A user's guide to ALF *

Thorsten Altenkirch      Veronica Gaspes      Bengt Nordström
Björn von Sydow
Department of Computing Science
University of Göteborg/Chalmers
S-412 96 Göteborg Sweden
Draft June 14, 1994

Contents

1 Introduction                      2
2 A first example                   3
3 Description of the system         10
   3.1 The two main windows          10
   3.2 The mouse                     11
   3.3 The scratch area              11
      3.3.1 The File-menu             11
      3.3.2 The Define-menu           12
      3.3.3 The Construct-menu        12
      3.3.4 The Edit-menu             13
      3.3.5 The Goal-menu             14
      3.3.6 The Context-menu          14
      3.3.7 The View-menu             14
   3.4 The constraints               15
4 The ALF Libraries                15
   4.1 The Micro Library             15
   4.2 The Contributions Library    21
5 Bugs and improvements to be made 21
6 History of the system            22
7 Acknowledgements                 22
A Syntax                            22

*This research has been done within the ESPRIT Basic Research Action "Types for Proofs and Pro-
grams". It has been paid by NUTEK and Chalmers.
1 Introduction

ALF ("Another Logical Framework") is a structure editor for Martin-Löf’s monomorphic type theory; i.e., it ensures that the constructed objects are wellformed and welltyped \(^1\). It can be used for the development of proofs and programs and for the integrated verification of functional programs. ALF emphasizes the interactive development of type-theoretic constructions, i.e., proof objects and programs, using a window-based user interface. Thus ALF supports an arbitrary mixture of top-down and bottom-up development.

A logical framework is a pure dependent Type Theory. It is an open theory, i.e., the user can add constants and equations. In the current implementation of ALF there is no check whether the theory is consistent. However, the user can restrict herself to the set theory described in [13] — this is implemented as a library. New inductive sets can be added following [7, 6]. Recently Coquand has proposed to use pattern matching to introduce new non-canonical constants [3], see also the discussion in [5]. The interactive definition of proofs/programs by pattern matching is supported by ALF. See also [4] to get a recent description of the type theory used in ALF.

The basic metaphor of ALF is the refinement of an incomplete proof object which is displayed in a window (scratch area). By using the mouse the user can fill in placeholders by first pointing to them and then selecting a previously constructed object from a menu. ALF uses unification to fill in further placeholders automatically. If this is not yet possible ALF generates a set of constraints, which can also be manipulated from the user interface. Once the construction of the current object has been completed the object is moved to another more permanent window (theory window).

ALF has been used to formalize parts of intuitionistic mathematics or to verify simple programs:

- Nora Szasz has given a formal proof [14] that Ackermann’s function is not primitive recursive.
- Björn von Sydow [15] showed the fundamental theorem of arithmetic, i.e., that every integer is a product of a unique multiset of primes.
- K.V.S. Prasad and Karlis Cerans have made experiments in expressing proofs about various process calculi.
- Michael Hedberg constructs a category of semilattices and approximable mappings and show that it is cartesian closed and wellpointed.
- Veronica Gaspes showed [9] functional completeness of combinatorial logic, i.e., that every function can be compiled into an expression only involving the basic combinator S, K and I.
- Daniel Fridlender presented formal proofs of Higman’s lemma and of an intuitionistic version of Ramsey’s theorem. [8].
- Peter Dybjer and Thierry Coquand formalized a normalization proof for intuitionistic propositional logic using gluing.

\(^1\)See [10] and [11] for some details of the implementation.
• Jan Cederquist formalized the intuitionistic approach to pointfree topology and domain theory developed by Martin-Löf and Sambin.

• Thierry Coquand developed a semantical proof of Cut-Elimination for propositional logic and formalized a type checker for simply typed lambda calculus.

• Catharina Coquand formalized a semantic analysis of simply typed lambda calculus including a normalization proof [2].

• Bror Bjerner implemented the language P (based on while-loops and registers).

• Gustavo Betarte formalized the set $Z$ of integer and the proofs that $Z$ with addition and multiplication form an integral domain [1].

• Thorsten Altenkirch formalized different sorting algorithms (insertion-sort, merge-sort and quicksort).

ALF is an experimental tool and still very much in development; there are a number of known bugs - see section 5. Later version of ALF should inculde a consistency check and a program extraction facility.

2 A first example

In this section we give a detailed presentation of how the system is used to develop a small example. We define the set of natural numbers, define addition and the $\leq$ relation and prove two results: that $\leq$ is transitive and that $m \leq m + n$ for all natural numbers $m$ and $n$.

When ALF is started two windows appear on screen: the theory window and the scratch window. Both are split into subwindows by horizontal lines; the theory window consists of two subwindows and the scratch window of three. All five subwindows are initially empty. In the initial state only the basic type theory is known to the system. A theory is developed by making a sequence of definitions of various kinds. At the top of the scratch window we find a menu bar. We use the mouse to select (with any mouse button) the Define entry and the following menu appears:

<table>
<thead>
<tr>
<th>File</th>
<th>Define</th>
<th>Construct</th>
<th>Edit</th>
<th>Goal</th>
<th>Matching</th>
<th>Context</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch</td>
<td>Explicit Constant...</td>
<td>Implicit Constant...</td>
<td>Set...</td>
<td>Type...</td>
<td>Context...</td>
<td>Substitution...</td>
<td></td>
</tr>
</tbody>
</table>

The different entries represent the kinds of objects that can be defined. We will introduce them one at a time and start to define the set of natural numbers. So we move the mouse cursor to the entry Set... and release the mouse button. A small text editor window pops up and prompts us to give the name of the set to be defined. We type the name $N$:
We then strike the Return key (or click on the Ok button) and the pop-up window disappears. Instead the incomplete definition appears in the scratch area, which is the top subwindow of the scratch window:

```
Scratch Area
N. ∈ . ?_N, T [ ]
```

The first reason that the definition is incomplete is that the type of the new constant $N$ has not yet been specified, as can be seen from its type $?_N, T$. For the moment we ignore the significance of the subscript. It suffices to know that we have to complete the definition of this placeholder. We also ignore the empty pair of brackets following the definition. So we select the placeholder, using the left mouse button. It appears in inverted video and we click on the right mouse button to get a menu with the available options:

```
N. ∈ . ?_N, T [ ]
Set
Prop
(?) ?
(x ∈ ?) ?
Paste
Edit As Text...
N. ∈ . ?_N, T
```

The choices **Set** and **Prop** represent the two predefined ground types of sets and propositions, respectively. (These two are actually synonyms, because of the identification of proposition and sets). The two next choices represent non-dependent and dependent function types. There is also the possibility to paste a type from some previous definition or to invoke the text editor. These six items are always present in this menu when the selection is a type expression. At the end there is a context-sensitive part, where additional possibilities may appear. In our case we choose the first item, i.e. **Set** and the placeholder is replaced by our choice. You may wonder at this point why we had to make this choice since we already had indicated that we wanted to define a set. This will be explained below, when we define $\leq$.

It is now clear to the system that the name $N$ denotes a set. However, to complete the definition we have to give the constructors of the set. In this case the constructors are 0
and the successor function. To define these, we select \( N \) (left mouse button!) and invoke the menu of options (right button!): 

\[
\begin{array}{|c|}
\hline
\text{N.\in\text{Set}} & [] \\
\hline
\text{Make constructor...} & \\
\text{Copy} & \\
\text{Clear} & \\
\text{Complete} & \\
\hline
\end{array}
\]

It should be obvious that the choice should be **Make constructor**... The text editor window pops up and we type 0 as the name of the first constructor. When the window closes, we see in the scratch area that we have to give the type of 0. We select the placeholder for the type and invoke the command menu. Now the final, context-sensitive item \( N \in \text{Set} \) is the correct choice\(^2\) and the situation is the following:

\[
\begin{array}{|c|}
\hline
\text{Scratch Area} & \\
N.\in\text{Set} & [] \\
0.\in N & [] \\
\hline
\end{array}
\]

As should be apparent already now, the ALF user does almost all work by selecting items using the left mouse button and invoking the pop-up menu of commands applicable to the selection with the right button. Occasionally some text has to be typed in the pop-up text editor.

We are not yet finished with \( N \); there is one more constructor. So we select \( N \), choose **Make constructor**... from the command menu and type \( \text{succ} \) in the editor. The next step is to give the type of \( \text{succ} \). Since it is a function, we should choose a function type from the command menu. The type of the successor function is non-dependent, so we could use the (?) entry. However, if we have a preferred variable name to use for an unspecified argument to the function, it is convenient to indicate this now. This will be explained in more detail below, so let us for the moment accept that it is a good idea to choose the \( (x \in ?) \) entry. The text editor prompts us for a name and we type \( n \). There are now two placeholders in the type for \( \text{succ} \) and we select them in turn and use the pop up menu to set them to \( N \). The definition of the set of natural numbers is now complete:

\[
\begin{array}{|c|}
\hline
\text{N.\in\text{Set}} & [] \\
\text{succ.\in\,(n \in N) N} & [] \\
0.\in N & [] \\
\hline
\end{array}
\]

Before we continue, we consider briefly what one can do in case of mistakes. Select the type of \( \text{succ} \), i.e. \( (n \in N) N \). In doing this, note that the selection is always a complete subexpression. Thus the best way of selecting the entire type is to click on one of the parentheses, since the smallest enclosing expression is then the entire type. Now invoke the pop up menu and choose **Clear**. The type is replaced by a placeholder and we may

\(^2\)Note that the type of \( N \) is known now, which it was not the previous time we saw the menu.
start again to give the type\(^3\). To do this, we use another method: We double-click on the place-holder and the text editor pops up. We may now give the entire type (using colon for \(\in\)): \((\mathbb{N}, \mathbb{N})\). This is a general feature; by double-clicking when selecting, the text editor pops up and one may edit the expression as text. Of course, this may produce a syntactically incorrect expression. In that case, one gets an error message and may open the editor again to correct the error.

The definition of \(\mathbb{N}\) is now complete. We select it (click on \(\mathbb{N}\)) and invoke Move-to-theory from the pop-up menu. The definition disappears from the scratch area and appears instead in the lower half of the theory window. Completed definitions should be moved to the theory. If it is necessary to change an already completed definition one has to move it back to the scratch area first by selecting it and choosing Move-to-scratch. If necessary multiple definitions have to be selected (\(\text{shift left-button}\)) and moved at once.

As our next step, we define the constants 1 and 2 as abbreviations or, in ALF terminology, \textit{explicit constants}. To do this, open the Define menu and choose Explicit constant.... When prompted for the name, we answer 1 and the result in the scratch area is

\[
1 \equiv \text{succ}(?_n) \in \mathbb{N} \quad [\] 

Both the type and the value of this constant have to be defined. We may start with either. We choose the type, so we select this placeholder and find the choice \(\mathbb{N}\) in the pop-up menu. Next, we select the value placeholder and find \textit{succ} in the menu. Choosing this will have the effect

\[
1 \equiv \text{succ}(?_n) \in \mathbb{N} \quad [\] 

i.e., a new placeholder is inserted for the argument to \textit{succ}. The type-checker of ALF has realized that if 1 is to be a natural number, it cannot be \textit{succ} itself, but rather \textit{succ} applied to a natural number. We complete the definition by selecting the remaining placeholder and choosing 0 from the menu. Here we can also note that in the menu we are also offered the choice \(1 \in \mathbb{N}\), which would give the circular definition \(1 \equiv \text{succ}(1) \in \mathbb{N}\). If this choice is attempted, an error message results. We see from this example that the context-sensitive choices offered are somewhat roughly computed and may not always be possible to select. In a similar way we define 2 as \textit{succ}(1).

We go on to define addition. This can be done by a pair of recursive equations with case analysis on one of the arguments. In ALF terminology, this is an \textit{implicit constant}. We note here also that the present version does not admit infix operators, so we prefer the name \textit{plus}. The first step is to choose \textit{implicit constant}... from the Define menu and reply with this name when prompted. The next step is to give the type of the constant. We choose to do this with the text editor (double-click!):
To fill in the definition, we select the name `plus` and choose `Make pattern` from the menu. A defining equation with a placeholder as right-hand side appears. However, we cannot fill in this placeholder yet. We want to do case analysis on the first argument (the choice of the first rather than the second is arbitrary), so we select the parameter `m` in the left-hand side and invoke `Make pattern` again. The equation is split in two, one for each possible constructor form of `N`. Now the right hand sides of the equations can be easily filled in, by a sequence of selections and choices from the pop-up menu.

\[
\begin{align*}
\text{plus} & . \in \text{plus} \\
\text{plus}(U, n) & . \equiv . n \\
\text{plus}(\text{succ}(n), n) & . \equiv . \text{succ}(\text{plus}(n, n))
\end{align*}
\]

We have defined `plus` by `pattern-matching` on its first argument. In such definitions, we allow recursive calls. To ensure that a recursive definition leads to a well-defined function it is necessary that there is an argument position in which the recursive call has a `structurally smaller` argument. In the case of `plus`, the first argument has this property. **The present version of ALF does not enforce this condition.** It is thus possible in ALF to make meaningless recursive definitions, such as \(f(x) = f(x)\). The user must manually check that recursive definitions are well-formed.

We may now move our completed definitions to the theory and go on to define `Leq`, the relation “less than or equal to” on `N`. For any two natural numbers `m` and `n`, `Leq(m,n)` is a proposition or, equivalently, a set. The ALF action to define this is therefore to choose `Set...` from the `Define` menu and choose the name `Leq` when prompted by the text editor. This leaves us in the situation

\[
\begin{align*}
\text{Leq} & . \in . ?\text{Leq} \\
\end{align*}
\]

and we now see why we have to give the type of `Leq` also in the `Set...` case. `Leq` is not a set itself; rather it is a function that given two natural numbers produces a set. We say that `Leq` is a family of sets. So we may double-click on the placeholder for the type and type \((m,n:N)\text{Set}\). We note that we have our first example of a dependent set; for each choice of `m` and `n` we have a distinct set (proposition), whose elements are the proofs of the proposition. Of course, this will mean that `Leq(2,1)` will have to be defined to be empty, while `Leq(1,2)` will be inhabited. Next we have to give the constructors. There are many ways to define `Leq`. A convenient one is an inductive definition: we have `Leq(0,n)` for any `n` and also that `Leq(\text{succ}(m),\text{succ}(n))` if `Leq(m,n)`. These two cases will give the constructors
of the set. We have to invent two names for the two cases and define the constructors as before:

\[
\begin{align*}
\text{leq}_0 & \in \forall (m, n \in \mathbb{N}) \text{Set} \quad [m] \\
\text{leq succ} & \in \forall (n \in \mathbb{N}) \text{leq}(0, n) \quad [n] \\
\text{leq succ succ} & \in \forall (m, n \in \mathbb{N}; p \in \text{leq}(m, n)) \text{leq}(\text{succ}(m), \text{succ}(n)) \quad [n]
\end{align*}
\]

To help understand this definition, we can check that \(\text{leq}_0(1)\) is a proof (element) of \(\text{leq}(0, 1)\) and \(\text{leq succ}(0, 1, \text{leq}_0(1))\) a proof of \(\text{leq}(1, 2)\). As a more interesting example, we now prove that \(\text{leq}\) is transitive. This theorem is represented by the set \((m, n, k; N; p: \text{leq}(m, n); q: \text{leq}(n, k))\text{leq}(m, k)\). To prove it, we have to define a constant \(\text{leq trans}\) of this type. Since the theorem involves arbitrary natural numbers, we define this as an implicit (recursive) constant. The first steps, including the first Make pattern, give the situation

\[
\begin{align*}
\text{leq trans} & \in \forall (m, n, k \in \mathbb{N}; p \in \text{leq}(m, n); q \in \text{leq}(n, k)) \text{leq}(m, k) \quad [m] \\
\text{leq trans} & \in \forall (m, n, k, p, q) \text{leq trans}(m, n, k, p, q) \equiv \ldots \text{leq trans } 0 \text{ E}
\end{align*}
\]

It now seems natural, following the ideas in defining \textit{plus}, to do pattern matching on one of the natural number arguments. However, it turns out to be even better to do pattern matching on the \textit{proof} arguments \(p\) and \(q\). To do this, we first have to move the definition of the family \text{leq} to the theory, since we are not permitted to do pattern matching on an argument whose type is in the scratch area. (Such a definition could later be extended with another constructor, which would invalidate the pattern matching done). After having done this, we select \(p\) and invoke Make pattern. The following happens:

\[
\begin{align*}
\text{leq trans} & \in \forall (m, n, k \in \mathbb{N}; p \in \text{leq}(m, n); q \in \text{leq}(n, k)) \text{leq}(m, k) \quad [m] \\
\text{leq trans} & \in \forall (m, n, k, q) \text{leq trans}(m, n, k, q) \equiv \ldots \text{leq trans } 0 \text{ E}
\end{align*}
\]

The two constructors for \text{leq} appear in the respective cases, but some arguments have been replaced by a \textit{wildcard} \_. This is because there is a dependency between the arguments. In the first equation, the third argument \text{leq}_0(\_) has (according to the typing of \text{leq trans}) type \text{leq}(m, n). On the other hand, according to the definition of \(\text{leq}\), \text{leq}_0(\_x) has for any \(x\) type \text{leq}(0, x). These two types are now \textit{unified} and ALF is able to conclude that \(m\) must be \(0\) and that also \(x = n\). The latter is enforced by choosing \(n\) as argument to \text{leq}_0. Such inferred arguments are not displayed by the system but instead shown as a wildcard. This device maintains the patterns in \textit{linear} form, i.e. no variable appears more than once.

The first equation can now easily be completed. To do this, we select the right hand side. Here we note a fact that we have not mentioned before. The small bottom subwindow of the scratch window displays the type of the selection, in this case \text{leq}(0, k). An element of this type is obvious: it is \text{leq}_0(k), which we construct by choosing \text{leq}_0 from the pop-up menu. The argument \(k\) is inferred by the system and filled in automatically. It remains to define the right hand side of the second equation. Selecting it we see its type \text{leq succ}(m1, k)) at the bottom of the theory window. We cannot directly give a proof of
this; \texttt{leq\_succ} requires both its first two arguments to be on successor form. Thus we do
pattern matching on \( q \). Now something interesting happens; only one of the cases appear:

\begin{verbatim}
leq\_trans e. (m, n, k e N; p e Leq(m, n); q e Leq(n, k)) Leq(m, k) []
leq\_trans e. (m, n, k e N; leq\_O(lq, g)) e. leq\_O(k)
leq\_trans e. (m, n, k e N; leq\_succ(m_1, n_1, p_1), leq\_succ(n, n, q)) e. leq\_succ(m, n, q)
\end{verbatim}

The reason for this is apparent when the situation is analyzed: Since \( p \) now has constructor \texttt{leq\_succ}, \( n \) cannot be \( 0 \) and thus the proof \( q \) cannot have constructor \texttt{leq\_O}. The type of the right hand side of the remaining case is \( \text{Leq}(\text{succ}(m_1), \text{succ}(n)) \) and
an application of \texttt{leq\_succ} seems appropriate. We try this and the right hand side is partly
completed to \( \text{leq\_succ}(m, n, ?) \), where the remaining placeholder has type \( \text{Leq}(m_1, n) \). The
situation is the following, where we have opened the pop-up menu to see what is available
to us:

\begin{verbatim}
leq\_trans e. (m, n, k e N; p e Leq(m, n); q e Leq(n, k)) Leq(m, k) []
leq\_trans e. (m, n, k e N; leq\_O(lq, g)) e. leq\_O(k)
leq\_trans e. (m, n, k e N; leq\_succ(m_1, n_1, p_1), leq\_succ(n, n, q)) e. leq\_succ(m, n, q)
\end{verbatim}

\begin{verbatim}
[x]?
[x...]? 
Paste
Edit As Text...

\( p_1 \) e. \( \text{Leq}(m_1, n_1) \)
\( p \) e. \( \text{Leq}(n_1, n) \)
leq\_succ e. (m, n e N;
\( p \) e \( \text{Leq}(m, n) \)) \( \text{Leq}(\text{succ}(m), \text{succ}(n)) \)
leq\_O e. (n e N) \( \text{Leq}(0, n) \)
leq\_trans e. (m, n, k e N;
\( p \) e \( \text{Leq}(m, n) \);
\( q \) e \( \text{Leq}(n, k) \)) \( \text{Leq}(m, k) \)
\end{verbatim}

We see that we have \( p_1 \) and \( p \) available, which would give us a term of the required
type by a recursive application of \texttt{leq\_trans} itself. We make this choice and have to fill in
the arguments \( p_1 \) and \( p \), which cannot be inferred by the system. This completes the proof
and it remains only to check that the recursive call is on a structurally smaller argument.
This is indeed the case, as the third argument in the recursive call is on \( p_1 \), which is one of
the arguments to the constructor in the corresponding position on the left hand side.

Finally, we prove the simple result \( \text{leq\_plus}(m, n; N) \text{Leq}(m, \text{plus}(m, n)) \). We define this as
an implicit constant and do pattern matching on \( m \). This gives us

\begin{verbatim}
leq\_plus e. (m, n e N) \( \text{Leq}(m, \text{plus}(m, n)) \) []
leq\_plus(0, n) e. \= \( \text{leq\_plus} 0.0.0.E \)
leq\_plus(succ(n_1), n) e. \= \( \text{leq\_plus} 0.1.E \)
\end{verbatim}
Selecting the right hand side of the first equation, we find that its type is $\text{Leq}(0, \text{plus}(0, n))$. But according to the definition of $\text{plus}$, $\text{plus}(0, n)$ is definitionally equal to $n$, which means that the required type is the same as $\text{Leq}(0, n)$, for which we can easily find the proof $\text{leq}_0(n)$. Similarly, in the second case, ALF can use the second defining equality for $\text{plus}$ to compute the type of the right hand side to $\text{Leq}(\text{succ}(m), \text{succ}(\text{plus}(m, n)))$ and we can complete the proof with $\text{leq}_\text{succ}$, using a recursive call to $\text{leq}_\text{plus}$ as third argument:

\begin{align*}
\text{leq}_\text{plus} & \in (m, n \in \mathbb{N}) \ \text{Leq}(m, \text{plus}(m, n)) \quad [] \\
\text{leq}_\text{plus}(0, n) & = \text{leq}_0(\text{plus}(0, n)) \\
\text{leq}_\text{plus}(\text{succ}(n_2), n) & = \text{leq}_\text{succ}(n_2, \text{plus}(n_2, n), \text{leq}_\text{plus}(n_2, n))
\end{align*}

3 Description of the system

3.1 The two main windows

Two main windows are opened when ALF is started.

One window presents the theory and the other the scratch area. There are also two minor windows, one window showing the type of the current expression, and one window showing the constraints which must hold. This will be explained later.
The theory is a list of constants with their types and definition. The upper part of the theory window contains the imported theories, i.e. constants which reside in files which have been loaded by the system. Imported theories cannot be changed, they can only be changed by invoking the editor on the files they reside in. The other constants, the current theory, can be changed by first moving them to the scratch area.

The scratch area is a place where objects and types are being edited, so it contains incomplete objects and types. These objects can be built using the constants defined in the current theory and in the scratch area. When an object or type is completely built, it is moved to the theory window by the user. Below the scratch area there is a subwindow containing a set of constraints and a window showing the type of the selected expression.

Before we describe the menus which can be used in the two windows we have to explain how the mouse is used.

3.2 The mouse

We use a mouse with three buttons. The left button is used to select a subexpression or a definition. By pressing the shift key it is possible to select more than one definition. This is used when several constants are to be moved simultaneously between the scratch area and the theory window. A double click on the left button opens a text editor on the selected expression. After the text has been edited, it will be parsed and typechecked and will replace the selected expression. The middle button is used to replace a placeholder with an application of a constant. So, first select a placeholder (with the left button) and then move the mouse to the definition of the constant and click on the middle button. A click on the right button will pop up a menu containing all commands which are applicable at the selection. If the selection is a placeholder, then it will also give an approximation of the list of all the applicable constants and locally bound variables. The placeholder will be replaced by a variable by choosing the variable in the list, and it will be replaced by an application of the constant which has been chosen.

3.3 The scratch area

The main menus in the scratch area are the following:

3.3.1 The File-menu

This menu is used for interaction with the file system, to read and write theories, to start a new theory and to quit the editor. Files will be searched in an order decided by the environment variable ALFPATH. This has a default setting, but can be changed by for instance putting the command `setenv ALFPATH directory1: directory2: ...` in the file `.cshrc`.

The file menu contains the following entries:

New... Creates a new empty theory.

Open... Restores the editor to a previously saved state.

Save Saves the current state of the editor.
Save As... Saves the current theory and scratch area with a new name. This will also be done automatically every time the complete command is used to move a definition from the scratch area to the theory window. It will then be saved in the file

```
##BACKUP_ALF##
```

Revert Reads the theory and the scratch area again, so that all changes becomes undone.

Import... Reads a file into the theory. The content of this file cannot be changed. The only way to change the content of a theory file is to open it.

Print... Produces a postscript file (default alfout.ps) of the Imports, Main Theory, Scratch area or Constraints. The postscript file can be previewed or printed directly or included in a TeX document by using a package like epsf.  

Quit Quits the editor.

### 3.3.2 The Define-menu

When you start to build something in the scratch area you first have to tell what kind of thing you are going to build and then giving it a name. This is done with selections from this menu.

**Explicit Constant** Create an explicitly defined constant. This has a definiendum (a name, a left hand side), a definiens (a right hand side), a type and a context. After giving the name the other parts are filled in by the user by replacing place-holders (question marks) with other expressions.

**Implicit Constant** Create a new implicitly defined constant. This is a constant which is defined by pattern-matching and possibly recursively defined. After giving it a name, you tell what type it should have (by incrementally editing placeholders).

**Set** Create a new a set or a family of sets.

**Type** Create an explicitly defined constant denoting a type.

**Context** Create a named context.

**Substitution** Create a named substitution. Not implemented yet.

### 3.3.3 The Construct-menu

When the current selection is a placeholder it is possible to replace the placeholder with different new incomplete expressions. Depending on where the placeholder is, different replacements can be made. These are the alternatives:

[x]? In the case that the expected type of a placeholder is a function or a place holder then it is possible to replace it by an abstraction.

---

4Note that it may be worthwhile to use postscript software to adjust the bounding box.

5Contexts consists of list of variable declarations, i.e. a list of variables together with their type. Every constant is defined in a context and the variables in the context can be given a name by using a substitution. This is in an experimental phase and only partly implemented.
[x...]? This replaces the placeholder with a repeated abstraction.

**case** ? Replaces the placeholder with a case-expression. This can only be done if the placeholder is the righthand side of an equation of an implicitly defined constant or is the righthand side of a branch of another case expression. These restrictions will be removed.

**Make a pattern** If the current selection is a typing of an implicitly defined constant or a variable in the lefthand side of an implicitly defined constant then this command will create new defining equations for this constant. This command is also used to create a pattern in a case-expression.

**Set** Replaces the current placeholder with the expression Set.

**Prop** Replaces the current placeholder with the expression Prop.

(?)? Replaces the current placeholder with the expression (?1)?2, i.e. a non-dependent function type.

(x : ?)? Asks for a variable name x and replaces the current placeholder with an expression (x ∈?1)?2 which is a template for a dependent function type.

**Make a constructor** If the current selection is a typing of a set or a family of sets, then this command will ask the user to input the name of a new constructor for this set. As an example, if the current selection is

\[ N \in \text{Set} \]

then the user is asked to give the name of a constructor of \( N \).

[] Creates an empty context. Not implemented yet.

... + ? Adds a context to a context. Not implemented yet.

x : ? extends a context with the declaration of a new variable (which is prompted for).

{} Creates an empty substitution. Not implemented yet.


### 3.3.4 The Edit-menu

**Undo** Undo the effects of the last command.

**Cut** Replaces the current selection with a placeholder and inserts the selected expression in the cut buffer.

**Copy** Copies the current selection into the cut buffer.

**Paste** Inserts the content of the cut buffer into the current placeholder.

**Clear** Replaces the current expression with a placeholder and deletes all expressions which are consequences of that expression.  

---

*The implementation of Clear is based on the concept of a local undo; details can be found in [10].*
Edit as Text  Asks the user to input text, which is parsed and replaces the current placeholder.

Unfold  Replace the current selection with its definition (if it is an explicit definition).

Normal Form  Replaces the current selection with its value in normal form.

Head Normal Form  Replaces the current selection with its value in head normal form.

Expand  If the current selection is a definition of an explicitly defined constant then this constant is removed from the scratch area and all occurrences of it is replaced by its definiens. Not implemented yet.

Move to theory  Moves a definition from the scratch area to the theory window.

Move to scratch  Moves a definition from the theory window to the scratch area.

3.3.5 The Goal-menu
Lists all remaining placeholders and their type. You can select a subgoal from this menu.

3.3.6 The Context-menu
Prints the context of the current selection.

3.3.7 The View-menu
This menu is used to change the presentation of expressions.

Show all arguments / hide arguments  This is a toggle which says if the system should show or hide the hidden arguments in an application.

Layout  This is used to hide the first arguments of a functional constant. You are asked to increment a counter to the number of hidden arguments. You can also show how to display the constant using other fonts (subscripts, a symbol font, different font sizes etc).

Normal Form  Shows the normal form of an object.

Head Normal Form  Shows the head normal form of an object.

Command...  Gives the user a possibility to give a command directly to the proof engine. It is intended for people who know the commands of the proof engine. It can be used for bug reports: you type history f to write on file f a command history of the present state. You can also use it to compute the normal form of an expression, etc.

Full view / Restricted view  Normally, only the type of the constants in the theory is shown. By selecting a constant and applying this menu, the definitions of the constant will be shown.
3.4 The constraints

The scratch area also shows a list of constraints, this is a set of definitional equalities which must hold for the scratch area to represent well formed partial expressions. Constraints are being added or deleted as an object is being edited.

It is possible to select a constraint and solve it by choosing “solve” with the right button. This will take a constraint of the form

\[ ?(x) \equiv e \]

where \( e \) does not contain placeholders and replace all occurrences of \( ? \) with \([x]e \) in the scratch area.

4 The ALF Libraries

The ALF libraries provide the user with a collection of theories that may be imported on demand. The libraries contain commonly used sets, functions and properties of these. Two libraries are provided: a very basic one (the Micro Library) containing most commonly used sets; and a more elaborate one (the Contributions Library) where one finds complete developments in ALF.

The libraries are organized in theories where all sets, functions and properties relating to some concept are defined. In order to use the constants defined in some theory, one must import the theory. This is done by using the Import entry in the File menu. ALF then prompts the user for the name of the theory. The user replies with a theory name, say \texttt{Nat}. ALF tries to find a file \texttt{Nat.alf} in any of the directories listed in the environment variable \texttt{ALFPATH}. If this search is successful, the file is loaded and displayed in the upper half of the ALF theory window. Imported theories may be used but not changed or amended. When the user saves her current theory (the \textit{open} theory, which is displayed in the lower half of the theory window), the system records all imported theories and automatically imports these the next time the current theory is opened. The ALF libraries reside in subdirectories of a common site-dependent root directory.

When importing these theories to ALF, applications of some of the constants are displayed by default with an initial sequence of arguments suppressed. This is indicated on screen by a vertical arrow (\( \downarrow \)) in front of the argument name in the type of the constant. This is intended to improve readability, but can be changed using the command \texttt{Show all arguments} in the \textit{View} menu. Some of the constants (such as \texttt{Sigma}) are displayed on screen using an extended font; these cases should be self-explanatory.

4.1 The Micro Library

This library collects very basic theories concerning the most commonly used sets together with some elementary properties. It may be imported alltogether invoking \texttt{Import} on \texttt{microlib}. A core theory provides an implementation of the sets presented in [13], though without the constants corresponding to the elimination rules. The theory \texttt{rel} provides constants to characterize binary relations, it formally defines what it means for a binary relation to be reflexive, symmetric, transitive, decidable, total and defines the accessible part of a relation. In the theory \texttt{id} properties of propositional equality are proved. The theory \texttt{algebra} contains definitions characterizing semigroups, monoids and commutative
monoids. The theories nat, lists and vec define natural numbers, lists and arrays, together
with some functions, relations and properties of these. We now list the theories of the
microlibrary. In this list, the primitive constants are shown as they stand, but for the
defined constants only the name and typing is shown (the definitions themselves have
been hidden, one may inspect them by importing the theories). In writing this library we
have followed the following conventions:

1. names of sets start with a capital letter,
2. names of elements of sets start with a small letter,
3. different words in a name are commenced with a capital letter,
4. for sets with one constructor the same name is used both for the set and the con-
   structor.

core
Empty ∈ Set
univ ∈ (↓A ∈ Set; Empty) A
Not ∈ (Set) Set
    not ∈ (↓A ∈ Set; (A) Empty) Not(A)
Unit ∈ Set
    unit ∈ Unit
Bool ∈ Set
    tt ∈ Bool
    ff ∈ Bool
Σ ∈ (A ∈ Set; B ∈ (A) Set) Set
    pair ∈ (↓A ∈ Set; ↓B ∈ (A) Set; a ∈ A; b ∈ B(a)) Σ(a, b)
Prod ∈ (Set; Set) Set
    prod ∈ (↓A, ↓B ∈ Set; a ∈ A; b ∈ B) Prod(A, B)
Plus ∈ (A, B ∈ Set) Set
    inl ∈ (↓A, ↓B ∈ Set; a ∈ A) Plus(A, B)
    inr ∈ (↓A, ↓B ∈ Set; b ∈ B) Plus(A, B)
Lift ∈ (Set) Set
    bot ∈ (A ∈ Set) Lift(A)
    in ∈ (↓A ∈ Set; A) Lift(A)
Π ∈ (A ∈ Set; B ∈ (A) Set) Set
    λ ∈ (↓A ∈ Set; B ∈ (A) Set; f ∈ (a ∈ A) B(a)) Π(A, B)
Fun ∈ (A, B ∈ Set) Set
    fun ∈ (↓A, ↓B ∈ Set; f ∈ (A) B) Fun(A, B)
Id ∈ (↓A ∈ Set; a, b ∈ A) Set
    id ∈ (↓A ∈ Set; x ∈ A) Id(x, x)
I ∈ (↓A ∈ Set; A)
o ∈ (↓A, ↓B, ↓C ∈ Set; (B) C; (A) B; A) C
rel
Refl ∈ (\downarrow A ∈ Set; R ∈ (A; A)Set) Set
refl ∈ (\downarrow A ∈ Set; R ∈ (A; A)Set; (a ∈ A) R(a, a)) Refl(R)
Sym ∈ (\downarrow A ∈ Set; R ∈ (A; A)Set) Set
sym ∈ (\downarrow A ∈ Set;
\downarrow R ∈ (A; A)Set;
(a, a' ∈ A; R(a, a')) R(a', a)) Sym(R)
Trans ∈ (\downarrow A ∈ Set; R ∈ (A; A)Set) Set
trans ∈ (\downarrow A ∈ Set;
\downarrow R ∈ (A; A)Set;
(a, a', a'' ∈ A; R(a, a'); R(a', a'')) R(a, a'')) Trans(R)
Dec ∈ (\downarrow A ∈ Set; P ∈ (A)Set) Set
dec ∈ (\downarrow A ∈ Set; \downarrow P ∈ (A)Set; (a ∈ A) Plus(P(a), Not(P(a)))) Dec(P)
DecR ∈ (\downarrow A ∈ Set; R ∈ (A; A)Set) Set
decR ∈ (\downarrow A ∈ Set;
\downarrow R ∈ (A; A)Set;
(a, a' ∈ A) Plus(R(a, a'), Not(R(a, a')))) DecR(R)
Tot ∈ (\downarrow A ∈ Set; R ∈ (A; A)Set) Set
tot ∈ (\downarrow A ∈ Set;
\downarrow R ∈ (A; A)Set;
(a, a' ∈ A) Plus(R(a, a'), R(a', a)) Tot(R)
Acc ∈ (\downarrow A ∈ Set; (A; A)Set; A) Set
acc ∈ (\downarrow A ∈ Set;
R ∈ (A; A)Set;
\downarrow a ∈ A;
\downarrow b ∈ A; R(b, a)) Acc(R, b) Acc(R, a)

id
substId ∈ (\downarrow A ∈ Set; \downarrow a, \downarrow b ∈ A; Id(a, b); C ∈ (A)Set; C(a)) C(b)
resId ∈ (\downarrow A ∈ Set; \downarrow a, \downarrow a' ∈ A; \downarrow f ∈ (A)B; Id(a, a')) Id(f(a), f(a'))
refId ∈ (\downarrow A ∈ Set) Refl(Id())
symId ∈ (\downarrow A ∈ Set; a, b ∈ A; Id(a, b)) Id(b, a)
transId ∈ (\downarrow A ∈ Set; a, b, c ∈ A; Id(a, b); Id(b, c)) Id(a, c)
transId ∈ (\downarrow A ∈ Set) Trans(Id())
algebra

\[ \text{Assoc} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{assoc} \in (A \in \text{Set}; \ op \in (A;A)A; \ (x, y, z \in A) \text{Id}(op(op(x, y), z), op(x, op(y, z)))) \text{ Assoc}(A, op) \]

\[ \text{Comm} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{comm} \in (A \in \text{Set}; \ op \in (A;A)A; \ (x, y \in A) \text{Id}(op(x, y), op(y, x))) \text{ Comm}(A, op) \]

\[ \text{WithUnitL} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{withUnitL} \in (A \in \text{Set}; \ op \in (A;A)A; \ e \in A; \ (x \in A) \text{Id}(op(e, x), x)) \text{ WithUnitL}(A, op) \]

\[ \text{Semigroup} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{semigroup} \in (A \in \text{Set}; \ op \in (A;A)A; \ \text{Assoc}(A, op)) \text{ Semigroup}(A, op) \]

\[ \text{WithUnitR} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{withUnitR} \in (A \in \text{Set}; \ op \in (A;A)A; \ e \in A; \ (x \in A) \text{Id}(op(x, e), x)) \text{ WithUnitR}(A, op) \]

\[ \text{Monoid} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{monoid} \in (A \in \text{Set}; \ op \in (A;A)A; \ \text{Assoc}(A, op); \ \text{WithUnitL}(A, op); \ \text{WithUnitR}(A, op)) \text{ Monoid}(A, op) \]

\[ \text{CommMonoid} \in (A \in \text{Set}; \ op \in (A;A)A) \text{ Set} \]
\[ \text{commMonoid} \in (A \in \text{Set}; \ op \in (A;A)A; \ \text{Assoc}(A, op); \ \text{Comm}(A, op); \ \text{WithUnitL}(A, op); \ \text{WithUnitR}(A, op)) \text{ CommMonoid}(A, op) \]
nat
Nat ∈ Set
0 ∈ Nat
s ∈ (Nat) Nat
Lt ∈ (Nat; Nat) Set
l0 ∈ (i ∈ Nat) Lt(0, s(i))
l1 ∈ (i, j ∈ Nat; Lt(i, j)) Lt(s(i), s(j))
add ∈ (Nat; Nat) Nat
injS ∈ (i, j ∈ Nat; Id(s(i), s(j))) Id(i, j)
n0ConfNat ∈ (i ∈ Nat; Id(0, s(i))) Empty
eqNat1 ∈ (i, j ∈ Nat) Plus(Id(i, j), Not(Id(i, j)))
eqNat ∈ DecR(Id())
Le ∈ (Nat; Nat) Set
ltLem1 ∈ (i ∈ Nat; Lt(i, 0)) Empty
ltLem2 ∈ (i, j ∈ Nat; Lt(i, s(j))) Plus(Id(i, j), Lt(i, j))
accLtaux ∈ (i ∈ Nat; (j ∈ Nat; Lt(j, i)) Acc(Lt, j); k ∈ Nat; Lt(k, s(i))) Acc(Lt, k)
accLt ∈ (i ∈ Nat) Acc(Lt, i)
ltLem3 ∈ (m, n ∈ Nat; Lt(s(m), s(n))) Lt(m, n)
decRLt1 ∈ (m, n ∈ Nat) Plus(Lt(m, n), Not(Lt(m, n)))
decRLt ∈ DecR(Lt)
trichLt ∈ (m, n ∈ Nat) Plus(Lt(m, n), Plus(Lt(n, m), Id(n, m)))
totalLe ∈ (m, n ∈ Nat) Plus(Le(m, n), Le(n, m))
leSuccLem ∈ (m, n ∈ Nat; Le(m, n)) Lt(m, s(n))
ltSuccLem ∈ (n ∈ Nat) Lt(n, s(n))
assocAdd1 ∈ (x, y, z ∈ Nat) Id(add(add(x, y), z), add(x, add(y, z)))
commAddLem1 ∈ (x ∈ Nat) Id(x, add(x, 0))
commAddLem2 ∈ (x, y ∈ Nat) Id(s(add(x, y)), add(x, s(y)))
commAdd1 ∈ (x, y ∈ Nat) Id(add(x, y), add(y, x))
withUnitLAdd1 ∈ (x ∈ Nat) Id(add(0, x), x)
withUnitRAdd1 ∈ (x ∈ Nat) Id(add(x, 0), x)
assocAdd ∈ Assoc(Nat, add)
commAdd ∈ Comm(Nat, add)
withUnitLAdd ∈ WithUnitL(Nat, add)
withUnitRAdd ∈ WithUnitR(Nat, add)
addCommMonoid ∈ CommMonoid(Nat, add)
lists
 List ∈ (A ∈ Set) Set
 nil ∈ (ιA ∈ Set) List(A)
 cons ∈ (ιA ∈ Set; a ∈ A; l ∈ List(A)) List(A)
 map ∈ (ιA, ιB ∈ Set; f ∈ (A) B; l ∈ List(A)) List(B)
 append ∈ (ιA ∈ Set; List(A); List(A)) List(A)
 ListAll ∈ (ιA ∈ Set; P ∈ (A) Set; List(A)) Set
 la0 ∈ (ιA ∈ Set; ιP ∈ (A) Set; List(A)) Set
 la1 ∈ (ιA ∈ Set; ιP ∈ (A) Set; ιa ∈ A; ιl ∈ List(A); ιP(a); ListAll(P, l)) ListAll(P, cons(a, l))
 length ∈ (ιA ∈ Set; List(A)) Nat
 listIdLem1 ∈ (ιA ∈ Set; ιa ∈ A; ιx ∈ List(A); Id(nil(), cons(a, x))) Empty
 listIdLem2 ∈ (ιA ∈ Set; ιa ∈ A;
  ιa, ιb ∈ A;
  Id(cons(a, x), cons(b, y))) Id(a, b)
 listIdLem3 ∈ (ιA ∈ Set; ιa ∈ A;
  ιa, ιb ∈ A;
  Id(cons(a, x), cons(b, y))) Id(a, b)
 decIdList1 ∈ (ιA ∈ Set;
  d ∈ (a, b ∈ A) Plus(Id(a, b), Not(Id(a, b)));
  x, y ∈ List(A) Plus(Id(x, y), Not(Id(x, y))))
 decIdList ∈ (A ∈ Set; d ∈ DecR(Id())) DecR(Id())
 assocUnitLAppend1 ∈ (A ∈ Set;
  x, y, z ∈ List(A)) Id(append(append(x, y), z),
  append(x, append(y, z)))
 withUnitLAppend1 ∈ (A ∈ Set; x ∈ List(A)) Id(append(nil(), x), x)
 withUnitRAppend1 ∈ (A ∈ Set; x ∈ List(A)) Id(append(x, nil()), x)
assocAppend ∈ (A ∈ Set) Assoc(List(A), append())
 withUnitLAppend ∈ (A ∈ Set) WithUnitL(List(A), append())
 withUnitRAppend ∈ (A ∈ Set) WithUnitR(List(A), append())
 monoidAppend ∈ (A ∈ Set) Monoid(List(A), append())
 appendLength ∈ (A ∈ Set;
  x, y ∈ List(A)) Id(length(append(x, y)),
  add(length(x), length(y)))

vec
 Fin ∈ (Nat) Set
 0Fin ∈ (n ∈ Nat) Fin(s(n))
 sFin ∈ (ιn ∈ Nat; Fin(n)) Fin(s(n))
 finToNat ∈ (ιn ∈ Nat; Fin(n)) Nat
 emb ∈ (ιn ∈ Nat; Fin(n)) Fin(s(n))
 addFin ∈ (ιm, ιn ∈ Nat; Fin(m); Fin(n)) Fin(add(m, n))
 Vec ∈ (Set) Nat Set
 nilVec ∈ (A ∈ Set) Vec(A, 0)
 consVec ∈ (ιA ∈ Set; ιn ∈ Nat; ιA; Vec(A, n)) Vec(A, s(n))
 nth ∈ (ιA ∈ Set; ιn ∈ Nat; Vec(A, n); Fin(n)) A
 update ∈ (ιA ∈ Set; ιn ∈ Nat; Vec(A, n); Fin(n); A) Vec(A, n)
 insert ∈ (ιA ∈ Set; ιn ∈ Nat; Vec(A, n); Fin(n); A) Vec(A, s(n))
 delete ∈ (ιA ∈ Set; ιn ∈ Nat; Vec(A, s(n)); Fin(n)) Vec(A, n)
4.2 The Contributions Library

This library is not based on the Micro Library, because it grew up independently of it. The different contributions build on one another. These contributions include the monomorphic set theory as presented in [13] with all the elimination constants and also their reading as logical constants; arithmetic with natural numbers; some results about the algebraic structure of integers and some basic functions and properties on lists. The library is organized as follows:

The directory **Monomorphic.Set.Theory** contains the following theories: **Empty**, N1, **Bool**, Nat, Sigma, **Product**, Pi, **Function**, Union, Id, Id.properties.

The directory **Predicate.Logic** contains the theories: Absurdity, True, And, Or, Imply, Exists, Forall.

The directory **Natural.Numbers** contains the theories: Nat.properties, Lift, It, plus, mult, div, monus.

The directory **Integers** contains the theories: Z, plussubs, ldZ, zsprops, zadd, zaddinv, zminus.

The directory **Lists** contains the theories lists, list.props, perms, gensym.

5 Bugs and improvements to be made

The current version of ALF is still very much experimental. Users are encouraged to report bugs or make proposals for improvements by email to alf@cs.chalmers.se. 

The following is a list of known bugs or shortcomings of ALF which will hopefully deal with in forthcoming releases.

- Sometimes the saved scratch file is inconsistent because the definitions are saved in the wrong order. This can be repaired by editing the scratch file.

- When editing the type of a constructor one sometimes gets an error message 

  *print_in_order cannot happen.* The only way out is to start editing this constructor again (but saving the current state by copy and paste).

- There is a problem when copying definitions with hidden variables. In this case one should select **Show all arguments** first or change the layout of the constants in question.

- There are some layout problems with the postscript file. E.g. expressions may disappear because they are too much to the right.

- The window interface has a memory leak and uses up more and more memory after some time. Therefore it can be useful to save and restart ALF from time to time.

- When ALF is busy it sometimes does not repaint the window which may lead to strange looking windows. This should not affect the usability of the system.

---

7Local users should rather consult the newsgroup cth.cs.alf.
• Some operations (like sl Clear) are performed asynchronously which may have the effect
that other operations (like the updating of the type of the selected expression) are
blocked.

• Sometimes the choice menu right button may get very long, such that the lower parts
cannot be selected.

• ALF first loads a file into memory before parsing. This may lead to strange behaviour
when one attempts to load a very large file (like a core dump).

• One cannot select anything in the type of the current expression.

• The menu items Normal Form, Head Normal Form and Expand may appear twice
in the choice menu (right button). In this case the first occurrence corresponds to the
Replace menu and the second to the View menu.

If you believe that you have found a new bug it is useful if you try to narrow down the
problem as much as possible and produce a history of the actions. This can be done by
selecting Command and typing history bug. The file bug together with the theory and the
scratch area and a description of the problem should be send to the above address.

6 History of the system

This version of ALF has been implemented by Lena Magnusson and Johan Nordlander.
Lena has implemented the proof engine and Johan the user interface.

The basic ideas in this system comes from the first version of ALF which was designed
by Thierry Coquand and Bengt Nordström in 1991, Thierry implemented the first proof
engine and Lennart Augustsson implemented the user interface.

7 Acknowledgements

We want to thank Thierry Coquand, Lena Magnusson and Johan Nordlander for all their
efforts put into this project.

A Syntax

The following grammar defines the syntax of expressions (EXP), types (TYPE) and pat-
terns (PATTERN). The syntax which is constructed by the window interface is automatic-
ically correct but it is useful to know the syntax when using Edit As Text.

An identifier (ID) in ALF is a sequence of letters, digits and the underscore character
(_).
\[
\begin{align*}
\text{EXP} &::= \text{VAR} \\
&\quad| \text{CONST} \\
&\quad| [\text{VARLIST}] \text{EXP} \\
&\quad| \text{EXP} \langle \text{DECLIST} \rangle \text{EXP} \\
&\quad| ? \\
\text{VAR} &::= \text{ID} \\
\text{CONST} &::= \text{ID} \\
\text{VARLIST} &::= \text{VAR} | \text{VAR} , \text{VARLIST} \\
\text{DECLIST} &::= \text{DEC} | \text{DEC} ; \text{DECLIST} \\
\text{PATTERN} &::= \text{VAR} \\
&\quad| \text{ID} \langle \text{PATTERNLIST} \rangle \\
&\quad| - \\
\end{align*}
\]

Note: The syntax of case-expressions is not included since it is subject to change. In the moment case-expressions are only possible on the top-level; i.e. not inside a \(\lambda\)-abstraction.

References


