Some tools for computer-assisted theorem proving in Martin-Löf type theory

Marcin Benke
e-mail: `marcin@cs.chalmers.se`

Department of Computing Science
Chalmers University of Technology
412 96 Göteborg, Sweden

Abstract. We propose some tools facilitating interactive proof and program development in the proof editor Alfa based on Martin-Löf Type Theory, in particular a tool for equality reasoning supported by tools for deriving equality (and proofs or its properties) for inductive datatypes as well as automated proof-search.

1 Introduction

1.1 Context: proof editors and type theory

Alfa [HR00] is a graphical, syntax-directed editor for the proof system Agda [Coq98], is an implementation of *s*tructured type theory (STT) [CC99], which is based on Martin-Löf's type theory [ML84].

Like its predecessors in the ALF family of proof editors [Mag94], Alfa allows one to, interactively and incrementally, define theories (axioms and inference rules), formulate theorems and construct proofs of the theorems. All steps in the proof construction are immediately checked by the system and no erroneous proofs can be constructed.

Alfa is a term-based proof editor. This means that the proof is presented and recorded as a term, rather than as a tactic expression as in tactic-based proof editors such as Coq [BBC$^+$97,pro99]. Of course this does not preclude usage of tactics.

Alternatively, you can view Alfa as a syntax-directed editor for a small purely functional programming language with a type system that provides dependent types, thus allowing specification and verification of program properties within its type system. In fact, the language is very similar to the functional language Cayenne[Aug98] by Lennart Augustsson (which in turn is based on Haskell).

As such, Alfa supports algebraic datatypes, pattern matching and general recursive definitions. To preserve logical consistency, all proofs are subject to termination check as well as typechecking.

Since checking termination is undecidable, the checker approximates, keeping on the safe side, hence some definitions may be unduly rejected. It is however more liberal than for example restricting to primitive or well-founded recursion [Wah00].

2 Equational reasoning

Equality proofs are important for verification of functional programs. Unfortunately, current type theory style equational proofs are tedious and not easily readable. We plan to develop methods and tools that would improve this situation, basing partially on preliminary results on deriving equality for algebraic types [Ben00] (summarized in Section 4).

An interesting starting point is an unpublished proposal by Lennart Augustsson [Aug99] describing a syntactical extension for equality proofs in Cayenne. We believe that it can be made much more convenient by using interactive mechanisms of *Alfa* and automated proof-search such as described in [Ben01] (which we briefly describe in Section 3).

Let us start with a simple example: assume we have defined natural numbers, equality on them, addition, and a couple of simple lemmas about it. We now want to prove that addition is commutative[1]

```
addComm (m::Nat)(n::Nat) :: eq (add m n) (add n m)
  = case n of {
      (Zero) ->
        trans (add m Zero) m (add Zero m) (zeroNeutral m)
          (sym (add Zero m) m (zeroNeutral_left m));
      (Succ n') ->
        trans (add m (Succ n')) (Succ (add m n')) (add (Succ n') m)
          (refl (add m n'))
          (trans (Succ (add m n')) (Succ (add n' m)) (add (Succ n') m)
             (congr (add m n') (add n' m) (addComm m n'))
             (trans (Succ (add n' m)) (add one (add n' m)) (add (Succ n') m)
                (sym (add one (add n' m)) (Succ (add n' m)) (thm1 (add n' m)))
                (trans (add one (add n' m)) (add (add one n') m)
                   (add (Succ n') m)
                   (sym (add (add one n') m) (add one (add n' m))
                   (addAssoc one n' m))
                   (addSameLeft m (add one n') (Succ n') (thm1 n')))));}
```

to quote Augustsson, “this is a total mess, and penetrating its meaning is only recommended for masochists”. What we actually would like to write is something like this:

[1] Of course such theorems about natural numbers can be proven by other means, eg. Coq's `Omega` tactic, but this is not the point; our tool applies equally well to other types.

```
addComm (m::Nat)(n::Nat) ::  (m + n == n + m)
  = case n of {
      (Zero) -> EqChain { m + Zero == m == Zero + m }
      (Succ n') -> EqChain {
       m + Succ n' == Succ(m + n') == Succ(n' + m) ==  1 + (n' + m)
                     == (1 + n') + m == (Succ n') + m == n + m
      }
   }
```

(or even something a bit simpler) while the first, "ugly" form is preserved "behind the scenes" and fed to the typechecker.

Trivial beautifications like infix operators and their presentation using symbol font have been present in Alfa for some time. Using Augustsson's notation brings us closer to the target, e.g. for the *Zero* case we get

```
  m + Zero ={ DEF }= m ={ zeroNeutral m }= Zero + m
```

But there still is a fair amount of noise, especially in more complicated cases. Moreover, this notation, while quite convenient for batch systems as well as making presentation substantially cleaner, is not a big improvement with an advanced interactive system like *Alfa*.

We propose a tool aimed specifically at interactive construction of proofs by equational reasoning (chains of equalities). Such construction using our tool proceeds as follows:

1. User states his goal, eg
$$?_0 \in (m + \mathbf{Zero} == \mathbf{Zero} + m)$$
and chooses the *EqChain* tool
2. System displays an initial (uncomplete) chain, eg
$$m + \mathbf{Zero} == ?_1 == \mathbf{Zero} + m$$
3. User proposes subsequent elements of the chain. At each step, system checks whether it can find the proof of equality between neighbouring elements. If not, the user has now the following options:
   – insert some intermediate elements into the chain,
   – state some additional lemma enabling automatic proof,
   – supply the proof manually.
4. At every step, the system also checks whether the chain is complete (i.e. whether it can find the proof of the last step and remove the question mark. The user can also "close" the chain manually by supplying the missing proof.

3 Automated proof search

3.1 Assumptions

– We consider interactive proof developments in a system based on Monomorphic Martin-Löf Type Theory.

– Proof development proceeds interactively in small steps (definitions/lemmas)[2].
– User states a lemma and (optionally) hints as to which previous lemmas he thinks might be useful in proving the current one.
– System tries to prove the lemma using hints.

The task of constructing an automated tool within this framework faces us with several challenges:

Given the interactive nature of the system, response time is an important consideration. The user should expect an answer within reasonable time.

Since Type Theory is an intuitionistic, higher-order logic, applicability of "classical" theorem proving methods is rather limited, even though possible for some fragments of the theory [TS98].

Every term may have many equivalent types; some of them give better clues as to possible proofs than others. Moreover the reduction relation is more complicated than in pure lambda calculus since one has to expand definitions and **case** expressions too.

3.2 Strategies

Given the assumptions above, a typical situation would look like

$$env \vdash f\ (\boldsymbol{x} \in \boldsymbol{T}) = ? \in G$$

where *env* contains existing definitions/lemmas (which will be further referred to as *globals*), f is the current definition and $\boldsymbol{x}$ represents a lists of *parameters* along with their types. Our aim is to find a proof term of type G (goal) to replace "?" (hole). Moreover the term must be such that the definition would pass the termination check.

The strategy is guided by a set of directives that ensure termination as well as — given the interactive nature of the environment — reasonable response time. For this reason it may fail to produce a proof, especially when no simple proof exists. The user should then either add one or more auxiliary lemmas (which of course may possibly be proven automatically) or reconsider her theorem.

4 Deriving equality

Algebraic datatypes are one of central features in functional programming languages like Haskell and Cayenne as well as in proof-editors based on Martin-Löf type theory such as Alfa and Agda. Usually, when defining a new datatype, one stipulates some standard properties. However, spelling them out in full is often quite tedious. This is even more visible in proof editors than in Haskell, as we not only need the functions but also their properties properties and proof objects. Haskell has a mechanism that can automatically derive instances of standard

[2] In Martin-Löf Type Theory there is no clear distinction between definitions and theorems

classes, as in

```
data List a = Nil | Cons a (List a) deriving Eq
```

The *Derive* plugin provides a similar facility for Alfa. Given a datatype, it derives equality definition, along with a proof that it is an equivalence relation. For first order types, it also generates proofs that the derived equality is substitutive and decidable.

References

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98).* ACM Press, September 1998.

[Aug99] Lennart Augustsson. Equality proofs in Cayenne. Available at `http://www.cs.chalmers.se/~augustss/cayenne/eqproof.ps`, 1999.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[Ben00] Marcin Benke. Automatic deriving of properties of algebraic datatypes in Martin-Löf type theory. Presentation at Annual ESPRIT BRA TYPES Meeting, Durham. Submitted for publication, December 2000.

[Ben01] Marcin Benke. Strategies for interactive proof and program development in Martin-Löf type theory. To appear in proceedings of 4th International Workshop on Strategies in Automated Deduction, June 2001.

[CC99] C. Coquand and T. Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages*, Paris, France, Sep 1999.

[Coq98] Catharina Coquand. The AGDA Proof System Homepage. `http://www.cs.chalmers.se/~catarina/agda/`, 1998.

[HR00] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer, 2000.

[Mag94] L. Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 1994.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[pro99] The LogiCal project. Coq Homepage. `http://coq.inria.fr/`, 1999.

[TS98] Tannel Tammet and Jan Smith. Optimized encodings of fragments of type theory in first-order logic. *JLC: Journal of Logic and Computation*, 8, 1998.

[Wah00] David Wahlstedt. Detecting termination using size-change in parameter values. Masters Thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 2000.