Strategies for interactive proof and program development in Martin-Löf Type Theory (extended abstract)

Marcin Benke

Department of Computing Science
Chalmers University of Technology
413 96 Göteborg, Sweden

Abstract. We propose some strategies for automatic tools facilitating interactive proof and program development in the proof editor Alfa based on Martin-Löf Type Theory.

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[1] , 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].

1.2 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"[3] 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.

2 Strategies

Given the assumptions above, a typical situation would look like

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

[1] By an algebraic type we mean a type being a sum of products, defined by listing its constructors, eg

```
data Tree = Empty | Leaf Int | Node Tree Tree
```

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

[3] By "classical", we mean methods that use *tertium non datur*, for example to prove a theorem by refuting its negation.

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.

2.1 Directives

The main directives of our strategy are:[4]

(DG) Use of globals: since most proof developments make use of numerous library modules, this leads to a very large search space; therefore usage of globals is limited to
 (DG1) globals specified in hints,
 (DG2) constructors ,
 (DG3) cheap globals (i.e. ones that do not increase the search space substantially)
 (DG4) "local" globals (i.e. ones defined in the current module)

(DR) Recursive calls: discouraged, especially where they might lead to non-termination (we shall call this subdirective DR1) (with a notable exception of the "recursive vs recursive" strategy described in (DA) below)

(DP) Use of parameters: encouraged; solutions using **all** parameters are preferred. Notice that within the scope of **case**, parameter lists are updated accordingly.

(DA) Algebraic types: if the initial proof search fails to produce satisfactory result and one or more parameters is of an algebraic type, use pattern-matching; if a type is recursive, try "recursive vs recursive" strategy: using recursive calls on the recursive constructor arguments (this subdirective is closely related to (DR) hence we call it DR2).

(DO) Ordering candidates: during the proof search, candidates for resolving particular subgoals are ordered according to anticipated "cost" of their use. However, we propose to use type ordering rather than an integer-based ranking.

It is worth noting that the directive (DP) may have negative impact on the efficiency of our strategy, so disabling it is desirable in many cases. However, it does have some merits, one of which is illustrated in the example in the following section. It should also be noted that it has global,

[4] The parenthesised directive abbreviations are used as a shorthand in the example in Section 3

rather than local character (but may be also implemented in a compositional way).

3 Motivating example: *map*

As an illustration of how the directives described above work, it will be shown how they allow deducing the right-hand side of a function definition from its type and left hand side. Even though the example seems to belong more to functional programming rather than proof development, but it is short and illustrates almost all directives. Besides, one should remember that in type theory all proofs are programs and are subject to similar phenomena. The choice of example is also motivated by our interest in program verification and deriving programs from specifications.

Assume we have

$$[A] = \textbf{data}\ [\,]\ |\ : (x \in A, xs \in [A])$$

and we want the system to fill the hole in

$$map\ (f \in A \to B)(xs \in [A]) \in [B] = ?_0$$

pursuant to the directives stated above.

Initial proof search yields two type-correct results:

- $[\,]$ — unsatisfactory due to (DP);
- $map\ f\ xs$ — rejected due to (DR1): fails termination check.[5]

Since the proof search failed to produce satisfactory results and the parameter xs is of an algebraic type, the system proceeds to case analysis (DA), replacing $?_0$ by

$$\begin{array}{l} \textbf{case}\ xs\ \textbf{of} \\ \quad [\,] \to ?_1 \\ \quad (x : xs') \to ?_2 \end{array}$$

Proof search for $?_1$ yields two candidates:

- $[\,]$
- $map\ f\ [\,]$ — not considered; we are in a "non-recursive" branch.

Proof search for $?_1$ yields five candidates (we are in a "recursive" branch, so use of recursion is allowed):

[5] In fact no candidates using *map* would be even considered; recursive candidates are considered only in specific situations, such as specified in (DR2)

– [ ]
– *map f xs'*
– $map\ f\ [\,]$
– $(f\ x) : [\,]$
– $(f\ x) : (map\ f\ xs')$

All considered candidates pass both type- and termination-check. However the only one satisfying (DP) is

$$\begin{array}{l} \textbf{case}\ xs\ \textbf{of} \\ \qquad [\,] \rightarrow [\,] \\ \qquad (x : xs') \;\rightarrow\; (f\ x) : (map\ f\ xs') \end{array}$$

which is the "usual" definition of *map*!

Similar results are obtained for other functions on lists, e.g. list concatenation. Interestingly enough, when provided with a hint to use *foldr* (provided an usual definition of foldr is visible), our strategy yields an alternative definition of list *concat* using *foldr* rather than pattern matching:

$$\begin{array}{l} concat\ ::\ [[a]] \;\rightarrow\; [a] \\ concat\ xss \;=\; foldr\ (+\!\!+)\ [\,]\ xss \end{array}$$

As a side note, let us remark that our tool also succeeded to generate a similar definition of ordinary list concatenation:

$$\begin{array}{l} (+\!\!+)\ ::\ [a] \;\rightarrow\; [a] \;\rightarrow\; [a] \\ xs\ +\!\!+\ ys \;=\; foldr\ (:)\ ys\ xs \end{array}$$

But we have to admit that This is however due to a bit of luck, since (without additional restrictions) there are two solutions satisfying our criteria. To distinguish between them, one should provide a more precise specification of list concatenation.

4 Type orderings

For the purposes of ordering candidates we propose using quasi-orders on types. One interesting ordering we are currently experimenting with is based on the following system:

$$0 \leq A$$

$$A \leq 1$$

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

$$A_i \leq A_1 + A_2 \text{ for } i = 1, 2$$

$$A_1 \times A_2 \leq A_i \text{ for } i = 1, 2$$

where 0 represents empty types, 1 represents one-element types and A, B (possibly with indices) represent arbitrary types.

The ordering is intended to represent how easy (or difficult) it is to find an element of given type, with "greater" meaning "greater ranking" i.e. "easier". The intuitions behind the rules are thus:

- It is difficult to provide an element of an empty type.[6]
- It is trivial to provide an element of a unit type.
- The stronger the assumptions and the weaker the thesis, the easier it is to prove the implication. On the extreme we have *ex falso quod libet*.
- Constructing an element of a sum type amounts to constructing an element of any of its components.
- Constructing an element of a product type amounts to constructing elements of all its components.

5 Implementation

A tool based on the strategies described in this paper has been implemented as a plugin for the Alfa proof editor. The plugin (as well as Alfa itself) is written in Haskell. Initial experiences with using it seem quite promising, however a *corpus* of test cases (see below) is needed for a performance evaluation.

6 Future work

- Basing on user feedback build a *corpus* of test cases and test the strategies more extensively.
- Compare relative merits of various type orderings against each other and against integer scores.
- Improve efficiency of the implementation.
- Extend the range of tactics used; better tactics for equational reasoning seem to be a priority.

[6] In most cases it is just impossible, but we must relativize with regard to context.

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].

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.

[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.

[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.