A Tool for Automated Theorem Proving in Agda*

Fredrik Lindblad and Marcin Benke

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

Abstract. We present a tool for automated theorem proving in Agda, an implementation of Martin-Löf's intuitionistic type theory. The tool is intended to facilitate interactive proving by relieving the user from filling in simple but tedious parts of a proof. The proof search is conducted directly in type theory and produces proof terms. Any proof term is verified by the Agda type-checker, which ensures soundness of the tool. Some effort has been spent on trying to produce human readable results, which allows the user to examine the generated proofs. We have tested the tool on examples mainly in the area of (functional) program verification. Most examples we have considered contain induction, and some contain generalisation. The contribution of this work outside the Agda community is to extend the experience of automated proof for intuitionistic type theory.

1 Introduction

Automated proving in first-order logic is well explored and developed. Systems based on higher-order logic have in general limited automation. This is in particular true for proof-assistants based on intuitionistic type theory. There is strong motivation for working with these formalisms and the tools based on them have a large user community. As a result, a lot of interactive proving is carried out to construct proofs or parts of proofs which could conceivably be solved automatically.

We have developed a tool for automated proving in the Agda system [3]. It is not a complete proof search algorithm. The aim is to automate the construction of the parts of a proof which are more or less straightforward. Often such parts can be tedious to fill in by hand, however significant time could be saved, allowing the user to spend her effort on the key parts of the proof.

Agda is an implementation of Martin-Löf's intuitionistic type theory [10], which, following the paradigm of propositions-as-types and proof-as-objects, can be used as a logical framework for higher-order constructive logic. Agda is a type-checker and an assistant for constructing proof terms interactively. The assistant is not based on tactics. Instead, the user sees a partially completed proof term during the process. The incomplete parts of the term are represented

* Research supported by the project *Cover* of the Swedish Foundation of Strategic Research (SSF).

J.-C. Filliâtre et al. (Eds.): TYPES 2004, LNCS 3839, pp. 154–169, 2006.

by place-holders, called *meta variables*. Each step consists of refining one of the remaining meta variables, which instantiates it to a term that may contain new meta variables. Refining one meta variable may also instantiate others, as a result of unification.

The tool does not rely on an external solver. During the proof search, the problem and the partial solution are represented as Agda terms. The tool is integrated with the Agda proof assistant, which runs under *emacs*. When standing on a meta variable, the user can invoke the tool, which either inserts a valid proof term or reports failure of finding a solution. Failure is reported if the search space is exhausted or if a certain amount of steps has been executed without finding a solution. There is also a stand-alone version of the tool, intended for development and debugging, where a proof search can be monitored step by step.

Before inserting a proof term, it is always verified by the Agda type-checker. Therefore, the importance of soundness is limited, and we will not further discuss this issue. Still, we believe that the basic steps of the proof search are consistent with the Agda typing rules and moreover that the algorithm is sound.

Since the area of application is within an interactive proof assistant, producing human readable proof terms is important and has received some attention in our work.

The tool handles hidden arguments, which is a new feature of Agda. It has however so far no support for inductive families [7], except for when they are used for representing equalities.

The fundamental restriction of the tool is that it does not do higher order unification. Instead, a decidable extension of first-order unification is used. In e.g. Dowek's algorithm for term synthesis in pure type systems [5], applying an elimination rule produces a constraint, which is successively solved by higher-order unification. Our tool instead only applies an elimination rule if the unification procedure returns a unifier.

In Agda, termination is not verified by the type-checker. There is a separate termination-check. In our tool, inductive proofs are restricted to structural recursion. Determining termination is in general undecidable and defining advanced criteria is an issue in itself. An alternative approach would be to ignore termination and let an external verification accept or reject a proof term. However, the tool is currently designed to produce only one solution, which makes it inappropriate for that approach. Although the flexibility of induction schemes is limited, the tool can do nested induction on several variables and also nested case split on one variable.

Elimination rules are generally only applied when there is a suitable variable in the context to apply it on. The exception to this is that the tool identifies a couple of cases of generalisation. In some cases this leads to a necessary strengthening of the induction hypothesis. The generalisation mechanism is however restricted to what can be syntactically deduced from the current goal type, like replacing a repeated subexpression by a fresh variable. There is also no synthesis of new hypotheses, i.e. the tool is unable to do lemma speculation.

In addition, there are a number of other restrictions, e.g. that new data types and recursive functions are not synthesised. This means that, when searching for a proof of a proposition which is existentially quantified over *Set* or a function space, a new data type or a new recursive function will not be synthesised.

Section 2 contains a small survey of related work. Section 3 describes the tool and contains a few subsections, which are devoted to special features. In section 4 we present a few examples and discuss the limitations of the tool. Section 5 gives conclusions and ideas for how the tool could be improved in the future.

2 Related Work

Although the type inhabitation problem is undecidable for a system like Agda, it is semi-decidable simply by term enumeration (plus decidability of type-checking). A complete proof synthesis algorithm for the pure type systems has been presented by Dowek [5]. Cornes has extended this to a complete algorithm for the Calculus of Inductive Constructions [4]. Although these algorithms are of theoretical interest, complete algorithms so far seem too time-consuming to be of practical use. In her thesis, Cornes elaborates on various enhancements, but, to our knowledge, there is no implementation available.

We now turn to a quick survey of related implementations, beginning with a piece of related research in the context of Agda, and then working our way outwards in wider circles. Smith and Tammet [12] have investigated using a FOL-solver to mechanically find proofs in *Alf* [9], the predecessor of Agda. The goal type together with the typing rules of Alf are encoded in first-order logic and a solver, *Gandalf*, is invoked. If a solution is encountered, a proof term is constructed using the information produced by Gandalf. The authors managed to generate some inductive proofs, but the approach seems rather inefficient.

The proof-assistant *Coq* is based on a language closely related to that of Agda. Coq has many sophisticated tactics, but the *auto* tactic, which is the nearest counterpart of our tool, does not produce any inductive proofs.

Also related to Agda is the *Logical Framework*, implemented in the system *Twelf*. Schürmann and Pfenning [11] have developed a proof search tool and supporting theory for a metalogic in *Twelf*, an implementation of the logical framework LF.

Andrews has successfully explored the area of mechanical proofs in classical type theory [2]. His work has resulted in *TPS*, a fully automatic proof system based on higher-order unification.

ACL2, *PVS* and *Isabelle* are other major proof assistants. ACL2 and PVS do have automation for induction, but none of the systems produces proof objects.

3 Tool Description

Agda has dependently-typed records and algebraic data types. The algebraic data types can be recursively defined. Function arguments are analyzed by

case-expressions rather than pattern matching. Functions can be recursively defined freely using self-reference. There is no fixpoint construction. Type-checking does not include termination check. Although there is a separate termination checker, the restrictions are not clearly defined in the semantics. Hence, a tool for automated proving must either ignore termination issues or define its own criteria for this. With a proof search algorithm capable of producing multiple solutions, the first approach could be used. For each solution an independent termination check is consulted. If it rejects the proof term, the search is continued. Our tool is however designed to come up with only one solution, so it adheres to the second approach. Proof terms are currently restricted to structural recursion.

Agda is monomorphic but polymorphism is in recent versions simulated by argument hiding. Properties and proof terms below are presented with some type arguments omitted. This is done to improve readability, but the hiding mechanism is not further discussed.

Just like the Agda proof assistant, the tool uses place holders to denote incomplete parts of a proof. Place holders are called meta variables and are denoted by '?'. They can be seen as existentially quantified variables.

The most significant characteristics of the tool are the following:

- Unification is first-order and is not an interleaved part of the proof search. The search state does not have constraints. Unification is decided immediately when applying elimination is considered.
- The order in which the meta variables are refined is dictated by depth-first traversal. After refining a meta variable, its subproofs are recursively addressed one by one.
- Meta variables are classified as either *parameter meta variables* or *proof meta variables*. Parameter meta variables are those which appear in the type of some other meta variable, whereas the rest are proof meta variables. The parameters are the term-like meta variables. Only proof meta variables are subject to resolution. Parameter meta variables are supposed to be instantiated when unifying one of the types where they appear. In [5] Dowek pointed out that variables should be instantiated from right to left. The parameter/proof distinction is an alternative to this rule and it postpones the instantiation of term-like variables in a more flexible way. The distinction is also made for local variables, although not that explicitly.
- A meta variable is refined by an elimination only when there is a suitable object in the context. This means that for each meta variable, all possible ways to eliminate any object in scope are considered. It can be any combination of function application, record projection and also case split for algebraic data types. The fact that this applies to disjoint unions makes it necessary to tag solutions with *conditions* for which it is valid. As an example, if we have $[h : A + B] \vdash ? : A$, then a solution is $[h \to \texttt{inl}\ a] \vdash a$, which means a is a valid term provided that h is of the form $\texttt{inl}\ a$. The idea is that conditional solutions at some point should pair off to form an unconditional proof of the full problem.

The program consists of an algorithm which explores the search space and the implementations of a set of refinement rules. The proof search algorithm is presented in subsection 3.1. The refinement rules define the atomic steps of refining a problem and how to construct the corresponding proof term. The implementations of the refinement rules will be referred to as *tactics*. We will not present a formal description of the rules, since they are closely related to the typing rules of Agda. There are refinement rules for constructing λ-abstractions, record objects and algebraic data type objects. There is a rule for elimination, which combines several eliminations of different kind in the same refinement, as described above. There is also a special elimination rule for equalities, which uses transitivity, symmetry and substitutivity in a controlled manner. Then there is one rule for case analysis and induction and one for generalisation. They are presented by example in subsections 3.2 and 3.4 respectively. Finally, there is a rule for case on expression, which combines case analysis and generalisation. It is presented in section 3.3.

In the tactics which perform elimination and at some other places in the search algorithm, first-order unification is used to compare types in the presence of meta variables. Unification is always performed on normalised terms. The tool uses an extension of normal first-order unification. This enables it to deal with more problems without resorting to higher-order unification. The extension is still decidable. However, while a first-order unification problem has either one or no unifier, a problem of the extended unification can have any number of unifiers. The extension is presented in subsection 3.5. When doing first-order unification in the presence of binders, attention must be paid to scope. We have chosen to solve this by having globally unique identifiers for variables. Whenever a meta variable is about to be instantiated, its context is checked against the free variables of the term. If not all variables are in scope, the unifier is rejected.

When a proof term has been found, the tool does a few things to make it more readable. This includes using short and relevant names for local variables and removing redundant locally defined recursive functions.

3.1 Proof Search Algorithm

The search space is explored using iterated deepening. This is necessary since a problem may in general be refined to infinite depth. It is also desirable since less complex solutions are encountered first.

We will describe the proof search algorithm by presenting a pseudo program for the main recursion. The style is a mixture of functional and imperative programming, but we hope that the meaning of the program is clear. It refers to a few subfunctions which are only described informally. To make the presentation cleaner, unification is assumed to produce only one unifier, i.e. it does not incorporate the extension described in subsection 3.5.

First we define a few types which describe the basic objects of the algorithm. The elementary entities are denoted by single letters. The same letters are used both to denote the types and the corresponding objects, hopefully without

confusion. The general form of a problem of finding an inhabitant of type T in the variable context Γ is the following:

$$\Delta_{par}, \sigma, \Gamma, \rho \vdash T \quad (\texttt{Prb})$$

The corresponding type is called `Prb`. Here, $\Delta_{par} = [\Gamma_i \vdash ?_i : T_i]$ is the collection of parameter meta variables. For each meta variable, its context and type is given. The next component, $\sigma = [?_i := M_i]$, is the set of current parameter meta variable instantiations, which should be taken into account when reducing terms. After the context of the current problem, Γ, we have $\rho = [M_i \to \mathtt{c}_i\ y_{i,1}\ \cdots\ y_{i,n_i}]$. This is the sequence of conditions which have emerged so far. The conditions for proof variables should be taken into account in order to avoid clashes when eliminating disjoint unions. The conditions for parameter variables should be respected when normalising types.

A solution, `Sol`, to a problem has the following form:

$$\sigma', \rho' \vdash M \quad (\texttt{Sol})$$

The term M inhabits the target type provided that the meta variable instantiations and conditions of the problem are extended by σ' and ρ'.

We will also use the notion of *refinement*, `Ref`, which specifies how a problem can be refined to a new set of problems:

$$\Delta'_{par}, \Delta_{prf}, \sigma', \rho' \vdash M \quad (\texttt{Ref})$$

Just as for a solution, the proof term, M, is an inhabitant assuming the extra instantiations and conditions, σ' and ρ'. In general, M contains new meta variables. The new meta variables are divided into parameters, Δ'_{par}, and proofs, $\Delta_{prf} = [\Gamma_i, \rho_i \vdash ?_i : T_i]$, according to the classification described above. For proof meta variables the information supplied is different from that of parameter meta variables. Instead of the full context, only the extra local variables are given. Moreover, not only the context and type are given but also a set of extra conditions which should be enforced in the corresponding branch of the proof. This is needed since parameter variables are treated differently from proof variables. The distinction is the same as for meta variables – parameters are variables which appear in some type. While proof variables are eliminated on demand, as described above, case splits for parameter variables must precede the proof of its branches as a separate refinement.

Finally, we need to talk about collections of problems, `PrbColl`, and collections of solutions, `SolColl`.

$$\begin{aligned} \Delta_{par}, \sigma, \Gamma, \rho \vdash \Delta_{prf} &\quad (\texttt{PrbColl}) \\ \sigma', \rho' \vdash \sigma^* &\quad (\texttt{SolColl}) \end{aligned}$$

A problem collection is a set of common instantiations and a set of common conditions followed by a list of proof meta variables. A corresponding solution collection contains the extra sets of common instantiations and conditions as well as a set of instantiations, σ^*, which gives a term for every proof meta variable in the problem collection.

The following functions describe the proof search algorithm:

```
search : Prb → [Sol]
search (Δ_par, σ, Γ, ρ ⊢ T) =
  refs := createRefs (Δ_par, σ, Γ, ρ ⊢ T)
  sols := []
  for each (Δ'_par, Δ_prf, σ'_1, ρ'_1 ⊢ M) in refs
    prbcoll := ((Δ_par ++ Δ'_par), (σ ++ σ'_1), (ρ ++ ρ'_1) ⊢ Δ_prf)
    solcolls := searchColl prbcoll
    for each (σ'_2, ρ'_2 ⊢ σ*) in solcolls
      case (compose ((σ'_1 ++ σ'_2), (ρ'_1 ++ ρ'_2), M, σ*)) of
        none → sols := sols
        some sol → sols := addSol (sol, sols)
      end case
    end for
  end for
  return sols

searchColl : PrbColl → [SolColl]
searchColl (Δ_par, σ, ρ ⊢ []) = ([], [] ⊢ [])
searchColl (Δ_par, σ, ρ ⊢ ((Γ_i, ρ_i ⊢ ?_i : T_i) : prbs)) =
  prb := (Δ_par, σ, (Γ ++ Γ_i), (ρ ++ ρ_i) ⊢ T_i)
  sols := search prb
  solcolls := searchColl (Δ_par, σ, ρ ⊢ prbs)
  solcolls' := []
  for each (σ'_1, ρ'_1 ⊢ M) in sols
    for each (σ'_2, ρ'_2 ⊢ σ*) in solcolls
      case (combine (σ'_1, σ'_2, ρ'_1, ρ'_2)) of
        none → solcolls' := solcolls'
        some σ'_c ρ'_c → solcolls' := (σ'_c, ρ'_c ⊢ ((?_i := M) : σ*)) : solcolls'
      end case
    end for
  end for
  return solcolls'
```

The types of the auxiliary functions are the following:

```
createRefs : Prb → [Ref]
addSol : Sol, [Sol] → [Sol]
combine : σ, σ, ρ, ρ → σ, ρ option
compose : σ, ρ, M, σ_prf → Sol option
```

The function `search` first invokes `createRefs`, which generates the list of refinements which are valid according to the set of refinement rules. Then, for each refinement it compiles a problem collection consisting of its proof meta variables. The collection is passed to `searchColl` which returns a list of solution collections.

For each collection, `compose` is called. This function lifts the parameter instantiations and conditions above the scope of the current refinement. For the instantiations, this means removing the entries which bind a meta variable introduced by the refinement. When removing such an instantiation, the meta variable is substituted for its value in M. For the conditions, it means checking whether any of the conditioned variables was introduced by the refinement. In that case, the solution would not be valid and the function returns nothing. Otherwise, the values in σ^* are substituted into M and a solution is returned.

If `compose` returns a solution, then a call to `addSol` adds it to the list of solutions. However, if there is already a better solution present, the new one is discarded. Conversely, if the new solution is better than some old one, that one is discarded. A solution A is said to be better then a solution B if A would combine with a solution C whenever B combines with C. In other words, A is better when B when its parameter instantiations and conditions are subsets of those of B.

Moreover, when adding a solution to the list, its conditions are checked against the conditions of the already present solutions. If there is a collection of solutions which can be combined with respect to instantiations and conditions, and which together discharge one condition, the solutions are merged to a single one. The proof term of this new solution is a case-expression where the case branches correspond to the proof terms of the constituting solutions. As an example, assume that there are two solutions,

$$[h \to \texttt{inl}\ a] \vdash M \quad \text{and} \quad [h \to \texttt{inr}\ b] \vdash N\,.$$

Then they are merged into the single solution

$$\vdash \textbf{case}\ h\ \textbf{of}\ \{\texttt{inl}\ a \to M; \texttt{inr}\ b \to N\}\,.$$

The function `searchColl` first calls `search` to generate the solutions for the first problem in the collection. It then does a recursive call to produce the solution collections for the rest of the problems. For each solution and each solution collection it then invokes `combine`. This function takes a pair of parameter instantiations and a pair of conditions. It merges the instantiations and conditions while checking that no inconsistency appears. Checking this involves comparing terms. A term in an instantiation or in a condition may contain meta variables. Thus, unification is performed rather than comparing and new instantiations may need to be added. If a combination is possible, the combined sets of instantiations and conditions are returned and the result is used to construct a collection which includes a solution for the current problem.

3.2 Induction

The tactic which does case split on a variable also adds a locally defined function around the case expression. The function can in a later refinement be invoked as an induction hypothesis. Special information is stored to limit the calls to structural recursion.

Any variable which is of algebraic data type and is a parameter, i.e. appears in a type, is a candidate for the tactic.

We will now give an example. The proof steps are presented as refinements. Unlike the definitions in section 3.1, the meta variable is made explicit in problems and refinements, in order to clarify the structure of the proof search. Types are displayed in their normal form, just as they appear at unification. In applications, some type arguments are omitted to increase readability. Also, in function definitions, the type of some arguments is omitted for the same reason. Variable names essentially correspond to what the tool produces.

The problem is commutativity of addition for natural numbers.

$$[a, b : \mathit{Nat}] \vdash ? \ : \ a + b == b + a$$

Addition is defined by recursion on the first argument. The proof search calls the case split tactic, which explores analysis on a. This gives the following refinement:

$$[[a \to \mathtt{0}] \vdash ?_b : b == b + \mathtt{0}, \quad [a \to \mathtt{s}\ a'] \vdash ?_s : \mathtt{s}\ (a' + b) == b + \mathtt{s}\ a'] \vdash$$
$$? \ := \ \left(\begin{array}{l} \textbf{let}\ r\ a\ b\ :\ (a + b == b + a)\ =\ \textbf{case}\ a\ \textbf{of}\ \{\mathtt{0} \to ?_b;\ \mathtt{s}\ a' \to ?_s\} \\ \textbf{in}\ \ r\ a\ b \end{array} \right)$$

The local function is given as arguments all parameters which appear in the target type and all hypotheses whose types contain the parameters, in this case parameters a and b. The two new proof meta variables have a condition corresponding to each branch.

The base case is solved by induction on b and appealing to reflexivity, *refl*, and substitutivity for equality, *cong*.

$$\begin{array}{r} \mathit{refl}\ (X : \mathit{Set})\ :\ (x : X) \to (x == x) \\ \mathit{cong}\ (X, Y : \mathit{Set})\ :\ (f : X \to Y) \to (x_1, x_2 : X) \to (x_1 == x_2) \to (f\ x_1 == f\ x_2) \end{array}$$

The proof for the base case is constructed by the following refinements:

$$\begin{array}{l} [[b \to \mathtt{0}] \vdash ?_{bb} : \mathtt{0} == \mathtt{0}, \quad [b \to \mathtt{s}\ b'] \vdash ?_{bs} : \mathtt{s}\ b' == \mathtt{s}\ (b' + \mathtt{0})] \vdash \\ \quad ?_b \ := \ \textbf{case}\ b\ \textbf{of}\ \{\mathtt{0} \to ?_{bb};\ \mathtt{s}\ b' \to ?_{bs}\} \\ \vdash ?_{bb} \ := \ \mathit{refl}\ \mathtt{0} \\ [\vdash ?_p : b' == b' + \mathtt{0}] \vdash ?_{bs} \ := \ \mathit{cong}\ (\lambda x \to \mathtt{s}\ x)\ b'\ (b' + \mathtt{0})\ ?_p \\ \vdash ?_p \ := \ r\ \mathtt{0}\ b' \end{array}$$

The first refinement is again generated by the case split tactic. The second and third are generated by the equalities tactics and the last by the normal elimination tactic.

In the step case, the induction hypothesis corresponding to structural recursion on a is used to rewrite the equality by referring to transitivity, *tran*.

$$\mathit{tran}\ (X : \mathit{Set})\ :\ (x, y, z : X) \to (x == y) \to (y == z) \to (x == z)$$

The refinement is:

$$[\vdash ?_q : \mathtt{s}\ (b + a') == b + \mathtt{s}\ a'] \vdash$$
$$?_s \ := \ \left(\begin{array}{l} \mathit{tran}\ (\mathtt{s}\ (a' + b))\ (\mathtt{s}\ (b + a'))\ (b + \mathtt{s}\ a') \\ \quad (\mathit{cong}\ (\lambda x \to \mathtt{s}\ x)\ (a' + b)\ (b + a')\ (r\ a'\ b)) \\ \quad ?_q \end{array} \right)$$

The equalities tactic combines the use of transitivity with the use of substitutivity and symmetry.

For $?_q$, case analysis on b will lead to a solution.

$$[[b \to \mathtt{0}] \vdash ?_{sb} : \mathtt{s}\ a' == \mathtt{s}\ a',\quad [b \to \mathtt{s}\ b'] \vdash ?_{ss} : \mathtt{s}\ (\mathtt{s}\ (b' + a')) == \mathtt{s}\ (b' + \mathtt{s}\ a')] \vdash$$
$$?_q\ :=\ \begin{pmatrix} \textbf{let}\ r'\ b\ a'\ :\ (\mathtt{s}\ (b + a') == b + \mathtt{s}\ a')\ = \\ \qquad\qquad\qquad\qquad \textbf{case}\ b\ \textbf{of}\ \{\mathtt{0} \to ?_{sb};\ \mathtt{s}\ b' \to ?_{ss}\} \\ \textbf{in}\ \ r'\ b\ a' \end{pmatrix}$$

The following refinements complete the proof term:

$$\begin{aligned} &\vdash ?_{sb}\ :=\ \mathit{refl}\ (\mathtt{s}\ a') \\ &[\vdash ?_r : \mathtt{s}\ (b' + a') == b' + \mathtt{s}\ a'] \vdash \\ &\quad ?_{ss}\ :=\ \mathit{cong}\ (\lambda x \to \mathtt{s}\ x)\ (\mathtt{s}\ (b' + a'))\ (b' + (\mathtt{s}\ a'))\ ?_r \\ &\vdash ?_r\ :=\ r'\ b'\ a' \end{aligned}$$

The equalities tactic generates the first two refinements, while the elimination tactic generates the third by using the induction hypothesis ($\mathtt{s}\ (b' + a') == b' + \mathtt{s}\ a'$).

3.3 Case on Expression

There is also a tactic for case-on-expression. This tactic looks at the subexpressions of the target type and of the variables in the context. Any data type subexpression which is an application with undefined head position is subject to the tactic. All occurrences of the subexpression are replaced by a fresh variable. Then, case analysis on the new variable is added. This is a special and very useful case of generalisation. Although the occurrences of the subexpression are replaced, new instances may appear at a later stage. Therefore, a proof that the new variable equals the subexpression is supplied.

The following example illustrates the use of case-on-expression. It is about lists and the functions *map*, which is defined recursively in the normal way, and *filter*. In order to allow *filter* reducing in two steps it is defined in terms of *if*.

$$\begin{aligned} &\mathit{if}\ \mathtt{true}\ x\ y\ \ = x \\ &\mathit{if}\ \mathtt{false}\ x\ y = y \\ \\ &\mathit{filter}\ f\ \ [\mathtt{]}\qquad\ = [\mathtt{]} \\ &\mathit{filter}\ f\ \ (x :: xs') = \mathit{if}\ (f\ x)\ (x :: \mathit{filter}\ f\ xs')\ (\mathit{filter}\ f\ xs') \end{aligned}$$

The reason for defining *filter* this way is that Agda, when normalising a term, only unfolds an application when the definition of the function reduces to something which is not a case expression. This, combined with the fact the first-order unification is used, makes it necessary to define *filter* to reduce in two steps. First it reduces to an *if*-statement when the list is known to be of the form $x :: xs$. Then it reduces again when the boolean value of $(f\ x)$ is known.

The problem is the following:

$$\begin{aligned} &[X, Y : \mathit{Set}, f : X \to Y, p : Y \to \mathit{Bool}, xs : \mathit{List}\ X] \vdash \\ &\quad ? : \mathit{filter}\ p\ (\mathit{map}\ f\ xs) == \mathit{map}\ f\ (\mathit{filter}\ (\lambda x \to p\ (f\ x))\ xs) \end{aligned}$$

The proof begins with induction on xs. In the step case, the goal type reduces to:

$$\begin{aligned}&\mathit{if}\ (p\ (f\ x))\ (f\ x :: \mathit{filter}\ p\ (\mathit{map}\ f\ xs'))\ (\mathit{filter}\ p\ (\mathit{map}\ f\ xs')) ==\\ &\quad \mathit{map}\ f\ (\mathit{if}\ (p\ (f\ x))\ (x :: \mathit{filter}\ (\lambda x \to p\ (f\ x))\ xs')\ (\mathit{filter}\ (\lambda x \to p\ (f\ x))\ xs'))\end{aligned}$$

The tactic identifies $p\ (f\ x)$ for generalisation and case analysis. The occurrences are replaced by the variable *pfx*. We will not give all refinements, but simply present the final generated proof term:

$$\begin{aligned}&\mathbf{let}\ r\ xs\ :\ \mathit{filter}\ p\ (\mathit{map}\ f\ xs) == \mathit{map}\ f\ (\mathit{filter}\ (\lambda x \to p\ (f\ x))\ xs)\\ &\quad [] \to \mathit{refl}\ []\\ &\quad x :: xs' \to\\ &\qquad \mathbf{let}\ g\ \mathit{pfx}\ (\mathit{peq} : \mathit{pfx} == p\ (f\ x))\ :\\ &\qquad\quad \mathit{if}\ \mathit{pfx}\ (f\ x :: \mathit{filter}\ p\ (\mathit{map}\ f\ xs'))\ (\mathit{filter}\ p\ (\mathit{map}\ f\ xs')) ==\\ &\qquad\qquad \mathit{map}\ f\ (\mathit{if}\ \mathit{pfx}\ (x :: \mathit{filter}\ (\lambda x \to p\ (f\ x))\ xs')\ (\mathit{filter}\ (\lambda x \to p\ (f\ x))\ xs'))\\ &\qquad\quad =\ \mathbf{case}\ \mathit{pfx}\ \mathbf{of}\\ &\qquad\qquad \mathtt{true} \to \mathit{cong}\ (\lambda y \to f\ x :: y)\ (\mathit{filter}\ p\ (\mathit{map}\ f\ xs'))\\ &\qquad\qquad\quad (\mathit{map}\ f\ (\mathit{filter}\ (\lambda x \to p\ (f\ x)\ xs')))\ (r\ xs')\\ &\qquad\qquad \mathtt{false} \to r\ xs'\\ &\qquad \mathbf{in}\ \ g\ (p\ (f\ x))\ (\mathit{refId}\ (p\ (f\ x)))\\ &\mathbf{in}\ \ r\ xs\end{aligned}$$

The case-on-expression tactic generates a refinement where the local function g is defined to **case** *pfx* **of** $\{\mathtt{true} \to ?_t;\ \mathtt{false} \to ?_f\}$. Each branch is then solved by the equalities tactic.

3.4 Generalisation

The generalisation tactic recognises two cases; generalise subexpression and generalise apart. Generalise apart means replacing multiple occurrences of a single variable with two different variables. Generalising subexpression means picking a subexpression and replacing it with a new variable. It is only applied for subexpressions which occur at least twice, as opposed to the more restricted generalisation introduced by the case-on-expression tactic.

Generalise subexpression seems to be the more useful of the two. We have only made use of generalise apart for simple problems like $2 \cdot (\mathtt{s}\ n) == \mathtt{s}\ (\mathtt{s}\ (2 \cdot n))$, where multiplication is defined in terms of addition by recursion on the first argument.

We give an example to illustrate the strengthening of the induction hypothesis using generalise subexpression; reversing a list twice.

$$[X : \mathit{Set}, xs : \mathit{List}\ X] \vdash ?\ :\ \mathit{rev}\ (\mathit{rev}\ xs) == xs$$

Reversing a list is defined in terms of concatenating two lists.

$$\begin{aligned}[]\ +\!\!+\ ys &= ys\\ (x :: xs) +\!\!+\ ys &= x :: (xs +\!\!+\ ys)\\ \\ \mathit{rev}\ [] &= []\\ \mathit{rev}\ (x :: xs) &= \mathit{rev}\ xs +\!\!+ (x :: [])\end{aligned}$$

The proof begins by induction on xs. In the step case, where $xs \to x :: xs'$, the goal is rewritten using the induction hypothesis and transitivity, yielding the type:

$$rev\ (rev\ xs' +\!\!+ (x :: [])) == x :: rev\ (rev\ xs')$$

Here, $(rev\ xs')$ is generalised to a new variable, xs''. The proof then follows by induction on xs''. The final proof term is:

$$\begin{array}{l}
\textbf{let}\ r\ xs\ :\ (rev\ (rev\ xs) == xs)\ =\ \textbf{case}\ xs\ \textbf{of} \\
\quad [] \to refl\ [] \\
\quad x :: xs' \to tran\ (rev\ (rev\ xs' +\!\!+ (x :: [])))\ (x :: rev\ (rev\ xs'))\ (x :: xs') \\
\qquad (\textbf{let}\ g\ xs''\ :\ rev\ (xs'' +\!\!+ (x :: []))) == x :: rev\ xs''\ =\ \textbf{case}\ xs''\ \textbf{of} \\
\qquad\quad [] \to refl\ (x :: []) \\
\qquad\quad x' :: xs_0 \to cong\ (\lambda x \to x +\!\!+ (x' :: []))\ (rev\ (xs_0 +\!\!+ (x :: []))) \\
\qquad\qquad\quad (x :: rev\ xs_0)\ (g\ xs_0) \\
\qquad \textbf{in}\ \ g\ (rev\ xs')) \\
\qquad (cong\ (\lambda y \to x :: y)\ (rev\ (rev\ xs'))\ xs'\ (r\ xs')) \\
\textbf{in}\ \ r\ xs
\end{array}$$

3.5 Extension of First-Order Unification

The tool is based on first-order unification. Restricted to this, when unification is invoked, the tool simply normalises the terms and first-order unification is applied. The strength of this is obviously limited in a higher-order setting. To improve this without making the tool too inefficient, we have added an extension which, in a sense, does first-order unification for function meta variables. Before the first-order mechanism is called, the terms are examined. Any occurrence of a function application where the head is a meta variable is replaced by a fresh meta variable. Then the usual first-order unification is called. If it was successful, all syntactically possible ways to construct a λ-abstraction and arguments are generated. The restrictions are that the resulting application should β-reduce to the correct term and that the arguments should be type correct.

We illustrate this by a simple example. Consider substitutivity for natural numbers:

$$[P : Nat \to Set,\ x_1, x_2 : Nat] \vdash ?\ :\ x_1 == x_2 \to P\ x_1 \to P\ x_2$$

The tool will generate a proof which starts with induction on x_1 followed by induction on x_2:

$$\begin{array}{l}
? \ :=\ \left(\begin{array}{l}
\textbf{let}\ r\ (P : Nat \to Set)\ (x_1, x_2 : Nat)\ (p : x_1 == x_2)\ (q : P\ x_1)\ :\ P\ x_2 \\
\qquad\quad =\ \textbf{case}\ x_1\ \textbf{of}\ \{\mathtt{0} \to ?_b;\ \mathtt{s}\ x_1' \to ?_s\} \\
\textbf{in}\ \ r\ P\ x_1\ x_2
\end{array}\right) \\
?_s\ :=\ \textbf{case}\ x_2\ \textbf{of}\ \{\mathtt{0} \to ?_{sb};\ \mathtt{s}\ x_2' \to ?_{ss}\}
\end{array}$$

For $?_{ss}$ the problem is:

$$[\ldots, p : (\mathtt{s}\ x_1' == \mathtt{s}\ x_2'), q : P(\mathtt{s}\ x_1')], [x_1 \to \mathtt{s}\ x_1', x_2 \to \mathtt{s}\ x_2'] \vdash ?_{ss} : P\ (\mathtt{s}\ x_2')$$

Applying r will be considered by the elimination tactic. The application (r $?_P$ $?_{x_1}$ $?_{x_2}$ $?_p$ $?_q$) has the type ($?_P$ $?_{x_2}$) where $?_P : (Nat \to Set)$ and $?_{x_2} : Nat$. The unification problem to consider is thus:

$$P\ (\mathtt{s}\ x_2') \stackrel{\mathrm{u}}{=} ?_P\ ?_{x_2}$$

The application on the right hand side is replaced by a fresh meta variable, $?'$. The standard first-order unification returns the unifier $[?' := P\ (\mathtt{s}\ x_2')]$. Now the extended unification produces a unifier for each possible way to partition the expression (P ($\mathtt{s}$ x_2')) onto the function, $?_P$, and its argument, $?_{x_2}$. These are the possibilities:

$$\{?_P = \lambda x \to P\ (\mathtt{s}\ x_2')\}$$
$$\{?_P = \lambda x \to P\ x,\ ?_{x_2} = \mathtt{s}\ x_2'\}$$
$$\{?_P = \lambda x \to P\ (\mathtt{s}\ x),\ ?_{x_2} = x_2'\}$$

The third unifier leads to a terminating proof term.

4 Results

We have used the number of generated refinements as a hardware independent measure for the tool's effort when solving a problem. On a normal desktop computer, the number of generated refinements per second is around 500. The problems presented in sections 3.2, 3.3 and 3.4 take between 50 and 100 refinements to solve.

We will now present some more difficult examples which demonstrate the limits of the tool. A larger collection of problems with proofs generated by the tool can be downloaded from [8].

The first three examples are problems in the context of list sorting. The propositions are about a list being sorted, *Sort*, a list being a permutation of another list, *Perm*, and an element being a member of a list, *Member*. The functions are defined as follows:

$$\begin{aligned}
&Sorted\ [] &&= \top\\
&Sorted\ (x :: []) &&= \top\\
&Sorted\ (x :: y :: xs) &&= x \leq y \wedge Sorted\ (y :: xs)
\end{aligned}$$

$$Perm\ xs\ ys = \forall x.count\ x\ xs == count\ x\ ys$$

$$\begin{aligned}
&Member\ y\ [] &&= \bot\\
&Member\ y\ (x :: xs) &&= x == y \vee Member\ y\ xs
\end{aligned}$$

$$\begin{aligned}
&count\ y\ [] &&= \mathtt{0}\\
&count\ y\ (x :: xs) &&= if\ (eq\ x\ y)\ (\mathtt{s}\ (count\ y\ xs))\ (count\ y\ xs)
\end{aligned}$$

List concatenation and the filter function will also appear. They are defined in the normal way. The relations '$\leq$' and '$>$' will be used both to denote the boolean functions and the correspond propositions.

The first proposition claims commutativity of list concatenation with respect to permutation:

$$Perm\ (xs +\!\!+ ys)\ (ys +\!\!+ xs)$$

The tool can be set to either look for lemmas among all the global definitions or to just use the local variables. The former mode is often too inefficient. When not including the globals, the user can give a list of lemmas as hints. The problem above is solved by giving $(a, b : Nat) \to a + \mathtt{s}\ b == \mathtt{s}\ (a + b)$ as a hint and it takes 4653 refinements. The proof consists of induction on *xs* and *ys*, a few case-on-expressions and some equality manipulation.

The tool has no rewriting system for equalities. The equalities tactic modifies equalities in a quite undirected way. Due to this, a lot of effort is often spent on finding relatively small proofs for problems involving equalities.

Another property of the equality reasoning is that simple lemmas are often needed, like the one for the problem above. Although the tool can easily solve such lemmas separately, it cannot always invent them as part of another proof. This is because transitivity is only applied for an equality when there is already a known fact which can be used to rewrite either the left or the right hand side. The tool never invents an intermediate value.

The next two examples are lemmas for a correctness proof for a quicksort algorithm.

$$\begin{aligned} &Perm\ xs\ (filter\ (x >)\ xs ++ filter\ (x \leq)\ xs) \\ &Sorted\ xs \to Sorted\ ys \to ((x : X) \to Member\ x\ xs \to |x \leq a|) \\ &\qquad \to ((x : X) \to Member\ x\ ys \to |a \leq x|) \to Sorted\ (xs ++ (a :: ys)) \end{aligned}$$

The first of these is solved in 1173 refinements with two hints, namely the same as in the previous examples as well as the proposition $count\ x\ (xs ++ ys) == count\ x\ xs + count\ x\ ys$. The second example is solved in 359 refinements with no hints. The proof includes double case analysis on *xs*.

Next example is the problem to show associativity for addition of integers defined in the following way:

$$Int = \mathbf{data}\ \mathtt{Zer} \mid \mathtt{Pos}\ (n : Nat) \mid \mathtt{Neg}\ (n : Nat)$$

The proposition is

$$(p + q) + r == p + (q + r)\ .$$

The proof takes 11285 refinements and no hint is needed.

Finally, consider the problem

$$(n : Nat) \to \exists\ Nat\ (\lambda(m : Nat) \to (n == 2 \cdot m) \vee (n == \mathtt{s}\ (2 \cdot m)))\ .$$

To solve this the tool needs $2 \cdot \mathtt{s}\ n == \mathtt{s}\ (\mathtt{s}\ (2 \cdot n))$ as a hint. The problem is a very simple example of a program carrying proof, namely division by two.

The tool has a few settings, one of which should be mentioned. There is a value defining the maximum number of nested case-splits applied to a variable. In our examples, this is set to two, which is enough for all problems we have tested.

5 Conclusion and Future Work

In our opinion, the efficiency of the tool is good enough for it to be useful in a proof assistant. It can definitely save the user time. If one would consider using the tool for larger problems and allowing it more time, we think that it would perform poorly. More advanced techniques to reduce the search space would be needed. Also, since the tool in written in Haskell, it would probably suffer from the automatic memory management.

One should also ask whether the tool is versatile enough to be of practical use. We think this is the case. The tool has some fundamental restrictions, such as first-order unification. But, apart from this, our goal has been to construct a general proof search algorithm.

Instead of writing a tool which performs a proof search directly, another approach is to translate the problem and solve it with a separate first-order solver. This is currently investigated by e.g. Abel, Coquand and Norell with promising results [1]. In this approach, the problems are restricted to first-order logic and no proof term is recovered. On the other hand, it allows making use of the many highly optimized existing first-order solvers. We believe that this could be combined with our work to create a more powerful tool. Different parts of a proof could be addressed by the two components, e.g. induction by the internal proof-search and equality reasoning by the external prover. Equality reasoning has turned out to be a major bottleneck for our tool.

One feature of the tool which may not have been such a good idea is the on-demand elimination of data type objects. This adds the necessity of annotating solutions with conditions and all administration that it brings along. Another thing is the parameter/proof classification which seems a bit to rigid.

One way to continue the work could be to restart with a term synthesis algorithm based on higher-order unification, such as the one presented by Dowek [6]. This would then be enriched by first-order unification, which would serve as a short-cut in the proof search. We believe that also in a system for higher-order logic, most subterms can be resolved by first-order unification, and that it would be beneficial to have a proof search that is biased in that direction.

Another interesting issue is to deal with a dense presence false subproblems. A false subproblem may occur already when applying ordinary *modus ponens*, but if we would add abilities for lemma speculation and stronger generalisation, most subproblems would be false. Maybe one could then improve efficiency by trying to prove the negation of a possibly false subproblem in parallel. If a proof of the negation is found, the corresponding branch can be pruned.

References

1. Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. Submitted, 2005.
2. Peter B. Andrews. Classical type theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 15, pages 965–1007. Elsevier Science, 2001.

3. Catharina Coquand. The AGDA Proof System Homepage. `http://www.cs.chalmers. se/ catarina/agda/ , 1998.`
4. Christina Cornes. *Conception d'un langage de haut niveau de représentation de preuves: Récurrence par filtrage de motifs, Unification en présence de types inductifs primitifs, Synthèse de lemmes d'inversion.* PhD thesis, Université Paris 7, 1997.
5. Gilles Dowek. A complete proof synthesis method for the cube of type systems. *J. Logic and Computation*, 3(3):287–315, 1993.
6. Gilles Dowek. Higher-order unification and matching. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume 2, chapter 16, pages 1009–1062. Elsevier Science, 2001.
7. Peter Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. In *Logical frameworks*, pages 280–306. Cambridge University Press, New York, NY, USA, 1991.
8. Fredrik Lindblad. Agsy problem examples. `http://www.cs.chalmers.se/~frelindb/Agsy_examples.tgz`, 2004.
9. Lena 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.
10. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.
11. Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for lf. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300. Springer-Verlag LNCS, July 1998.
12. Tannel Tammet and Jan Smith. Optimized encodings of fragments of type theory in first-order logic. *Journal of Logic and Computation*, 8, 1998.