A short description of Another Logical Framework

Lennart Augustsson, Thierry Coquand, Bengt Nordström

Department of Computer Science
University of Göteborg/Chalmers
S-412 96 Göteborg
Sweden

Abstract

This note is an incomplete description of an incomplete implementation of another logical framework.

Major characteristics

ALF is a system for editing proofs and theories. It is based on a combination of a general type system (GTS) and Martin-Löf's logical framework. It is possible to add equations when defining a theory. This possibility is implemented in the present version (it is thus possible to define the operations of Martin-Löf's monomorphic set theory inside ALF).

The most important characteristic is that ALF takes seriously the idea of "proofs as objects", in the sense that the interaction consists in building a proof-term, which is actually displayed on the screen. The major proof editing operations is to refine a placeholder (standing for an incomplete subproof) to a partial proof and to delete a partial proof, i.e. replace it with a placeholder.

The (partial) proof-term is presented together with constraints that are typing constraints (context and types of the subgoals), and equational constraints (context and pair of terms of the same type in this context). The invariant is that if all the constraints are satisfied then the (partial) term is a correct proof of the goal.

There is no direct unification algorithm. Instead, the system uses only the first part of the higher-order unification algorithm, that is the simplification of the equational constraints. These equational constraints are then displayed to the user in a simplified form. The user then control the remaining (and non deterministic) part of the higher-order unification algorithm, that is the choice of projections or imitation. So far, only a very rudimentary approximation of this is allowed, but it is enough to deal with most easy induction proofs.

The system has a tiny beginning of "structure" for theories, a theory being recursively a list of declaration of constants (primitive or defined), and theories.

The logical framework

So far ALF represents the following GTS. The sorts are Prop and Type_k, $k = 0, 1, \ldots$. The typing axioms are $\mathsf{Prop} : \mathsf{Type}_k$, and $\mathsf{Type}_i : \mathsf{Type}_j$ for $i < j$, and the typing conditions are $(s, \mathsf{Prop}, \mathsf{Prop})$

[1]This research has been done within the ESPRIT Basic Research Action "Logical Frameworks".It has been paid by STU and Chalmers.

and $(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_k)$ for $i, j \leq k$. We recall that to have the condition (s_1, s_2, s_3) means that we can form the following product

$$\frac{A : s_1 \quad B[x] : s_2\ [x : A]}{\Pi(A, B) : s_3}$$

It is intended that (following R. Pollack's "operational" presentation of GTS) , ALF will in a near future allow the user to declare any kind of GTS.

A GTS that we want to have is the one corresponding to Martin-Löf's logical framework. In this case, the sorts are Set and Type, the typing axiom is $\mathsf{Set} : \mathsf{Type}$ and the typing conditions are $(\mathsf{Type}, \mathsf{Type}, \mathsf{Type})$ and $(\mathsf{Set}, \mathsf{Set}, \mathsf{Type})$.

It is also intended to compare the "GTS approach" with a direct representation of Martin-Löf's logical framework. One important difference, for instance, is the fact that, in the latter approach, every constant has a fixed arity (and this is not the case in general in a GTS).

The theory editor

A theory is presented by a list of typings and definitions of constants. We distinguish between primitive and defined constants. A *primitive constant* has only a type, it doesn't have a definition. It gets its meaning in other ways (outside the theory). Such a constant is also called a constructor, since it computes to itself. Examples of primitive constants are N, succ and $\mathsf{0}$, where $\mathsf{N} \in \mathsf{Set}$, $\mathsf{succ} \in \mathsf{N} \rightarrow \mathsf{N}$ and $\mathsf{0} \in \mathsf{N}$.

A defined constant has a type and a definition:

$$c \in A \equiv a$$

The definiendum c computes in one step to its definiens a. A defined constant can either be explicitly or implicitly defined. An *explicitly defined constant* (macro) is just an abbreviation of its definiens (which has to be a welltyped expression). An *implicitly defined constant* is in general defined in terms of itself. Whether this is meaningful or not can only be checked outside the theory. The recursion operator over natural numbers is an example of an implicitly defined constant. When we read the constant as a name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and finally, an explicitly defined constant is a defined rule.

The system used is monomorphic. However, a constant can be declared with hidden parameters. This is only a comment for the pretty printing. In the reverse direction, we use place holders to input the hidden parameters. In this way, the user gets a polymorphic view of the system.

Groups

A problem with presenting the theory as one list of identifiens is that the list grows easily to an unmanageable size. Some structure is obtained by using the *let*-construct which makes it possible to introduce local definitions (or local lemmas). But this is not enough. We seem to need some kind of Σ-type in the framework to get more structure in the list of definitions. In the current implementation, we can put together some identifiens in a *group*, which is mainly a presentational device (it is possible to hide or show the identifiers in a group by clicking on the name of the group). A theory can be saved in a file and a group may also be stored in separate files. In our description of Martin-Löf's constructive set theory we have a group for each set-forming operation with its formation, introduction- and elimination-rules.

Commands of the theory editor

There are commands to extend a theory with new constants and groups. It is possible to include another theory (as a group) stored in a file. It is possible to move a constant between groups. It is also possible to invoke a syntax editor which makes it possible to change the concrete syntax of expressions. In the current implementation it is possible to declare a constant to be an infix, prefix or postfix operator or to be written as a quantifier.

The most important command, however is the one which invokes the proof editor which is used to edit proofs.

The proof editor

The proof editor can be described as a machine with the following registers:

- a theory (i.e. a list of definitions and typings),
- a (partial) proof term with a cursor pointing to the selection (a subproof),
- the type of the selection,
- the goal to be proven.

The following can be computed from the state of the machine:

- the type of each subterm,
- the local context, i.e. the typing of all variable bound at the selection,
- the constraints which is to hold for the place holders in order for the machine to represent a correct proof.

The screen will display all these.

The basic editing command is the one which replaces a place holder with a proof term. An example of this command is the one which applies a rule. The user then double click on one of the constants in the theory or on a variable in the local context. The system will then replace the selection (which has to be a place holder) with the term

$$c(?, ?, ..., ?)$$

where c is the name of the rule and $?, ?, ..., ?$ are enough new place holders to match the arity of the current subgoal.

The inverse command (delete) takes a selected subterm and replaces it with a new placeholder. In this way it is possible to undo arbitrary parts of the proof, and not only the most recent proof step.

There are also commands which massages the type of the current subgoal in different ways, for instance replaces a subterm of the subgoal with its definiens or its definiendum.

Proof Experiments

The formalization of Martin-Löf's monomorphic set theory is straightforward, it took only a few hours to do this.

The major experiment was done by Nora Szasz, who proved formally that Ackermann's function is not primitive recursive. The proof consists of about 200 theorems, of which around 50 are part of the proof of the growth of Ackermann's function. The rest are lemmas of properties of primitive recursive functions and natural numbers.

Acknowledgements

We want to thank the programming logic group at the department for many discussions on this topic. In particular we want to thank Lena Magnusson for her insightful comments.

The work has been done within the ESPRIT basic research action on design and implementation of logical frameworks. It has been paid by STU.