfies a propositional formula $x$ whenever the tautology checker—called with the normal form of $x$ and some partial valuation $\theta$—reports failure $\bot$. This lemma is the “key lemma” of the soundness proof, and lemma “$\vdash$ is sound” follows immediately from lemma “$\vdash$ soundness-lemma” simply by replacing $\theta$ with $\phi$.

4.4. Completeness of the Tautology Checking Procedure

A tautology checker is complete if an affirmative answer is returned when checking a propositional formula $x$ which is satisfied by each valuation $\sigma$ of the propositional variables in $x$. Consequently, the contraposition of this requirement demands that a falsifying valuation for a formula $x$ exists, if the tautology checker returns a negative answer for $x$.

When using procedure $\vdash$ as the tautology checker, a negative answer is represented by a result of form $\text{return}(\sigma)$, where $\sigma$ is a falsifying valuation for $x$. Hence completeness of $\vdash$ can be stated by

$$\forall x: \text{IF.Expr} \; \text{return}(\sigma \vdash x) \rightarrow \exists \sigma: \text{list}[\text{pair}[\mathbb{N}, \text{TruthValue}]] \neg \sigma \models x. \quad (4.1)$$

However, statement (4.1) cannot be written as an $\mathcal{L}$-lemma, because it is not a universal formula by presence of the existential quantifier in the conclusion of the implication. Hence a stronger statement is formulated, viz.

$$\forall x: \text{IF.Expr} \; \text{return}(\sigma \vdash x) \rightarrow \neg \text{result}(\sigma \vdash x) \models x \quad (4.2)$$

and, obviously, statement (4.2) entails statement (4.1). Since statement (4.2) is a universal formula, completeness of the tautology checker now can be formulated by the $\mathcal{L}$-lemma

lemma $\vdash$ is complete $\iff \forall x: \text{IF.Expr}$

let{$\rho := \sigma \vdash x$}; if{$\text{return}(\rho), \neg \text{result}(\rho) \models x, \text{true}$} \,.

To prove this theorem, two further auxiliary lemmas are needed:

lemma $\vdash$ extends valuations $\iff \forall \sigma: \text{list}[\text{pair}[\mathbb{N}, \text{TruthValue}]], x: \text{IF.Expr}, n: \mathbb{N}$

let{$\rho := \sigma \vdash x$}; if{$\text{return}(\rho), (\nabla \sigma, n), \sigma \cap n = \text{result}(\rho) \cap n, \text{true}$} \,.

states that $\vdash$ computes an extension of the input valuation $\sigma$ indeed (provided $\vdash$ returns a valuation at all), and

lemma $\vdash$ falsifies $\iff \forall x: \text{IF.Expr}, \sigma: \text{list}[\text{pair}[\mathbb{N}, \text{TruthValue}]]$

let{$\rho := \sigma \vdash x$}; if{$\text{return}(\rho), \neg \text{result}(\rho) \models x, \text{true}$} \,.

states that $\vdash$ returns a valuation which falsifies the input formula $x$, provided $\vdash$ returns a valuation at all. This lemma is the “key lemma” of the completeness proof, and lemma “$\vdash$ is complete” follows immediately from lemma “$\vdash$ falsifies” simply by replacing $\sigma$ with $\phi$.

---

4 “< >” denotes list concatenation defined by procedure `<>` in Example 3.

5 This example illustrates the general approach for the elimination of existential quantifiers in order to formulate statements by $\mathcal{L}$-lemmas: For a conjecture of form (1) $\forall x: \tau_x \exists y: \tau_y \phi[x, y]$, a procedure $f$ with signature $f : \tau_x \rightarrow \tau_y$ has to be defined, such that (2) $\forall x: \tau_x \phi[x, f(x)]$ holds. Then with a proof of (2), (1) is proved as well.
5. Appendix

5.1. Character Set

$L$-expressions are formed of characters from a subset of Unicode.\(^1\)

5.2. Name Spaces

The following name spaces must be disjoint:

- the set of all type variables,
- the set of all type constructors,
- the set of all function symbols,
- the set of all formal parameters, local variables and variables of a lemma, and
- the set of all lemma names.

5.3. Naming Conventions

- **type variables**: any alphanumeric sequence of ASCII characters preceded by “@” and not containing blanks.
- **type constructors, function symbols, formal parameters, local variables, variables of a lemma**: any sequence of Unicode characters not preceded by “@”, “∇” or “∆” and not containing blanks as well as any sequence of Unicode characters beginning and ending with “/”.
- **lemma names**: any sequence of Unicode characters neither preceded by “@” nor containing “<=” as well as any sequence of Unicode characters beginning and ending with “/”.

---

\(^1\) **VeriFun** provides a table of all Unicode characters available for writing $L$-expressions.
5.4. Reserved Words and Symbols

∀ ( N all esac infixr postfix
: ) * bool false lemma pred
, { > case fi let prefix
; } = else function nat structure
+ [ ] , end if not succ
- | ¬ end_case in of then
<= := end_if infix other true
end let infixl outfix

as well as identifiers

- preceded by “∇” and not containing blanks,
- preceded by “∆^p” and not containing blanks, where p is a superscripted number,
- ending with “p-bounded”, where p is a number,
- ending with “p-projection”, where p is a number, or
- containing one of the symbols with code 0x2592, 0x22D5, 0x2019.

5.5. Separation of Symbols

The following symbols have to be enclosed in blanks:

- function symbols of fixity infix, infixl and infixr as well as the predefined symbols “=” (equality) and “>” (greater),
- the argument list of function symbols with fixity outfix,
- the negation operators “¬” and “not”,
- “:” when used in a type declaration or a case-expression.

5.6. Predefined and Implicitly Defined Procedures and Lemmas

- The data structures bool and N, cf. Example 1, and procedure “>”, cf. Example 3, are predefined in L.
- Certain lemmas about +, -, *, and > are predefined in L.\(^2\)
- Domain procedures \(\nabla f\) are implicitly defined for each function symbol \(f\) different from if and case, cf. Section 3.3.
- Difference procedures \(\Delta^p f\) are implicitly defined for some function symbols \(f\), cf. [WS05a].
- Boundedness- and projection lemmas “\(f\ p\text{-bounded}\)” or “\(f\ p\text{-projection}\)” are implicitly defined for some of the difference procedures \(\Delta^p f\).

\(^2\) These lemmas are collected in the folder Predefined which is part of each \(\text{VeriFun}\) session.
5.7. Aliases

The following aliases ease typing of $\mathcal{L}$-expressions:\footnote{The use of parentheses instead of curly brackets may introduce syntactical ambiguities an $\mathcal{L}$-parser cannot resolve. Therefore curly brackets have to be used in those rare cases.}

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{N}$</td>
<td>$\text{nat}$</td>
</tr>
<tr>
<td>$+$ (…)</td>
<td>$\text{succ}$ (…)</td>
</tr>
<tr>
<td>$-$ (…)</td>
<td>$\text{pred}$ (…)</td>
</tr>
<tr>
<td>{...}</td>
<td>(…)</td>
</tr>
<tr>
<td>end_if</td>
<td>end</td>
</tr>
<tr>
<td>end_case</td>
<td>end</td>
</tr>
<tr>
<td>end_let</td>
<td>end</td>
</tr>
<tr>
<td>$\forall$</td>
<td>$\text{all}$</td>
</tr>
<tr>
<td>$\neg$</td>
<td>$\text{not}$</td>
</tr>
<tr>
<td>$x_1 : \text{type}, \ldots, x_k : \text{type}$</td>
<td>$x_1, \ldots, x_k : \text{type}$</td>
</tr>
<tr>
<td>$+^{(\ldots)} (0 \ldots)$</td>
<td>$n$, where $n \leq 42$</td>
</tr>
<tr>
<td>$n$ times</td>
<td></td>
</tr>
</tbody>
</table>
actual parameter, 10
all, 14
ASCII, 30
bool, 4, 31
\(C(P)\), 16
\(C(P)_\tau\), 16
case, 6
case-expression, 6, 8
computation
  calculus, 17
  diverging, 16
  failing, 16
  rule, 17
  step, 17
  stuck, 16, 19
  succeeding, 16
conditional
  boolean, 6
  structural, 6
connective, 14
constructor function, 6
constructor ground term, 16
data structure, 4
domain procedure, 23, 31
end, 6, 11
d\_end\_case, 6
d\_end\_if, 6
eval\_p, 16, 20
except\_proc, 16
exception, 16
exception guard, 16
fair completion, 20, 21
fixity declaration, 7, 10
formal parameter, 10, 30
functional notation, 6, 11
\(G(P)\), 16
\(G(P)_\tau\), 16
ground term, 16
if, 6
if-expression, 6
in, 11
infix, 7, 8, 31
infixl, 7, 8, 31
infixr, 7, 8, 31
interpreter, 16
\(L\)-programm, 16
lemma, 13
let, 11
let-expression, 11, 14
local variable, 11, 14, 30
nat, 5
\(N\), 5, 31
not, 14
notation
  functional, 6, 11
  procedural, 6, 11
other, 6
outfix, 7, 31
parameter
  actual, 10
  formal, 10, 30
postfix, 7
precedence, 7, 8
predefined, 4, 10
prefix, 7
procedural notation, 6, 11
procedure, 9
  incompletely defined, 12
selector function, 6
structure predicate, 6–8
stuck computation, 16, 19
term, 6
  constructor ground, 16
  ground, 16
termination
  of procedures, 20
  of programs, 20
type, 4, 5
  constructor, 4, 30
  monomorphic, 4, 5
  polymorphic, 4, 5
  variable, 4, 30

UNICODE, 30

variable
  local, 11, 14, 30
  of a lemma, 13, 30

\ddownarrow_p, 17
\Rightarrow_p, 17
\Rightarrow^i_p, 17
\Rightarrow^l_p, 17
\Rightarrow^r_p, 17

=, 6, 31
>, 10, 31
*, 12, 16
Bibliography


