An Experiment in Automated Theorem Proving in Type Theory

Fredrik Lindblad

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

Abstract

A tool for automated theorem proving in type theory has been developed. It is based on the Agda language, which is an implementation of Martin-Löf type theory. Agda is used in a system for formalization of propositions and proofs in higher-order intuitionistic logic. The tool is intended to be added to this system. It would facilitate the process of proving by automatically solving problems which are not too difficult. When a solution is found, it is presented as an Agda proof term. The correctness of the proof term is verified by the Agda type-checker.
The proof search is carried out by backward reasoning. The tool implements a number of tactics which refine the given problem step by step. There are tactics which cover predicate logic. There are also tactics for case-analysis, induction and generalizations.
The tool can construct proofs containing multi-level induction and it handles equalities with some intelligence. The generalization tactic is very experimental. It is only successful in a few simple cases.
Experiments with a notion of normal forms for types have been made. The use of normal forms can restrict the amount of computations. It also removes some problems which are otherwise imposed by the fact that unification is of higher order in type theory.
A number of examples have guided the developed. These problems mainly concern natural numbers and lists. A case study of the correctness of the quicksort algorithm has been carried through. The problem was divided into a number of lemmas which can be solved by the tool one by one.

Acknowledgements

I wish to thank my supervisor Marcin Benke who has supported me for a long time. I'm grateful to Peter Dybjer and Thierry Coquand for giving me many advices and showing interest in my work. I have also been encouraged by the interest and support of many other persons.

Contents

1 Introduction

A tool for automated theorem proving in type theory has been developed. It is intended as an extension to the system for proof formalization called Alfa[1]. Alfa is a graphical interface to the Agda language[2], which is an implementation of Martin-Löf type theory[3]. Type theory can be used to formalize mathematics and program specifications. It hosts higher order logic and is in itself disposed for doing intuitionistic logic, but, by adding axioms, classical logic can also be represented. Implementations of type theory are much like functional program languages, with the addition that the output type of a function is allowed to depend on the input value. A consequence of this relaxation is the ability to represent program specification and implementation in the same language.

Alfa checks the correctness of proofs by invoking the Agda type-checker and by a separate termination assertion. It also helps the user by giving suggestions of what the next step in a proof can be. But it has not the ability to automatically construct proofs. The aim of this work has been to add some automation. The purpose is to support the user in the interactive search, not to solve large problems by itself. The user should be able to push an auto-button when she suspects that the current problem is rather simple. This way the user could spend more time on the overall structure and the crucial passages of the proof. There exists an Alfa extension for testing[5]. Automation and testing are complementary and can together be a good support to the user.

Solutions are presented as Agda terms. That means that the Agda type-checker can be used to verify the correctness of the proof found. Having external verification is satisfactory for certainty reasons. It also means that the issue of correctness can be somewhat put aside when elaborating the proof search system. The tool that has been developed principally does backwards reasoning. The aim has been to harmonize with Alfa's notion of refinements and meta variables and to make it solve a problem in a way similar to how a person would do it interactively. This way it is hoped that the resulting proof terms will be fairly readable. These criteria have sometimes been favored at the expense of speed. The tool cannot compete when it comes to finding large proofs. Furthermore, the work has had a rather experimental character. Little effort has been put into establishing theoretical results supporting the methods used. The work has resulted in a plugin for Alfa and there is also a command line version where interactive proof search can be performed with the same tactics that are used in the automatic mode. No ability to deal with the new Agda component for defining indexed inductive families[6] has been included in the implementation.

1.1 Overview of the report

Section 2 gives an introduction to the basic concepts of this work. Section 3 describes the tactics used for solving a problem. These are called refinement rules. Section 4 discusses a number of other issues related to the search algorithm, e.g. how search space is limited. Section 5 presents a few test examples which show how a proof is constructed and what the program can and cannot do. There is one longer example, which illustrates the intended use of the tool. The correctness of the quicksort algorithm is proved by splitting up the problem into smaller pieces which in turn can be solved automatically. Finally, limitations, conclusions and future work end the report.

1.2 Related work

A system for automatic theorem proving in Martin-Löf type theory has been developed by Tammet and Smith[10]. The implementation, which is called Gandalf, was designed to work with ALF, the predecessor of Agda. As for the tool of this report, Gandalf produces explicit terms, which can be checked by the type-checker. In Gandalf, the problems are however not solved directly. The authors mention the presence of higher-order unification as an obstacle of a direct approach. Instead, a theoretical basis for encoding ALF types into a representation of first-order intuitionistic logic is developed. The problems are then solved using various existing techniques for first-order logic. The authors argue that one advantage of the approach is that the efficiency of these techniques and the fact that the larger portion of the problems are essentially first-order together result in a quick theorem prover which can handle a considerable amount of problems. Effort has been made to optimize the encodings and thereby further increase the speed of problem solving. When a problem has been solved, the corresponding ALF term is extracted. Gandalf can construct inductive proofs. The encoding of such problems however seems to result in a rather time consuming search.

There is another system for formalization and proof-checking called Coq[7]. It is is similar to Agda/Alfa in the way that an implementation of type theory forms the basis of the system. When you do a proof interactively in Alfa, a clickable list of possible incomplete terms is at each step displayed. In Coq you instead have a number tactics at your disposal. There are elementary tactics, such as introducing λ-abstractions for the arguments if the current

goal is a function type, referring to a hypothesis of the problem or forming an inductive proof on a term. There are also a number of automatic tactics which perform a search. Some of these use a combination of elementary tactics recursively. Others are more stand-alone and implement automatic decision algorithms for specific fragments of what can be formalized in the system, e.g. propositional logic and first-order logic. There are also a few tactics which are devoted to special algebraic structures, like e.g. rings and fields. The standard auto tactics use a system of hints to define what part of the global environment that should be considered in the search. This set of hints is user definable and you can specify to what goals a hint should apply and also assign a cost to it. There is also a system for rewriting equalities using a user-definable set of rewriting rules.
What has been done in the project of this report is to construct a proof search tool similar to the standard auto tactics of Coq. But in contrast to Coq, the proof search has been made to automatically try tactics for induction. There is also a simple method for generalization. The diversification of search alternatives increases the complexity, so the tool does not respond as quickly as the auto tactics of Coq. The term 'goal' used in Coq is in this report called 'problem' and the elementary tactics of Coq would correspond to the refinement rules of this work.

In the context of automated theorem proving in systems based on type theory there are some important theoretical issues. These include the questions of when the existence of a proof term inhabiting a given type is decidable or when you can count the number of normal inhabitants. Other topics are the time complexity of related algorithms. In recent years, considerable work has been done in these areas. One example is the work of Broda and Damas[11]. They develop a method for representing types of simply type λ-calculus in a form called *formula-tree*. From the formula-tree of a type a *proof-tree* can then be constructed. These representations are intended to give a better understanding of the relation between types and the set of inhabitants. Simply typed λ-calculus covers only a fragment of propositional logic, which in turn is only a part of the higher-order logic which corresponds to type theory. Nonetheless, the concepts are relevant.

2 Basics of the proof search

2.1 The language

The tool that has been developed is based on the Agda language[2]. This subsection gives a brief description of the language and the notation used in this report. Only a fragment of Agda is used and the notation differs slightly. Agda is an implementation of Martin-Löf type theory. Its type system contains an infinite hierarchy of sorts; #0, #1, #2 etc. That feature is not important in the context of automated proof-search, so only one sort will be used. It is denoted by '$*$'. Single colon is used for type judgment, so $x : X$ means that x is an element of the type X. Variable substitution will be designated by $M[N/x]$, where the expression N replaces all occurrences of x in M.
There will be a quick presentation of the typing rules. They are intended to present the syntax and semantics of the language in a compact way. Note that several typing rules can be used for type-checking only, not for type inference.

Let Γ denote a variable environment and let $\Gamma(x : X)$ mean Γ extended by the declaration $x : X$. There are three kinds of types in the language; function types, signatures and algebraic data types.

$$\frac{\Gamma \vdash\ X : * \quad \Gamma(x : X) \vdash\ Y : *}{(x : X) \to Y\ :\ *}\ \text{(function formation)}$$

The output type of the function may depend on the input value. Non-dependent function types can be written in the form $X \to Y$.

$$\frac{\Gamma \vdash\ X_1 : * \quad \ldots \quad \Gamma(x_1 : X_1, \ldots, x_{n-1} : X_{n-1}) \vdash\ X_n : *}{\Gamma \vdash\ \mathbf{sig}\ \{(x_1 : X_1);\ \ldots; (x_n : X_n)\}\ :\ *}\ \left(\begin{array}{c}\text{signature}\\ \text{formation}\end{array}\right)$$

The signature represents dependent products of any finite length. The type of each constituent may depend on the value of all the preceding ones.

$$\frac{\begin{array}{l}\Gamma \vdash\ X_{1,1} : * \quad \ldots \quad \Gamma(x_{1,1} : X_{1,1}, \ldots, x_{1,n_1-1} : X_{1,n_1-1}) \vdash\ X_{1,n_1} : * \\ \qquad \vdots \\ \Gamma \vdash\ X_{m,1} : * \quad \ldots \quad \Gamma(x_{m,1} : X_{m,1}, \ldots, x_{m,n_m-1} : X_{m,n_m-1}) \vdash\ X_{m,n_m} : *\end{array}}{\Gamma \vdash \left(\begin{array}{lll}\mathbf{data} & \mathsf{c}_1\ (x_{1,1} : X_{1,1})\ \ldots\ (x_{1,n_1} : X_{1,n_1}) \\ & \vdots \\ \mid & \mathsf{c}_m\ (x_{m,1} : X_{m,1})\ \ldots\ (x_{m,n_m} : X_{m,n_m})\end{array}\right)\ :\ *}\ \left(\begin{array}{c}\text{data type}\\ \text{formation}\end{array}\right)$$

The data types can have any number of constructors, including zero, and each constructor takes any fixed number of arguments. Type dependencies may appear between the arguments of a constructor. Algebraic data types are used for disjoint unions as well as for recursive data types. Recursive definitions are expressed by referring to the variable being defined. This is not formalized here. As an example of a recursive data type, take the following definition of natural numbers:

$$Nat : * = \textbf{data}\ \mathtt{0} \mid \mathtt{s}\ (n : Nat)$$

Throughout this report, $\mathcal{S}$ will be used as a short form for the generic signature.

$$\mathcal{S} \quad \equiv \quad \textbf{sig}\ \{x_1 : X_1;\ x_2 : X_2;\ \ldots;\ x_n : X_n\}$$

Analogously, $\mathcal{D}$ will refer to the generic algebraic data type.

$$\mathcal{D} \quad \equiv \quad \left(\begin{array}{ll} \textbf{data} & \mathtt{c}_1\ (x_{11} : X_{11})\ (x_{12} : X_{12})\ \ldots\ (x_{1,n_1} : X_{1,n_1}) \\ & \vdots \\ \quad\mid & \mathtt{c}_m\ (x_{m,1} : X_{m,1})\ (x_{m,2} : X_{m,2})\ \ldots\ (x_{m,n_m} : X_{m,n_m}) \end{array} \right.$$

The following rules introduce members of the respective types. First there is λ-abstraction:

$$\frac{\Gamma \vdash\ (x : X) \to Y\ :\ * \quad \Gamma(x' : X) \vdash\ M : Y[x'/x]}{\Gamma \vdash\ \lambda(x' : X) \to M\ :\ (x : X) \to Y}\ \text{(function introduction)}$$

Then structures, which are the elements of signature types. Each item in the signature is assigned to a term, M_k.

$$\frac{\begin{array}{l}\Gamma \vdash\ \mathcal{S} : * \\ \Gamma \vdash\ M_1 : X_1 \quad \ldots \quad \Gamma \vdash\ M_n : X_n[M_{n-1}/x_{n-1}, \ldots, M_1/x_1]\end{array}}{\Gamma \vdash\ \textbf{struct}\ \{x_1 = M_1;\ \ldots;\ x_n = M_n\}\ :\ \mathcal{S}}\ \left(\begin{array}{c}\text{signature}\\ \text{introduction}\end{array}\right)$$

Substituting the variables for the elements' values is done in descending order since the rightmost preceding variable is the most local.
Elements of a data type are introduced by using any of its constructors.

$$\frac{\begin{array}{l}\Gamma \vdash\ \mathcal{D} : * \\ \Gamma \vdash\ M_1 : X_{j,1}\ \ldots\ \Gamma \vdash\ M_{n_j} : X_{j,n_j}[M_{n_j-1}/x_{j,n_j-1}, \ldots, M_1/x_{j,1}]\end{array}}{\Gamma \vdash\ \mathtt{c}_j\ M_1\ \ldots\ M_{n_j}\ :\ \mathcal{D}}\ \left(\begin{array}{c}\text{data type}\\ \text{introduction}\end{array}\right)$$

The rule is valid for any j between 1 and m.

Elements are eliminated using function application, projection and case expression. First we have application:

$$\frac{\Gamma \vdash\ M : (x : X) \to Y \quad \Gamma \vdash\ N : X}{\Gamma \vdash\ M\ N\ :\ Y[N/x]} \left(\begin{array}{c}\text{function}\\ \text{elimination}\end{array}\right)$$

Since the output type may be dependent, x must be substituted by N in Y. Next we have projection which picks one of the structures' components, say x_k:

$$\frac{\Gamma \vdash\ M : \mathcal{S}}{\Gamma \vdash\ M.x_k\ :\ X_k[M.x_{k-1}/x_{k-1}, \ldots, M.x_1/x_1]} \left(\begin{array}{c}\text{signature}\\ \text{elimination}\end{array}\right)$$

References to preceding variables are qualified by inserting the correct projection. For data types we have case expression:

$$\frac{\begin{array}{l}\Gamma \vdash\ y : \mathcal{D}\\ \Gamma \vdash\ T : *\\ \Gamma'_1 \vdash\ N_1 : T'_1\\ \vdots\\ \Gamma'_m \vdash\ N_m : T'_m\end{array}}{\Gamma \vdash \left(\begin{array}{ll}\textbf{case}\ y\ \{ & \mathtt{x}_1\ x'_{1,1}\ \ldots\ x'_{1,n_1} \to N_1;\\ & \vdots\\ & \mathtt{x}_m\ x'_{m,1}\ \ldots\ x'_{m,n_m} \to N_m\\ \quad\quad\ \} & \end{array}\right) : T} \left(\begin{array}{c}\text{data type}\\ \text{elimination}\end{array}\right)$$

To conclude that a case expression is of type T we must check that the body of each branch, N_j, is member of T'_j which is T specialized to that particular branch.

$$T'_j \ \equiv\ T[\mathtt{c}_j\ x'_{j,1}\ \ldots\ x'_{j,n_j}/y]$$

In each branch, all arguments of the constructor can be accessed by referring to any x'_{jk}.

$$\Gamma'_j \ \equiv\ \Gamma(x'_{j,1} : X_{j,1}, \ldots, x'_{j,n_j} : X_{j,n_j}[x'_{j,n_j-1}/x_{j,n_j}, \ldots, x'_{j,1}/x_{j,1}])$$

The object which is inspected in the case-expression is called the *scrutinee*. In the rule above it is intended to be a variable. It may however also be any type correct expression, but in this case no specialization of the the type T is performed when checking the branches.

The terms presented above are reduced by β-reduction and by picking components of compound data.

$$(\lambda(x : X) \to M)\ N \to M[N/x]$$

$$(\textbf{struct}\ \{x_1 = M_1;\ \ldots;\ x_n = M_n\}).x_k \to M_k$$

$$\left.\begin{array}{l} \textbf{case}\ \mathsf{c}_j\ M_1 \ldots M_{n_j}\ \{\mathsf{c}_1\ x'_{1,1} \ldots x'_{1,n_1} \to N_1; \\ \qquad\qquad\qquad\quad \vdots \\ \qquad\qquad\qquad\quad \mathsf{c}_m\ x'_{m,1} \ldots x'_{m,n_m} \to N_m \\ \qquad\qquad\qquad\quad \} \end{array}\right\} \to N_j[M_{n_j}/x'_{j,n_j}, \ldots, M_1/x'_{j,1}]$$

In addition, identifiers of defined objects are of course replaced by their value. Apart from these basic components, also the term construction **let** ... **in** ... will appear. It introduces local definitions and is used in the proof search for defining recursive functions. All occurrences in this report will have the following form:

$$\begin{array}{l} \textbf{let}\ f\ (\vec{x} : \vec{X}) : T\ = M \\ \textbf{in}\ \ f\ \vec{N} \end{array}$$

The symbols $\vec{x}$, $\vec{X}$ etc. denote collections of entities. The term defines a function, f, which takes a number of arguments. The function is invoked in the body of the expression. The continuation of the proof is dwelled in M, where reference to f can be made.

A few comments will conclude this subsection. Signatures are not necessary. They could be replaced by one-constructor-data types. The presence of signatures however adds some convenience, since their elements are easily accessed using projection. Agda has a system for making structured code, called *packages*. This is described in [9]. The implemented tool partly supports packages, but they do not appear in this report. The new amendment to Agda for defining indexed inductive families, **idata**, is not covered in this work. Note that termination issues lie outside the Agda type-checker. There is a separate termination check. Solutions which are generated by the tool sometimes do not pass the termination check. As far as we know, termination still holds in those cases.

2.2 The basic concepts of the proof search

The implemented tool for automated proving is intended to be used within a system for doing proofs interactively. It has therefore been designed to solve a problem in a way similar to how a human would do it. This way it is not so difficult to produce answers that can easily be understood. It also makes it possible to use the tool itself in a step-by-step mode to do interactive search. The tool produces solutions as Agda proof terms. These are inserted into the definition of the currently selected proposition. It works with Agda expressions and environments throughout the search, i.e. a problem is not translated to another representation before it is solved. The search algorithm of the tool in most parts works in a backward reasoning manner, beginning with the goal and then using deduction rules backwards to produce subgoals corresponding to the premises.

The basic concepts of the proof search are *target type*, *meta variable* and *refinement*. These are adopted from Alfa[1] in a slightly modified sense. A presentation of the concepts can be found in [9]. The target type is the formulation of a proposition as an Agda type. A problem is completely represented by a target type together with a variable environment in which an element of the target type is to be found. If such an element is found, it is an Agda term proving the proposition. During the search, meta variables are used as place holders for subexpressions which are not yet known. These are denoted by indexed question-marks. If we have an initial, or global, environment Γ and a target type T, then the initial problem of the search can be depicted like this:

$$\Gamma \vdash ? : T$$

The task is to find an expression not containing meta variables which can replace ?. This is done by successively replacing a meta variable by an expression possibly containing new meta variables. The new meta variables correspond to the subproblems, or, in other words, the premises of the inference rule that was used backwards. For each problem several rules may apply. These give rise to different proof terms and subproblems. The collection of partially given proof terms together with their associated subproblems are called the refinements of the problem. Except for those that contain no subproblems, it is difficult to say which refinements will lead to a solution. Thus they are all tested one by one.

The formulation of the initial problem must be extended a bit to encompass all subproblems. The information for each new meta variable will be stored in a *meta environment*. For each meta variable, the meta environment gives the variable environment and the type associated to the meta variable. You can visualize the meta environment as a list of entities of the form

$$\Gamma \vdash ? : T$$

The letter Δ will be used for meta environments. The meta environment not only dictates the variable environment and type of the current subproblem. In addition, meta variables may appear in the types of a subproblem. Variable environments will be denoted by Γ in order to avoid confusion.
One more entity may appear in a problem. The way data type elimination is handled by the tool, solutions to subproblems are tagged with a list of *value constraints*. These are statements which specify which form a data type term must have in order to make the solution valid. In the end, no such constraints may of course remain for the solution of the main problem. Solutions with value constraints are combined in a case expression to generate a solution with less constraints. This is explained in detail in subsections 3.2 and 4.1. The value constraints have the form

$$M = \mathsf{c}_j \ x_{j,1} \ \ldots \ x_{j,n_j}$$

The significance of such a constraint is that the term M must reduce to a term with the constructor c_j in the head position in order for the solution to be valid. Value constraints will be denoted by σ.
We can now write the general form of a problem. First there is a meta environment, then a variable context, Γ, which is associated to the meta variable of the subproblem. After this there may be a set of value constraints, which refer to variables in Γ. In this environment, a proof term should be found which have the type of the meta variable.

$$\Delta, \Gamma, \sigma \vdash \ ? \ : \ T$$

There is a number of different rules used by the tool to generate refinements. But they are all expressed in a uniform way to allow them to be processed equally. The refinement rules are much like the typing rules of the system turned upside-down. They are the *tactics* of the program. A refinement consists of a proof term. If the proof term is not complete, i.e. it contains new meta variables, the meta environment must be accordingly extended. New value constraints may also have emerged. In addition, the refinement can induce *meta variable constraints*. These have the form

$$? = M$$

A meta variable constraint says that the refinement is valid only if the meta variable is bound to the specified term. Collections of meta variable constraints will be denoted by ρ.
The refinement rules will have this general form:

$$\frac{\Delta, \Gamma, \sigma \vdash \ ? \ : \ T \quad < \text{side conditions} >}{\Delta', \rho, \Gamma, \sigma' \vdash \ ? \rightsquigarrow M \ : \ T}$$

The meaning of such a rule is the following. Above the line the problem is given, together with any side conditions which are specific to each refinement rule. Below the line is the refinement. The significance of the rule is that the refinement is valid for the problem when the side conditions hold. This means that the term M can replace ?. The new meta environment, Δ', is Δ extended with the new meta variables and ρ is the meta variable constraints of the refinement. Finally, σ' is σ extended with additional value constraints. There is of course a strict correspondence between the variable environment and the type of a meta variable on the one hand and its location in the proof term on the other. Otherwise the refinement rule would be violating the typing rules. The proof term must be type correct assuming that the meta variables have the specified target types. Also, the variable environment of each meta variable is an extension of the last one and the meta variable must in the proof term be preceded by the corresponding variable declarations, such as a λ-abstraction. No check is performed concerning this. It is up to the methods that generate the refinements to respect this correspondence.

Refinements are recursively generated as long as new subproblems arise. When a refinement with no subproblems is generated, it is a *solution* to the current problem. A refinement that does have subproblems is combined with the solutions of the subproblems, if such were found. Any solution that reach the top of the search potentially solves the given problem and contains all the necessary meta variable assignments to compile the proof term. Apart from the proof term, solutions also inherit the meta variable and value constraints of the constituting refinements. The concept of solutions is further described in subsections 4.1 and 4.3.

One notion which is important to the proof search is the *proof/parameter classification*. All objects are classified as either proofs or parameters. By object we mean both variables and meta variables. Parameters are those which appear in other objects' types, i.e. those which other objects depend on. The rest are considered to be proofs. Note that a parameter can depend on another parameter. The classification affects the way case analysis is carried out. It is also used to determine which new meta variables should be treated as subproblems. Only proofs should be searched for. Searching for parameters is trivial and the result is arbitrary. The parameters' values should instead be settled as a side effect when searching for solutions to the proof objects.

Section 3 describes the collection of rules that are used to generate refinements. There are rules which construct members of function types, data types and signatures. Then there is a rule that incorporates every possible way to use elements declared in the environment. This corresponds to invoking global lemmas, problem premises and induction hypotheses as well as using the elimination rules for data types and signatures. A special method for using the hypotheses of the problem is employed when the target type is an equality. It has some built-in knowledge which allows reflexivity, symmetry, transitivity and substitutivity to be utilized in a better way than would be the case if the standard method would do the job. One rule introduces case analysis for data type variables. For recursively defined data types the generated refinement is potentially the starting point of an inductive proof. When dealing with an existence proof it is sometimes beneficial to make an assumption on the form of the witness. If, for instance, the witness is a natural number, the assumptions would be that it is either zero or the successor of a new arbitrary number. One refinement rule is devoted to this task. Finally, there is a rule which uses a couple of simple schemes to generalize the given problem. Generalization is sometimes imperative in order to find a solution. The program manages to do a correct generalization in some simple cases.

2.3 Expression reduction and comparison

Comparing types is essential to the proof search. At each step the target type is compared to the type of every relevant object in the environment. One approach is to include term reduction in the comparison. If the two types are not equal, they are further reduced and compared again. This however entails the same computation being done many times. Instead, one can normalize the types in advance and thus remove this step in the comparison. Experiments have been done with both approaches, but the program currently works along the latter.

When types are stored in a normalized form, reduction is considerably limited. Only α-conversion is performed in the actual comparison. One problem with normalizing in advance is that the types often becomes very long. The length not only makes the types highly unreadable. It also slows down the comparison. So, instead of full normalization, the types are reduced to a level where they are still fairly small and readable. This is here called *simplification* of types. When simplifying a type, computation is carried on until the next reduction step would introduce a non-reducible λ-abstraction, **case**-expression, **data**-expression or signature. This means that simplified types normally consist of applications where the terms are either identifiers, projections or λ-abstractions deriving from the form that was stated by the user.

When a type is simplified, reduction is not only performed on the head of

the expressions. Also the subexpressions are reduced in the same way. This is necessary since there are refinement rules that do not just look at the head of the target type. The method for equalities examines every subexpression to see if it can be replaced using substitutivity. However, when simplifying subexpressions some caution must be used to avoid obscure effects of reducing terms within the scoop of local declarations.
Using simplified types to do comparison entails the risk that two expressions that actually represent the same type, i.e. are convertible, is judged to be unequal. The simplification is designed to make most equal types syntactically equal modulo α-conversion. But in some situations this is not the case. To avoid this, a few technical things can be kept in mind when stating the problem. The speed that is gained when pre-simplifying types makes the approach motivated. We also think that the simplification could be improved to achieve higher fidelity to actual equality. Apart from improving speed, the question of how far canonical expressions can be used for types is of theoretical interest.

One effect of type dependencies is that meta variables can occur in types. A meta variable represents an arbitrary value. This means that comparing two types involves assigning meta variables to values. Comparison is therefore replaced by a form of unification. In type theory, higher-order unification is required. This means that you have to take conversion equality into account when unifying. Higher-order unification is undecidable. Confer for instance [12]. This problem is avoided here since pre-simplified types are used and thus no reduction is allowed when unifying. But α-conversion is active at the comparison and can therefore cause problems. A pragmatic attitude towards this fact has been taken. The unification used in the program simply disregards the remaining problems inflicted by the higher-order nature. This may of course lead to errors. On the other hand, the system of simplified types minimizes the occurrence of λ-expressions. This should reduce the amount of unification malfunction.

The unification works like comparison except that a list of meta variable bindings is carried around. Whenever a point is reached where at least one of the subexpressions consists of a meta variable, binding is tried in order to unify the types. If a meta variable already appears in the list of bindings, it is replaced by its value and comparison continues. If a meta variable is instead previously unbound, it is assigned to the value of the opposite expression. In addition, no binding which would introduce circular referencing is allowed. This way of dealing with meta variables appearing in types works fine for data type meta variables, which are most common. But meta variables can also be functions or structures. These must be treated with additional care. Some experiments addressing this issue have been done, but the the program does not currently support it. A discussion on this can be found in subsection 6.1.

If the unification succeeds, the list of meta variable bindings is returned. The bindings are then used as constraints of the resulting refinement. Unification will be denoted as in the following example. Assume that two types, T_a and T_b, are unified in the meta environment Δ and variable environment Γ.

$$\Delta, (?_1 = M_1, ?_2 = M_2), \Gamma \vdash\ T_a \ \stackrel{\mathrm{s}}{=}\ T_b$$

In the example the unification entails the binding of two meta variables. The symbol '$\stackrel{\mathrm{s}}{=}$' means that the types are first simplified and then unified. Unification is often denoted by '$\doteq$'. Let *simp* be the function that takes a variable environment and a type and returns the simplified form of the type. Informally we could then define the relation like this:

$$\Delta, \rho, \Gamma \vdash\ T_a \ \stackrel{\mathrm{s}}{=}\ T_b \quad \equiv \quad \Delta, \rho, \Gamma \vdash\ \mathit{simp}(\Gamma, T_a) \doteq \mathit{simp}(\Gamma, T_b)$$

where ρ is the meta variable constraints of the unification.

The types are however not only compared to each other. In some refinement rules ordinary head normalization is needed, denoted by the symbol '$\stackrel{n}{\twoheadrightarrow}$'. When normalization is used, the conclusion is always either of the following.

$$T \ \stackrel{\mathrm{n}}{\twoheadrightarrow}\ (x : X) \to Y \qquad T \ \stackrel{\mathrm{n}}{\twoheadrightarrow}\ \mathcal{S} \qquad T \ \stackrel{\mathrm{n}}{\twoheadrightarrow}\ \mathcal{D}$$

The normalization depends on the definitions in the variable environment, but this is not indicated by this notation.
Above, $\mathcal{S}$ and $\mathcal{D}$ refer to the generic signature and data type defined in subsection 2.1.

Although computation is limited using pre-simplified types, many problems involves repeated re-simplification. When the program tries to construct a proof by induction on a parameter, the types are specialized for each case-branch. When specializing a type it may turn into a non-simplified state, so it must be simplified again. Computation can therefore not be completely avoided in the actual proof search, so some measures have been taken to speed up computation, e.g. sorting the context and package elements in lexicographic order and pre-inserting the necessary qualifiers, i.e. projections, in the elements of packages.

3 Refinement rules

In this section all the implemented refinements rules are presented. The refinements represent a single step in the proof search. How the refinements are combined to create a solution, i.e. the search algorithm, is described in section 4.
In the rules below it is assumed that the given problem is

$$\Delta, \Gamma, \sigma \vdash ? \ : \ T$$

The notation for refinement rules used here was introduced in subsection 2.2. New meta variables that are introduced in the rules are given various subscripts. It is however implicitly assumed that they all represent fresh meta variables, i.e. previously unused.

3.1 Introduction rules

If the target type normalizes to a function type, the analogous λ-abstraction is a refinement.

$$\frac{\Delta, \Gamma, \sigma \vdash ? \ : \ T \qquad T \overset{\mathrm{n}}{\twoheadrightarrow} (x : X) \rightarrow Y}{\Delta(\Gamma(x : X) \vdash ?_Y : Y), \Gamma, \sigma \vdash ? \rightsquigarrow (\lambda(x : X) \rightarrow ?_Y) \ : \ T} \left(\begin{array}{c}\text{abstraction}\\ \text{refinement}\end{array}\right)$$

The requirement on the normal form of T appears as a side condition to the rule. A new meta variable associated to the body of the function is added. Its variable environment is extended with the parameter x. When new variables are introduced, like in the refinement above, it may of course be the case that the variable name is already in use. Name clashes could lead to the blocking of a solution. Fresh variable names should therefore always be picked. This is however for simplicity reasons not indicated in the abstraction refinement, nor in any of the other refinements which declares new variables.

Abstraction refinement in logical terms corresponds to $\forall$-introduction and implication. We now move on to the counterpart of $\wedge$- and $\vee$-introduction, as well as $\exists$-introduction.
Let $\mathcal{S}$ be the generic signature defined in subsection 2.1. If target type normalizes to a signature, the corresponding structure is a refinement.

$$\frac{\Delta, \Gamma, \sigma \vdash ? \ : \ T \qquad T \overset{\mathrm{n}}{\twoheadrightarrow} \mathcal{S}}{\Delta', \Gamma, \sigma \vdash ? \rightsquigarrow \textbf{struct}\ \{x_1 = ?_1; \ \ldots; \ x_n = ?_n\} \ : \ T} \left(\begin{array}{c}\text{signature constr.}\\ \text{refinement}\end{array}\right)$$

The new meta environment, Δ', is Δ extended with one new meta variable for each constituent of the structure.

$$\Delta' \ \equiv \ \Delta(\Gamma \vdash ?_1 : X'_1, \ldots, \ \Gamma \vdash ?_n : X'_n)$$

The target types of the subproblems, X'_k, are equal to X_k unless there is type dependency between the signature elements. In that case the appropriate meta variables must be substituted into the types. Generally, the following substitutions are carried out.

$$X'_k \;\equiv\; X_k[?_{k-1}/x_{k-1}]\ldots[?_2/x_2][?_1/x_1]$$

As indicated in the rule above, the context for the constituents of a structure is not extended with preceding identifiers. In Agda you can do this kind of reference within a structure. It is however only useful as an abbreviation for something that appears more than once. The program does not know that two objects can be chosen to be equal until it has actually synthesized the same thing twice. Not extending is on the other hand cheaper.

If the target type instead normalizes to a data type, any of the constructions is a refinement.

$$\frac{\Delta,\Gamma,\sigma \vdash ?\ :\ T \qquad\qquad T \overset{\mathrm{n}}{\twoheadrightarrow} \mathcal{D}}{\Delta'_j,\Gamma,\sigma \vdash ? \rightsquigarrow (\mathtt{c}_j\ ?_1\ \ldots\ ?_{n_j})\ :\ T} \left(\begin{array}{c}\text{data type constr.}\\ \text{refinement}\end{array}\right)$$

The refinement rule is valid for any j between 1 and m, i.e. for each constructor of the data type. The meta environment for the refinement is extended in the same way as in the case of signature type.

$$\Delta'_j \;\equiv\; \Delta(\Gamma \vdash ?_1 : X'_{j,1}, \ldots,\ \Gamma \vdash ?_{n_j} : X'_{j,n_j})$$

Also the target types of the new meta variables are modified in the same way.

$$X'_{jk} \;\equiv\; X_{jk}[?_{k-1}/x_{j,k-1}]\ldots[?_2/x_{j,2}][?_1/x_{j,1}]$$

In the cases where there are type dependencies, meta variables will enter the target types. Later on they may also appear in the variable environment, as a consequence of further λ-abstractions. This is why unification of types is required and also why refinements and sub solutions are tagged with meta variable constraints.

Examples

There will now be a number of concrete refinements that exemplify the rules. They correspond to the usual introduction rules of logic.

$\rightarrow$-introduction, $P \rightarrow Q$

$$\frac{\Delta,\Gamma,\sigma \vdash ?\ :\ T \qquad\qquad T \overset{\mathrm{n}}{\twoheadrightarrow} P \rightarrow Q}{\Delta(\Gamma(p:P) \vdash ?_Q : Q),\Gamma,\sigma \vdash ? \rightsquigarrow (\lambda(p:P) \rightarrow ?_Q)\ :\ T}$$

$\forall$-introduction, $(x : X) \to P\ x$

$$\frac{\Delta, \Gamma, \sigma \vdash ?\ :\ T \qquad T \overset{\mathrm{n}}{\twoheadrightarrow} (x : X) \to P\ x}{\Delta(\Gamma(x : X) \vdash ?_b : P\ x), \Gamma, \sigma \vdash ? \leadsto (\lambda(x : X) \to ?_b)\ :\ T}$$

$\wedge$-introduction, $P \wedge Q$
Let

$$(\wedge)\ (X,\ Y : *)\ :\ *\ =\ \mathbf{sig}\ \{x : X;\ y : Y\}$$

The refinement will be

$$\frac{\Delta, \Gamma, \sigma \vdash ?\ :\ T \qquad T \overset{\mathrm{n}}{\twoheadrightarrow} P \wedge Q}{\Delta', \Gamma, \sigma \vdash ? \leadsto \mathbf{struct}\ \{x = ?_x;\ y = ?_y\}\ :\ T}$$

where

$$\Delta' \ \equiv\ \Delta(\Gamma \vdash ?_x : P,\ \Gamma \vdash ?_y : Q)$$

$\exists$-introduction, $\exists\ X\ P$
Let

$$\exists\ (X : *)\ (Y : X \to *)\ :\ *\ =\ \mathbf{sig}\ \{w : X;\ p : Y\ w\}$$

With this definition the introduction refinement will be

$$\frac{\Delta, \Gamma, \sigma \vdash ?\ :\ T \qquad T \overset{\mathrm{n}}{\twoheadrightarrow} \exists\ X\ P}{\Delta', \Gamma, \sigma \vdash ? \leadsto \mathbf{struct}\ \{w = ?_w;\ p = ?_p\}\ :\ T}$$

where

$$\Delta' \ \equiv\ \Delta(\Gamma \vdash ?_w : X,\ \Gamma \vdash ?_p : P\ ?_w)$$

$\vee$-introduction, $P \vee Q$
Let

$$(\vee)\ (X,\ Y : *)\ :\ *\ =\ \mathbf{data}\ \mathtt{inl}\ (x : X) \mid \mathtt{inr}\ (y : Y)$$

We have two refinements, one for each constructor.

$$\frac{\Delta, \Gamma, \sigma \vdash ?\ :\ T \qquad T \overset{\mathrm{n}}{\twoheadrightarrow} P \vee Q}{\begin{array}{c}\Delta_l, \Gamma, \sigma \vdash ? \leadsto \mathtt{inl}\ ?_l\ :\ T \\ \Delta_r, \Gamma, \sigma \vdash ? \leadsto \mathtt{inr}\ ?_r\ :\ T\end{array}}$$

where

$$\begin{array}{rcl}\Delta_l & \equiv & \Delta(\Gamma \vdash ?_l : P) \\ \Delta_r & \equiv & \Delta(\Gamma \vdash ?_r : Q)\end{array}$$

3.2 Elimination rules

In the case of introduction rules, the search is well guided by the target type. For elimination rules the situation is different. The motivation to use an elimination rule should always be that there exists a variable in the environment that can be eliminated accordingly. If we established an elimination rule without taking this into account, standard left component $\wedge$-elimination would look something like this:

$$\frac{\Delta, \Gamma, \sigma \vdash \; ? \; : \; T}{\Delta(\Gamma \vdash ?_* : *, \Gamma \vdash ?_1 : T \wedge ?_*), \Gamma, \sigma \vdash \; ? \rightsquigarrow ?_1.x \; : \; T}$$

Here, '$\wedge$' refers to the same definition as in the examples of the previous subsection. The problem becomes clear in the rule, because it does not have any side conditions. The refinement is valid for any target type. Including such a rule would lead to a fatal amount of blind search. What we want is to first look at the variables in the environment. If part of an object matches the target type we then choose the suitable elimination. For $\wedge$-elimination the rule would instead be

$$\frac{\Delta, \Gamma, \sigma \vdash \; ? \; : \; T \quad \Gamma \vdash \; p : X \wedge Y \quad \Delta, \Gamma \vdash \; T \stackrel{\mathrm{s}}{=} X}{\Delta, \Gamma, \sigma \vdash \; ? \rightsquigarrow p.x \; : \; T}$$

Here we demand the presence of a conjunction where the left conjunct equals to the target type.

We need to generalize this idea to embrace all kinds of objects. There are three kinds; function types, signatures and data types. The elimination of each of them can be turned into a general rule. But this is not enough since types can be compound in more than one step. We can for instance have any of these types.

$$A \to B \wedge C, \quad (A \to B) \vee (C \to D), \quad (A \vee B) \wedge (C \to D)$$

To handle these types, repetitive elimination must be allowed within the same refinement.
We add a new kind of judgement, called *elimination judgement*. It has the following general form:

$$\Delta, \Gamma, \sigma \vdash \; M \; : \; U$$

It means that given the meta environment Δ, the variable environment Γ and value constraints σ, the term M is of type U. Elimination judgements will result from a number of *elimination inference rules* which are listed further on.

By referring to an elimination judgement, we can now formulate a general elimination refinement rule.

$$\frac{\begin{array}{c}\Delta \subseteq \Delta', \quad \sigma \subseteq \sigma' \\ \Delta', \Gamma, \sigma' \vdash \ M \ : \ U \\ \Delta, \Gamma, \sigma \vdash \ ? \ : \ T \quad \Delta', \rho, \Gamma \vdash \ T \overset{\mathrm{s}}{=} U\end{array}}{\Delta', \rho, \Gamma, \sigma' \vdash \ ? \rightsquigarrow M \ : \ T} \left(\begin{array}{c}\text{elimination}\\ \text{refinement}\end{array}\right)$$

It is similar to the second version of $\wedge$-elimination above. There are four side conditions. The two at the top say that Δ' is an extension of Δ and σ' an extension of σ. This is guaranteed due to the constitution of the elimination inference rules which we will use. The third side condition is the elimination judgement saying that M is of type U. The last one assures that the target type can be unified with U. An elimination judgement can include extension of both the meta environment, Δ, and the set of value constraints, σ. The extensions are reflected in the resulting refinement. The unification may produce meta variable constraints, ρ, which are also added to the refinement. A slightly disturbing detail of the rule is that T appears in the environment Δ but is then unified in Δ', but this is not a problem since an extension of a meta environment always consists of fresh meta variables.

In order to be able to use the elimination refinement rule, we need a machinery to produce elimination judgements. This is the role of the elimination inference rules. The idea is to apply these rules iteratively on every object in the environment. This way every possible elimination will be generated, even for objects which are compound in several steps.
First we have the initial and trivial rule that every variable can be used as it is.

$$\frac{\Gamma \vdash \ x : X}{\Delta, \Gamma \vdash \ x \ : \ X} \text{ (reference)}$$

The occurrence of x above the line should be interpreted as the variable and the one below the line as the term referring to x. This rule is used as the starting point for each variable.
After this we have the actual elimination inference rules. They all require a previous elimination judgement. A chain of elimination judgements is formed by starting with a reference inference rule. First we have function invoking:

$$\frac{\Delta, \Gamma, \sigma \vdash \ M \ : \ U \qquad U \overset{\mathrm{n}}{\twoheadrightarrow} (x : X) \to Y}{\Delta(\Gamma \vdash \ ?_X : X), \Gamma, \sigma \vdash \ (M \ ?_X) \ : \ Y[?_X/x]} \text{ (application)}$$

The side condition is of course that U normalizes to a function type. Function invoking extends Δ with a subproblem corresponding to the argument of the application. For type dependent functions, the new meta variable will appear in the type of the elimination.

Next, if the type of the current elimination judgement normalizes to a signature, projection can be used to generate a new judgement for each element.

$$\frac{\Delta, \Gamma, \sigma \vdash \ M \ : \ U \qquad\qquad U \ \overset{\mathrm{n}}{\twoheadrightarrow} \ \mathcal{S}}{\Delta, \Gamma, \sigma \vdash \ M.x_k \ : \ X'_k} \text{ (projection)}$$

where k is between 1 and n. The types of the new judgements must be modified so that all references to other elements are correctly qualified.

$$X'_k \ \equiv \ X_k[s.x_{k-1}/x_{k-1}] \ldots [s.x_2/x_2][s.x_1/x_1]$$

We must also be able to do data type elimination. In Agda the elimination of such objects is done using a case-expression. Data type elimination introduces a new problem. Is is not enough to require that there must exist an appropriate data type object before an elimination refinement is generated. The rule would still be applicable to any target type. This is because there is no connection between the eliminated type and the inferred type. Ordinary $\vee$-elimination requires a premise of the form $A \vee B$, but can be used to prove any proposition C if also $A \to B$ and $A \to C$ are provided. As a consequence it is difficult to know at which point the elimination involving a certain data type object should appear and if it should take place at all. The approach chosen here is to search the parts of every construction for data types just as if they are all accessible. If a match is encountered a refinement with a value constraint is generated. Value constraints were first discussed in subsection 2.2. The constraint says the object in question reduces to one specific construction. The idea is that solutions with complementary value constraints later on will be combined to form one solution with no constraints. More about how this is done is discussed in subsection 4.1.

The data type elimination inference rule looks like this:

$$\frac{\Delta, \Gamma, \sigma \vdash \ M \ : \ U \qquad\qquad U \ \overset{\mathrm{n}}{\twoheadrightarrow} \ \mathcal{D}}{\Delta, \Gamma, \sigma(M = \mathsf{c}_j \ x_{j,1} \ \ldots \ x_{j,n_j}) \vdash \ x_{jk} \ : \ X_{jk}} \text{ (data type elimination)}$$

where $j \in [1, m]$ and $k \in [1, n_j]$. There are $\sum_j n_j$ possible elimination judgements, one for each component of the data type. For each judgement, the set of constraints is extended with the constraint that M has the form of a certain construction. Judgements belonging to the same constructor have identical value constraints. Note that the term of the new judgement refers to a variable that only appears in the value constraints. The constraints thus in a sense serves as an extra variable environment.

There is one situation which is not covered by the main elimination refinement rule. When there is a member of the empty type in the environment, it can used to prove anything. Or, in logical terms, if one of the hypotheses is the absurdity we can draw any conclusion. A special elimination rule is required. The empty type is in Agda represented by a data type with no constructors. In accordance with the typing rules, a case expression with no branches matches any type. We add the following rule for absurdity elimination:

$$\frac{\begin{array}{c} \Delta \subseteq \Delta', \quad \sigma \subseteq \sigma' \\ \Delta', \Gamma, \sigma' \vdash \ M \ : \ U \\ \Delta, \Gamma, \sigma \vdash ? \ : \ T \qquad U \overset{\mathrm{n}}{\twoheadrightarrow} \mathbf{data} \end{array}}{\Delta', \Gamma, \sigma' \vdash ? \leadsto (\mathbf{case}\ M\ \{\}) \ : \ T} \left(\begin{array}{c} \text{absurd. elim.} \\ \text{refinement} \end{array} \right)$$

The difference compared to the standard elimination refinement rule is that instead of performing unification with T, we demand that U normalizes to the empty type.

Example

Now follows a trivial example illustrating the ideas of elimination judgements and refinement. It contains several refinement steps in order to fully illustrate how the program does $\vee$-elimination. Let the proposition to prove be

$$(A, B, C, D : *) \to A \wedge B \to (A \to C) \vee (B \to D) \to C \vee D$$

We assume that all the abstractions have already been done, so we are facing the problem

$$\Delta \ \equiv \ (\Gamma \vdash ? : C \vee D)$$

where

$$\Gamma \ \equiv \ (A, B, C, D : *, \ p : A \wedge B, \ q : (A \to C) \vee (B \to D))$$

The initial problem can be refined by the data type construction rule on page 22.

$$\frac{\Delta, \Gamma \vdash ? : C \vee D}{\left\{ \begin{array}{l} \Delta_l \equiv \Delta(\Gamma \vdash ?_C : C), \Gamma \vdash ? \leadsto \texttt{inl}\ ?_C : C \vee D \\ \Delta_r \equiv \Delta(\Gamma \vdash ?_D : D), \Gamma \vdash ? \leadsto \texttt{inr}\ ?_D : C \vee D \end{array} \right.}$$

We get two refinements, representing two paths in the search. We will follow both paths. Each refinement has one new meta variable, $?_C$ and $?_D$. $?_C$ can be refined by the following elimination:

$$\dfrac{\Delta_l, \Gamma \vdash ?_C : C \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash \ q : (A \to C) \vee (B \to D)}{\Delta_l, \Gamma \vdash \ q : (A \to C) \vee (B \to D)}}{\Delta_l, \Gamma, (q = \texttt{inl}\ x) \vdash \ x : A \to C}}{\Delta_l(\Gamma \vdash ?_A : A), \Gamma, (q = \texttt{inl}\ x) \vdash \ x\ ?_A : C}}{\Delta_l' \equiv \Delta_l(\Gamma \vdash ?_A : A), \Gamma, \sigma_l \equiv (q = \texttt{inl}\ x) \vdash ?_C \rightsquigarrow x\ ?_A : C}$$

Here we used elimination refinement rule together with a chain of elimination judgements, which started in the trivial judgement that the reference to q is an elimination. After this, data type elimination is used to extract the contents of the first construction. This adds the corresponding value constraint. The last judgement is a result of the application rule, which extends the meta environment by a new meta variable representing the argument of the application.
The same thing is done for $?_D$, except that the second part of q is extracted instead.

$$\dfrac{\Delta_r, \Gamma \vdash ?_D : D \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash \ q : (A \to C) \vee (B \to D)}{\Delta_r, \Gamma \vdash \ q : (A \to C) \vee (B \to D)}}{\Delta_r, \Gamma, (q = \texttt{inr}\ y) \vdash \ y : B \to D}}{\Delta_r(\Gamma \vdash ?_B : B), \Gamma, (q = \texttt{inr}\ y) \vdash \ y\ ?_B : D}}{\Delta_r' \equiv \Delta_r(\Gamma \vdash ?_B : B), \Gamma, \sigma_r \equiv (q = \texttt{inr}\ y) \vdash ?_D \rightsquigarrow y\ ?_B : D}$$

Again, each of the two refinements contains one subproblem, $?_A$ and $?_B$ respectively. To solve these, the left and right components of p can be used. This is done by the projection elimination rule. For $?_A$ we get

$$\dfrac{\Delta_l', \Gamma, \sigma_l \vdash ?_A : A \qquad \dfrac{\dfrac{\Gamma \vdash \ p : A \wedge B}{\Delta_l', \Gamma, \sigma_l \vdash \ p : A \wedge B}}{\Delta_l', \Gamma, \sigma_l \vdash \ p.x : A}}{\Delta_l', \Gamma, \sigma_l \vdash ?_A \rightsquigarrow p.x : A}$$

and for $?_B$

$$\dfrac{\Delta_r', \Gamma, \sigma_r \vdash ?_B : B \qquad \dfrac{\dfrac{\Gamma \vdash \ p : A \wedge B}{\Delta_r', \Gamma, \sigma_r \vdash \ p : A \wedge B}}{\Delta_r', \Gamma, \sigma_r \vdash \ p.y : B}}{\Delta_r', \Gamma, \sigma_r \vdash ?_B \rightsquigarrow p.y : B}$$

Neither of these refinements contains any subproblem. We have thus reached the bottom of the search. There are two solutions to the problem, both of which have a value constraint.

$$(q = \texttt{inl}\ x) \vdash\ (? \rightsquigarrow \texttt{inl}\ ?_C, ?_C \rightsquigarrow x\ ?_A, ?_A \rightsquigarrow p.x) \Rightarrow ? \rightsquigarrow \texttt{inl}\ (x\ p.x) \qquad (S1)$$

$$(q = \texttt{inr}\ y) \vdash\ (? \rightsquigarrow \texttt{inr}\ ?_D, ?_D \rightsquigarrow y\ ?_B, ?_B \rightsquigarrow p.y) \Rightarrow ? \rightsquigarrow \texttt{inr}\ (y\ p.y) \qquad (S2)$$

A notation for solutions has not been defined, but the meaning should be clear. A solution consists of a set of constraints and a meta variable assigned to a proof term. The two solutions have value constraints. Thus, none of them does on its own represent a valid top level solution. But since the constraints are complementary, we can combine the two solutions to form one that is valid. The variable q is the scrutinee of the value constraints, (S1) belongs to the `inl` branch and (S2) to the `inr` branch. A complete solution to the problem is thus

$$\textbf{case}\ q\ \{\texttt{inl}\ x \rightarrow \texttt{inl}\ (x\ p.x);\ \texttt{inr}\ y \rightarrow \texttt{inr}\ (y\ p.y);\}$$

3.3 Case analysis and induction

Case analysis, normally combined with induction, is an important component of most proofs. As seen in subsection 3.2 case expressions appear when combining conditional solutions. This however only deals with *proof* object scrutinees. There is a special refinement rule which introduces case analysis for *parameter* object scrutinees. It corresponds to the `induction` and `destruct` tactics of Coq. The proof/parameter classification was described in subsection 2.2. Parameter objects are dealt with quite differently since the case expression is introduced as a refinement before the solution to any of its branches is searched for. It may seem better to treat both kinds equally. Apart from the higher consistency, the proof search would completely enjoy the trait that no case branching occurs unless called for, which was the main motivation for how proof scrutinees are handled. But, when it comes to parameter scrutinees, this approach entails a rather complicated and time consuming machinery of the type comparison. Since the value of parameters affect the type of some proof object in the context or the target type, a lot of different combinations of assumptions on the present parameters must be tested when comparing types. The cost for doing this seems to high compared to the advantages. The equal treatment approach was the basis of the initial experiments, but it was then abandoned.

The case refinement rule must have some source which provides it with the collection of scrutinees to try. One place to look is in the environment. All objects that are parameters, are of data type and have not earlier been analyzed could qualify. You may object that a lot of alternatives would arise and that the choice is not really considering the parameter's role in the proof types. Another way to generate scrutinees is to look at the target type and the environment elements' types and see which ones halt the computation, i.e. which scrutinees cannot be reduced to a construction and thus stop the type from being simplified any further. This seems somewhat more cunning than the first alternative, because it actually looks at the structure of the types involved. Another important advantage is that this method not only presents scrutinees that are single parameters, but also compound expressions, such as function applications where the argument is unknown. Case analysis on compound scrutinees is sometimes necessary. There is a special version of the case refinement rule dedicated to this. It is described further on in this subsection.

Experiments have indicated that a combination of these two methods is needed. The second does produce most relevant scrutinees, including compound ones. But it sometimes leaves out parameters that have a passive role in the types but still needs to be analyzed at that certain point in the proof.

For each candidate scrutinee which is a parameter, the refinement method generates a case expression with branches for each constructor of the data type. It also adds a locally defined function just outside the case expression. This is intended for recursive calls from within the branches or, in other words, for reference to the induction hypotheses of the problem. As arguments, the function of course takes the parameter itself, but to be flexible enough it must also take all objects in the environment whose type depends on the parameter. Furthermore, any other parameter present in those objects' types should also be included and so forth. It may happen that no reference is made to the locally defined function, especially when the data type is not recursive. In that case the definition is removed before the solution is presented.

In the subproblems of the refinement, the target type and variable environment are specialized for each branch by substituting the parameter with the correct construction and then re-simplifying the types. Also, the locally defined function is added to the environment as well as its arguments and any arguments of the construction. The function is however treated in a special way since we want to avoid bad recursive behaviour. The program currently restricts the use of the function in a way which only allows structural recursion, which guarantees program termination. This limitation is imposed by indicating that when the elimination judgements are generated for the function, it is not itself used as the starting point. Instead, the function applied to any part of the parameter serves a starting points. As an example, let us look at a parameter, t, that is a binary tree:

$$t \;:\; BT\ X$$
$$BT\ (X : Set) \;:\; * \;=\; \textbf{data}\ \texttt{lf}\ (x : X) \mid \texttt{nd}\ (t_1 : BT\ X)\ (t_2 : BT\ X)$$

Let the locally defined function be called f. The starting points for the elimination judgements would then be none in the `lf` branch which is the base case. In the `nd` branch on the other hand there will be two, $f\ t_1$ and $f\ t_2$, corresponding to the two induction hypotheses of the inductive step.

To summarize, the rule for case refinement looks like this:

$$\frac{\Delta, \Gamma, \sigma \vdash ?\ :\ T \qquad \Gamma \vdash\ y : Y \qquad Y\ \overset{\mathrm{n}}{\twoheadrightarrow}\ \mathcal{D}}{\Delta', \Gamma, \sigma \vdash ? \rightsquigarrow \left(\begin{array}{l} \textbf{let}\ f\ (y' : Y)\ (\vec{z'} : \vec{Z}) : T = \\ \quad \textbf{case}\ y'\ \{ \\ \qquad \texttt{c}_1\ x_{1,1}\ \ldots\ x_{1,n_1} \rightarrow ?_1; \\ \qquad \vdots \\ \qquad \texttt{c}_m\ x_{m,1}\ \ldots\ x_{m,n_m} \rightarrow ?_m \\ \quad \} \\ \textbf{in}\ \ f\ y\ \vec{z} \end{array}\right) \ :\ T} \ \text{(case refinement)}$$

The side conditions of the rule are that y, the main parameter, is of type Y and that it in turn normalizes to a data type. The declarations $\vec{z} : \vec{Z}$ is the collection of secondary parameters and dependent objects. They must be included as arguments in order to vary with the main parameter. The meta environment is extended by one new meta variable for each branch of the case expression.

$$\Delta' \;\equiv\; \Delta(\Gamma'_1 \vdash ?_1 : T'_1, \ldots, \Gamma'_m \vdash ?_m : T'_m)$$

The variable environments and target types of these meta variable are the following.

$$\Gamma'_j \;\equiv\; \Gamma((f : (y' : Y) \to \vec{Z} \to T)\ (y' : Y)\ (\vec{z'} : \vec{Z}'_j)\ (x_{j,1} : X_{j,1}) \ldots (x_{j,n_j} : X_{j,n_j}))$$

$$T'_j \;\equiv\; T[\texttt{c}_j\ x_{j,1}\ \ldots\ x_{j,n_j}/y]$$
$$\vec{Z}'_j \;\equiv\; \vec{Z}[\texttt{c}_j\ x_{j,1}\ \ldots\ x_{j,n_j}/y]$$

For proofs where some type refers to a function which includes a case expression with a compound scrutinee, it may be necessary to do case analysis for this compound expression. When the type-checker encounters a case expression with a parameter as scrutinee, it uses substitution to specialize the types in the branches. For compound expressions it is unable to do this since the type-checker does not do substitution of subexpressions. The type-checker must in these cases be supported by replacing the compound expression with a new variable. Every occurrence of the compound expression is then replaced by this new variable and the type-checker can thus confirm the correctness.
For compound scrutinees, the case refinement method uses a special variation of the rule above. The side condition $\Gamma \vdash \; y : Y$ is replaced by $\Gamma \vdash \; M : Y$ where M is an arbitrary term and y' serves as the new parameter replacing M. Also, instead of substituting the variable y for the different constructions, every occurrence of the subexpression M is replaced. There is one more difference, namely that whenever possible, a proof of $M == y'$ is passed as an argument to the locally defined function. The proof is equally provided by using reflexivity. The presence of such a proof has seemed useful in experiments, but the question whether it is necessary remains.

3.4 Generalization

Generalization is vital to proving. It does not merely lead to a more general result. In some proofs it is necessary because without generalization you simply do not get any further. Experiments with a method for introducing generalizations have been done, but it has very limited ability. Automatic generalization is a field of considerable interest at the time.
The method has two simple procedures to suggest generalizations. They are both only applied on target types which represent equalities. The first looks for several occurrences of the same subexpression. If at least two occurrences are found, a generalization is constructed by replacing that subexpression with a new parameter of the correct type. Here is an example.

$$x + b == b + x \quad < \quad f\ a + b == b + f\ a$$

The '<' indicates that the left type is more general than the right one and x is the new parameter. The second procedure looks for two or more occurrences of the same parameter on both sides of the equality. Upon finding this, the parameter is separated into two or more new ones as in this example:

$$a_0 + \mathtt{s}\ a_1 == \mathtt{s}\ (a_0 + a_1) \quad < \quad a + \mathtt{s}\ a == \mathtt{s}\ (a + a)$$

Here a_0 and a_1 are new parameters. The program generates a generalization for each unique assignment distribution of the new parameters.

The rule for generalization refinements has the following general form:

$$\frac{\Delta, \Gamma, \sigma \vdash ?\ :\ T \qquad \begin{array}{c} \Gamma \vdash\ \vec{X} : * \\ \Gamma(\vec{x} : \vec{X}) \vdash\ T' : * \\ \Gamma \vdash\ \vec{M} : \vec{X} \\ \Delta, \rho, \Gamma \vdash\ T\ \stackrel{s}{=}\ T'[\vec{M}/\vec{x}] \end{array}}{\Delta', \rho, \Gamma, \sigma \vdash ? \leadsto \left(\begin{array}{l} \textbf{let}\ g\ (\vec{x} : \vec{X})\ \ : T' = ?_g \\ \textbf{in}\ \ g\ \vec{M} \end{array} \right)\ :\ T} \left(\begin{array}{c} \text{generalization} \\ \text{refinement} \end{array} \right)$$

where

$$\Delta' \ \equiv\ \Delta(\Gamma(\vec{x} : \vec{X}) \vdash ?_g : T')$$

The variables $\vec{x}$ are the new parameters introduced by the generalization procedure and T' is the generalization of T. The rule has four side conditions. The first presents the types of the new parameters, $\vec{X}$. The second says that T' is a valid type in Γ extended by the new parameters. The third introduces the set of terms that will be the arguments of the function application. Finally, the last one ensures that by replacing $\vec{x}$ by $\vec{M}$ in T' the two types become equal. The rule does not actually require that T' is more general than T. But it is intended to be used together with the two generalization procedures described above. The same rule can be used if more procedures are added. The fact that T' is a generalization is reflected in the terms $\vec{M}$. For the currently implemented procedures the arguments may either be compound expressions or the same parameter appearing more than once. If other kinds of generalizations were also included, different kinds of terms would arise, e.g. constant values.

3.5 Parameter assumption

The rule described in subsection 3.3 handled scrutinees which are variables. Parameter meta variables can however also occur as scrutinees. Case analysis can of course not be done for meta variables. They can have only one value within the same solution. But sometimes we need to make assumptions on the form of data type meta variables to find a solution. The resulting refinements represent wild guesses, so they can be used only for trivial proofs.
The rule generates one refinement for each constructor of the data type and

looks like this:

$$\frac{\Delta, \Gamma, \sigma \vdash ? \ : \ T \quad \Delta, \Gamma_p \vdash ?_p : Y \quad \Gamma_p \subseteq \Gamma \quad \Gamma \vdash \ Y \overset{\text{n}}{\twoheadrightarrow} \mathcal{D}}{\Delta'_j, (?_p = \mathsf{c}_j \ ?_1 \ \ldots \ ?_{n_j}), \Gamma, \sigma \vdash ? \rightsquigarrow ?_s \ : \ T} \left(\begin{array}{c} \text{parameter} \\ \text{assump. ref.} \end{array} \right)$$

The meta variable $?_p$ is the parameter meta variable upon which assumptions will be made. The first side condition introduces the variable environment and type of $?_p$. The second requires that Γ is an extension of Γ_p. It is also implicitly assumed that no name clashes were imposed when going from Γ_p to Γ. With these requirements we can conclude that Y has the same meaning in Γ. The requirements are fulfilled in the implementation as a consequence of how the refinement rules are used in the search algorithm. The last side condition says that Y should normalize to a data type.
The rule produces one refinement for each constructor of the data type. The meta environment is extended by the meta variables appearing in the construction and by that of the new subproblem, $?_s$.

$$\Delta'_j \ \equiv \ \Delta(\Gamma \vdash ?_1 : X'_{j,1}, \ldots, \Gamma \vdash ?_{n_j} : X'_{j,n_j}, \Gamma \vdash ?_s : T)$$

The types of the new parameter meta variables are defined as follows.

$$X'_{jk} \ \equiv \ X_{jk}[?_{k-1}/x_{j,k-1}] \ldots [?_2/x_{j,2}][?_1/x_{j,1}]$$

This rule is typically needed in the base case of inductive existence proofs. In the base case it is often the case that no proof object can be invoked. Thus a witness cannot be found as a side effect of type unification. Instead it must be constructed and the program has no ability to do this by analyzing the target type. It therefore have to do guesses using the meta variable assumption rule.

3.6 Equalities

When the target type is an equality, a special rule replaces the ordinary one for elimination refinements. It is similar, but designed to respect the specific properties of equalities with some intelligence. The utilization of reflexivity, symmetry, transitivity and substitutivity for equalities is hard-coded into the program. Also, the way equalities are represented is hard-coded. It adopts the representation of the Agda/Alfa standard library written by Michael Hedberg, namely using ‘Datoids’ and ‘Setoids’. A short presentation of these definitions can be found in section 5. The program has no ability to recognize any other relation that fulfills the requirements for an equivalence relation and then treat it accordingly.

The general form of target type which causes the problem to be redirected to this method is the following.

$$M \stackrel{X}{==} N$$

Here, X is the Setoid and $|X|$ denotes the actual set, so $M, N : |X|$. Reflexivity actually works fine without special treatment. M and N are compared using unification just as if they were types themselves. If M and N can be unified a refinement containing a call to $X.\mathit{ref}$ is generated. Any meta variable bindings of the unification are added as constraints.
For symmetry and transitivity, a systematic use can reduce search space, since these otherwise tend to make the search go round in circles. Substitutivity cannot at all be handled by the standard method for elimination. All these three properties are addressed in the same procedure. For both M and N this procedure looks at the entire expression as well as every subexpression for which the Datoid can be concluded. The Datoid must be known in order to use substitutivity. For each subexpression, E, it searches in the environment for an object whose type matches '$E \stackrel{Y}{==} ?_1$' or '$?_1 \stackrel{Y}{==} E$', where Y is the Setoid of E. A refinement is generated for each matching object. Symmetry is invoked if needed. Transitivity is inserted if the equality was not closed and substitutivity is used if E is indeed a subexpression and not the entire M or N.

In addition, problems of equality carry some extra information to avoid reasoning in circles, but more could be done to narrow down search space. Not only circular behaviour is a problem, but also that with a couple of relevant lemmas in the context there are often quite a few valid refinements since a substitution can be made for any subexpression. This results in a large search space even if returning to prior states is avoided. And it is difficult to conclude which refinements would not lead to a solution, since they all represent equivalent problems. The search in equality chains has a horizontal character. A suitable way to deal with this is to enforce some systematic notion of direction in the lemmas used for transformation. This is often known as *rewriting rules* and is successfully implemented in for example the Coq system[7]. In our experiments nothing similar has been elaborated. Allowing unlimited equality chains is therefore often very time consuming and seldom leads to a happy end. To amend this, the program restricts the length of equality chains to a very modest number; one or two intermediate links. This of course in theory impedes some solutions from being found, but in practice most of them would have taken too much time anyway.

As mentioned, the program seeks to build an equality chain both from the left and from the right hand side. But, since transitivity is never used unless there is an object modifying either the left or the right hand side, no more than one new equality can be established. At each stage while exploring a chain, other refinements can be used to construct a proof of the current equality. But the program will never invent an intermediate value, i.e. one which is not proclaimed by an existing proof object. If transitivity was allowed without fixing the intermediate value, such a thing could in principle occur, but would in practice be tough since the program does not have the ability to come up with intelligent guesses.

4 Further details

The previous sections presented the refinement rules that the implementation is based on. Each refinement is only one step in the proof search. This section gives a brief description of the main search algorithm. Many other kinds of search algorithms could be used without replacing the refinements as the fundamental concept of the proof search. What characterizes the one of the implementation is more or less that it is simple. It is certainly possible to make a more advanced algorithm with the ability to solve larger problems. One flaw of the current algorithm that seems to be essential is that it is depth-first.

4.1 The search algorithm

Except for problems and refinements, which are already familiar, the algorithm relies on entities called solutions. A solution consists of information to construct the proof term together with a set of meta variables constraints as well as a set of value constraints. The constraints must be fulfilled for the solution to be valid.
When trying to solve the given problem along a refinement it must be clear which the subproblems are. Therefore we modify all refinement rules so that the generated refinement also includes a list enumerating the new meta variables introduced by it.

The search is carried out by a recursive function that takes a problem and returns a list of solutions. It does the following. First, all the refinement rules are applied to the problem. This results in a list of possible refinements. This list is then analyzed to see if several refinements are essentially equivalent. If so, only one copy is kept and the rest are discarded. The refinement methods are designed not to generate the same refinement more than once, but in some situations it is difficult to avoid, e.g. when the proof objects in the environment exert some symmetry. Apart from discarding duplicates, the refinements are also sorted so that those, which are best in some sense, come first. These issues is further described in subsection 4.2.

The program then starts off with an empty list of solutions and goes through the refinements one by one. Every refinement produces zero or more solutions and these are incrementally included in the growing list of solutions. The subfunction that includes a new solution into the list compares it to the ones that are already present. The new solution can turn out to be inferior to another, and is therefore discarded. It can also be superior to another and in that case that other solution is taken away. After this is settled, the subfunction also tries to combine the new solution with others in case it carries some value constraints. More about this can be found in subsection 4.3.

Each refinement contains a number of new meta variables. These are divided into proofs and parameters. As mentioned, only proofs are treated as subproblems. The value of a parameter is decided as a side effect when searching for the subproblems whose types depend on the parameter. The algorithm looks at one subproblem at a time. It is thus depth-first. A breadth-first approach has probably some advantages. This is discussed in section 6. The subproblems are however not independent of each other. There are two things which connects them. First, the subproblems can depend on the same parameter meta variable. Since solutions to the problems would be part of the same proof, the value must of course be the same for all sub solutions. Secondly, the situation is the same for value constraints. If a data type object is assumed to have a certain form in one sub solution, this must not be in conflict with any constraint of the others. This is why refinements and solutions carry information about both meta variable constraints and value constraints. Subsection 4.4 describes how the algorithm takes these issues into account when searching for solutions to the subproblems of a refinement.

The use of the refinement rules is restricted in some ways in order to narrow down search space. Abstraction refinement is always applied as far as possible before any other refinement is considered. This may lead to a solution which in the end contains abstractions and function applications that could be mutually cancelled, but it is an easy job to do such cancellations as a last step of the search. The advantage, on the other hand, is that we reduce search space without excluding any candidates. Always doing full abstraction means that when the target type is compared to the objects in the environment, it is never a function type. As a consequence, it is useless to consider function type elimination judgements. Therefore those judgements are just used as intermediate steps leading to non-function types.

The algorithm uses the parameter/proof classification to discern which meta variable to treat as subproblems. This method is not the obvious way to do it. The objective is to avoid situations where a solution is searched for a meta variable that is not a proof and so is quite arbitrary, whereas a search for the subproblem which depend on it would easily bind it to an appropriate value. As an alternative to this classification, you could for example perform the proof search beginning from the last subproblem and moving forward. That way binding would occur before the problematic meta variables were visited and the program would know which ones not to search for. There are however occasions when a meta variable should not be bound during the search for a solution to any of the succeeding subproblems of the same refinement. Instead, binding should occur at some other place in the search hierarchy.

A reasonable thought is that administrating the value constraints is rather expensive and that, if there is a data type proof object among the premises, it should probably be used to get a solution. But if you include parts of the global environment, then some objects may be totally irrelevant to the current problem. Also, if case-expressions were introduced in an a priori way, the search for the different branches would do partially the same thing. If we for instance have $A \vee B$ in the environment and a case-expression for this object was generated as a refinement, then the search for the solution of the first branch would look at some $a : A$ and everything else while for the second branch it would be some $b : B$ and the same everything else. The form of a data type proof object does not change the problem, i.e. target type and environment. The situation is different for data type *parameter* objects. They appear in some type which thus depends on their form. Parameter objects are due to this important difference not handled in this way, as we have seen in subsection 3.3.

4.2 Comparing and discarding refinements

Before the refinements are searched, they are inspected by comparing them pairwise. If a refinement is definitely worse than another it should be removed. Moreover, if a refinement is cheaper to test than another it is moved towards the head of the list. To determine the relation between refinements, three things are taken into account, which are all represented by partial ordering. First, the target types of the subproblems are compared. If the set of subproblem target types of the first refinement is identical to that of the second, the subproblems are deemed equal. If the set of target types of one of the refinements is strictly included in that of the other, the second refinement is definitely no easier to solve and hence can be considered worse when it comes to subproblems. It the sets are not at all related by inclusion, they are considered to be unordered. The equality of target types is settled using the same operation as for comparing a target type to the elimination judgements, namely '$\stackrel{s}{=}$' . The only difference is that meta variables that are external to the refinements are not treated as arbitrary. Meta variable constraints resulting from the unification are taken care of so that inequality is not induced by parameter meta variables merely having different indices in the two refinements. Apart from the refinements which have no subproblems, and which are definitely better in this sense, inequalities are rare, since they occur when one proposition demands the same premises as another plus something extra. This is an unnatural situation as the more demanding proposition is inferior and should be removed. The comparison does not take apart target types to see that for instance equality could be claimed in the following situation.

$$(?_1 \ : \ A, \ ?_2 \ : \ B) \quad \text{vs.} \quad (?_1 \ : \ A \wedge B)$$

If a subproblem introduces extra environment elements, it is considered unordered to any other. The program could in this case go further and maybe establish a non-strict inequality relation, but those cases also seem very hypothetical and therefore no effort has been made to include it. This comparison is effective mostly when there are refinements with no subproblems and this could of course be checked much easier. But in some cases, e.g. for propositions that have some symmetry and can be invoked in several ways but with equal results, the equivalence can by this comparison by revealed.

Secondly, the constraints of the refinements are compared. The partial order relation is decided in the same way as for the target types, i.e. by determining if one set of constraints is included in the other. The third comparison only comes into effect if the two latter deem the refinements to be equally good. It looks at the length of the value. Shorter proofs terms are considered better. To take example, a frequently occurring situation is the following.

$$\Gamma(x : \bot) \vdash ?_i \ : \ \bot$$

By comparing proof term lengths, 'x' will be favored at the expense of '**case** x {}'.

4.3 Comparing and combining solutions

Refinements are generated on the way down in the recursive search. When the end of a search branch is reached, i.e. refinements with no subproblems are encountered, the results are passed upwards. A refinement without subproblems is alone a solution to the current problem. Refinements which have subproblems form a complete solution if solutions are found to each of them. A solution mainly contain a tree structure of meta variable bindings which reflect the proof search that lead to it and which by successive substitutions build up a proof term, free of meta variables. The solution carry the meta variable and value constraints which must hold for the refinements constituting it to be valid. If constraints are present, it is still called a solution in

this context, although it is of course of no use at the top level of the proof search.
Each solution encountered is compared to the ones already found. The comparison is similar to that of refinements. If the constraints of one solution are included in the those of another, the latter solution is removed. If the constraints are the same, only the one with the shorter proof term is kept.

When comparing a new solution to the existing ones, the program also looks for collections of complementary value constraints. If there is one solution for each different construction of a data type, a new solution is constructed. This consists of a case expression with the associated solutions as branches. The new solution inherits the constraints of the constituents unionwise, minus the one constraint addressed at the fusion. The other constraints of the refinements must clearly not be in conflict, i.e. they must not assume different values for the same object, scrutinee or meta variable. Depending on how the components' constraints overlap, the new refinement may or may not be superior to some of the constituting refinements. In any case, refinements which were created by combination are never allowed to disqualify uncombined ones. This is because it may be the case that the combination of solutions should follow some other path, although it is not known at this point in the search. This may for example happen when there are redundancies among the premises of a proposition. If a solution with constraints is found, the search of course continues, but it does not mean that only complementary solutions are looked for. If a solution without constraints is then encountered before all of complementary solutions are found, the search of the current subproblem immediately stops.

All solutions to a problem are passed upwards to the parent problem, even those with value constraints, since a successful combination may not be possible at the level where the conditional solution was found, but rather higher up in the hierarchy. At one point, however, solutions with value constraints must be removed. It is when the declaration of some parameter appearing in the scrutinee is passed. Case branching must of course take place below this point.
Meta variables are also introduced at some specific refinement, just as ordinary variables. When a solution with meta variable constraints passes the point where any of the bound meta variables were introduced, it should however not be disqualified. On the contrary, what happens is that since the meta variable becomes local, its value is no longer a constraint of the solution. So, at this point we simply remove the meta variable constraint. Any solution that stays until the top level of the search will therefore contain

neither meta variable nor value constraints.

4.4 Searching for solutions to coexisting subproblems

When we encounter a refinement having more than one subproblem, we cannot just first search for a solution to the first one and then a solution to the second one etc. The presence of constraints has the effect that we cannot settle for the first solution encountered, unless it has no constraint. We must make sure that the solutions to the subproblems can coexist in the sense that there is a set of constraints making them all valid. This may seem to be a big disadvantage of the approach of dealing with data type elimination in terms of constraints, but the situation would be the same without this, since there would still be meta variable constraints. These are difficult to get rid of in the realm of dependent type theory.

You can deal with this issue in several ways. The implementation of this work handles it in a rather simplistic way. Experiments have indicated that for larger problems, another approach would probably be more successful. This is described further on. The program currently does the following. All solutions for the first subproblem are produced. Before moving on to the next subproblem inferior solutions are discarded in the way described in the previous subsection. Then the second subproblem is searched once for every solution of the first. The constraints of the related solution are imposed before searching. This means that appearing meta variables are substituted by their bound values in the relevant types. Also, the value constraints are passed on in the search to prevent conflicting assumptions from being made. The resulting solutions of each such search is combined with the prior constraints. Then all the solutions are concatenated and the program moves on to the next subproblem. This way simultaneous validity is ascertained. If there is no solution to a subproblem given a certain set of constraints, that particular path will die out, reflecting the fact that the associated combination of solutions to the subproblems does not lead to a solution. Moreover, since only one solution for each set of constraints is kept, the search does not become more complex simply because there are several equivalent solutions to a subproblem.

This approach seems plausible in many cases. But for some proofs the search gets very slow and we suspect that a different method could then be more efficient. The program sometimes gets in trouble for problems which include proving existence. If there are several ways to construct a witness and this can be done recursively, a large amount of witnesses can be generated. Take the following type as an example.

$$\exists\ X\ (\lambda(x : X) \rightarrow (P\ x \wedge Q\ x))$$

For a real problem, it should be non-trivial to find a witness that fulfills both $P\ x$ and $Q\ x$. At the same time it could be very easy to find a witness fulfilling only $P\ x$. And this is what the program does at first. If it after that has several hundred solutions, each associated with a unique witness, the search for the second subproblem will be carried out hundreds of times. This situation advocates the conclusion that it might be better to search for the succeeding subproblem only once and without any conditions imposed. Afterwards, the new solutions would be compared with the previous ones to see which combinations can be done, if any. This however disregards the solutions to the prior subproblems when searching for the current one, but it may still be cheaper. An even better method would probably be to do the search breadth-first instead of depth-first. This way search space is possibly narrowed down more quickly, since portions of all subproblems are taken into account in parallel.

4.5 Speed issues and search restrictions

The greater part of the work has been devoted to make the system in principal capable of solving a fair amount of problems. Different schemes have been designed to deal with different situations. These are incorporated in the refinement rules. Nevertheless, some time has been spent to investigate speed issues. As a tool for supporting interactive solving, it is desirable that an answer is reported without much delay. The choice to work with the notion of simplified types and the preparations of the global environment to speed up computations are examples of efforts aimed in this direction.

To control the time consumption, there are a number of parameters of the algorithm which will be described in the following. In order to make the program terminate in the first place, a preset maximum search depth in the refinement hierarchy is used. Since the time it takes to complete a search to a certain depth may differ a lot depending on the problem, a counter which roughly mimics the elapsed time is also used. It simply counts the total number of refinements produced. A third parameter restricts the number of solutions to the same problem that is stored. If this number is exceeded the tail of the list is expunged. The idea is that the correct solutions hopefully is found early. To support this a few simple things are done to sort the generated refinements putting the most promising ones first. The parameter is added to tackle the kind of problem described in the previous subsection. Ideally, there would be only one parameter of course. If a proof is not found you would then just increase this number. Several parameters make it possible to choose different combinations guided by some knowledge about the problem, which is somewhat dissatisfactory.

One difficulty is case refinements. If there are a few parameters in the problem, a lot of combinations of case expressions can be constructed. Case refinements are also costly, because the target type and variable environmet must be specialized for each branch. To reduce search space, there is an optional restriction which enforces case refinements to appear right away. This means that for refinements other than case refinements, all scrutinees which were possible at the current stage are disabled below. This however does make some proof terms a bit longer than would otherwise be the case.
The fact that case analysis can be recursively and infinitely performed also poses a problem. The program tries to deal with this by doing an analysis of the problems' types with respect to recursive depth of the referenced functions. The idea is that if only depth-one recursion appears in the types, only depth one-recursion is needed in the proof, and so forth. It is not claimed that this is always correct or that no proofs this way are overlooked.
In spite of these amendments, a large portion of the time is often spent on finding the right combination of case expressions when the number of possible scrutinees are about five or more. In an earlier version of the program, case analysis for parameter objects was treated in the same way as for proof objects, to the extent possible. This approach was discussed in subsection 3.2. There are examples of problems which are solved faster by the old version, probably because the current, relatively blind search for the correct combination was partly avoided.
One might argue that one should look at the target type to decide which parameter to do case analysis for. If you normalize the type, you will end up with a case expression. The scrutinee of this is stopping the normalization from proceeding. It seems natural to choose this parameter. This is probably often correct, so you should maybe favor that particular choice. But one cannot disregard the other ones, because in some cases it is preferable or imperative to pick another parameter.

Finally, a few words about equality chains. The inability of the program to deal with these in a good way has already been described in section 3.6. When the problem is an equality where the LHS or RHS are rather complex and there are a few fairly general proof objects in the environment, there tend to be many equality chain refinements. To limit the time cost of the highly unguided search in equality chains, the use of these refinements is limited. The program does consider them, but if the chain grows beyond one or two intermediate forms, it is done very reluctantly. This way proofs involving no or short equality chains are found quicker.

5 Examples

In this section a number of examples will be presented. They are intended to illustrate how the tool works, as well as to give a sample of what kind of problems it can solve. There will first be a couple of single proposition problems, followed by a case study where the tool solves a number of lemmas which prove the correctness of the quicksort algorithm.

The examples depend on some definitions in the standard library of Agda/Alfa. We will give a brief description of the components which are needed. First there are a few logical symbols adding to the ones already seen. We have the trivial set, the empty set, negation and equivalence.

$$\begin{aligned}
&\top : * = \textbf{data}\ \texttt{tt} \\
&\bot : * = \textbf{data} \\
&\neg\ (A : *) : * = A \to \bot \\
&(\leftrightarrow)\ (A, B : *) : * = (A \to B) \wedge (B \to A)
\end{aligned}$$

The use of Setoids and Datoids is hard-coded in the equalities refinement method. These were mentioned in subsection 3.6. If X is a Setoid or Datoid, $|X|$ will denote the actual set. The Setoid establishes reflexivity, symmetry and transitivity. For a Setoid, X, we have

$$\begin{aligned}
&X.\mathit{ref}\ :\ (x : |X|) \to (x \stackrel{X}{==} x) \\
&X.\mathit{sym}\ :\ (x_1, x_2 : |X|) \to (x_1 \stackrel{X}{==} x_2) \to (x_2 \stackrel{X}{==} x_1) \\
&X.\mathit{tran}\ :\ (x_1, x_2, x_3 : |X|) \to (x_1 \stackrel{X}{==} x_2) \to (x_2 \stackrel{X}{==} x_3) \to (x_1 \stackrel{X}{==} x_3)
\end{aligned}$$

Substitutivity is provided by the function *respEq*. To use this, the Datoid of the subexpression must be specified. In order to allow for all kinds of equality rewritings, the Datoid of a set should always be present. A Datoid can mechanically be transformed into a Setoid, but not *vice versa*. The transformation is done by the function *D2S*.

$$\mathit{D2S}\ :\ \mathit{Datoid} \to \mathit{Setoid}$$

The type of *respEq* is the following:

$$\begin{aligned}
&\mathit{respEq}\ (X : \mathit{Datoid})\ (Y : \mathit{Setoid})\ (f : |X| \to |Y|)\ (x_1, x_2 : |X|) \\
&\quad :\ (x_1 \stackrel{(\mathit{D2S}\ X)}{==} x_2) \to (f\ x_1 \stackrel{Y}{==} f\ x_2)
\end{aligned}$$

The Datoid/Setoid issues are merely technicalities of the system and can be disregarded by readers who are not interested in details of the implementation.

In previous sections, '*Nat*' has denoted the set of natural numbers. Now it will instead refer to the Setoid. Thus, the set of natural numbers will be written '$|Nat|$'. We will also use the Setoids of booleans and lists, *Bool* and *List*. The corresponding Datoids will be designated by 'Nat_D' etc. The sets are defined as follows.

$$
\begin{array}{l}
|Nat| : * = \textbf{data}\ \mathtt{0} \mid \mathtt{s}\ (n : |Nat|) \\
|Bool| : * = \textbf{data}\ \mathtt{t} \mid \mathtt{f} \\
|List\ (X : Setoid)| : * = \textbf{data}\ \mathtt{[]} \mid (::)\ (x : |X|)\ (xs : |List\ X|)
\end{array}
$$

Infix notation will be used for the constructor '$::$'.
We now turn to the examples.

5.1 Commutativity of addition

As the first example, we look at commutativity for addition of natural numbers.

$$(a,\ b : |Nat|) \rightarrow a + b \overset{Nat}{==} b + a$$

Addition for natural numbers is assumed to be defined as follows

$$
\begin{array}{rcl}
\multicolumn{3}{l}{(+)\ (a, b : |Nat|) : |Nat|} \\
\mathtt{0} + b & = & b \\
\mathtt{s}\ a' + b & = & \mathtt{s}\ (a' + b)
\end{array}
$$

The solution to this problem is presented in a rigorous way to illustrate how refinements are put together to form a complete solution. Side conditions to the refinement rules are omitted. The example mainly includes refinement rules for case analysis and equalities elimination. Our initial problem is

$$\Delta \vdash ? : (a,\ b : |Nat|) \rightarrow a + b \overset{Nat}{==} b + a$$

where

$$\Delta \ \equiv\ (? : (a,\ b : |Nat|) \rightarrow a + b \overset{Nat}{==} b + a)$$

First we introduce λ-abstractions for the arguments. They are both introduced at once in the following refinement.

$$\frac{\Delta \vdash ? : (a,\ b : |Nat|) \rightarrow a + b \overset{Nat}{==} b + a}{\Delta_1 \vdash ? \rightsquigarrow (\lambda(a, b : |Nat|) \rightarrow ?_1) : (a,\ b : |Nat|) \rightarrow a + b \overset{Nat}{==} b + a}$$

where

$$\Delta_1 \ \equiv\ \Delta(\Gamma \vdash ?_1 : a + b \overset{Nat}{==} b + a)$$

$$\Gamma \;\equiv\; (a, b : |Nat|)$$

We proceed by induction on a.

$$\frac{\Delta_1, \Gamma \vdash ?_1 : a + b \overset{Nat}{==} b + a}{\Delta_2, \Gamma \vdash ?_1 \rightsquigarrow \left(\begin{array}{l} \textbf{let } f\ (a, b : |Nat|) : a + b \overset{Nat}{==} b + a = \\ \qquad \textbf{case } a\ \{ \\ \qquad\quad \mathtt{0} \to ?_2; \\ \qquad\quad \mathtt{s}\ a' \to ?_3 \\ \qquad \} \\ \textbf{in}\ \ f\ a\ b \end{array}\right) : a + b \overset{Nat}{==} b + a}$$

where

$$\Delta_2 \;\equiv\; \Delta_1(\Gamma_2 \vdash ?_2 : b \overset{Nat}{==} b + \mathtt{0},\ \Gamma_3 \vdash ?_3 : \mathtt{s}\ (a' + b) \overset{Nat}{==} b + \mathtt{s}\ a')$$

$$\Gamma_2 \;\equiv\; \Gamma(f : (a, b : |Nat|) \to a + b \overset{Nat}{==} b + a,\ a, b : |Nat|)$$

$$\Gamma_3 \;\equiv\; \Gamma(f : (a, b : |Nat|) \to a + b \overset{Nat}{==} b + a,\ a, b, a' : |Nat|)$$

The target types of $?_2$ and $?_3$ was calculated by first substituting a by $\mathtt{0}$ and $\mathtt{s}\ a'$ respectively, and then applying simplification. The notion of simplification used was described in subsection 2.3. For the ground case, $?_2$, we refine with a case analysis on b.

$$\frac{\Delta_2, \Gamma_2 \vdash ?_2 : b \overset{Nat}{==} b + \mathtt{0}}{\Delta_3, \Gamma_2 \vdash ?_2 \rightsquigarrow \left(\begin{array}{l} \textbf{let } f'\ (b : |Nat|) : b \overset{Nat}{==} b + \mathtt{0} = \\ \qquad \textbf{case } b\ \{ \\ \qquad\quad \mathtt{0} \to ?_4; \\ \qquad\quad \mathtt{s}\ b' \to ?_5 \\ \qquad \} \\ \textbf{in}\ \ f'\ b \end{array}\right) : b \overset{Nat}{==} b + \mathtt{0}}$$

where

$$\Delta_3 \;\equiv\; \Delta_2(\Gamma_4 \vdash ?_4 : \mathtt{0} \overset{Nat}{==} \mathtt{0},\ \Gamma_5 \vdash ?_5 : b' \overset{Nat}{==} b' + \mathtt{0})$$

$$\Gamma_4 \;\equiv\; \Gamma_2(f' : (b : |Nat|) \to b \overset{Nat}{==} b + \mathtt{0},\ b : |Nat|)$$

$$\Gamma_5 \;\equiv\; \Gamma_2(f' : (b : |Nat|) \to b \overset{Nat}{==} b + \mathtt{0},\ b, b' : |Nat|)$$

The target type of the ground case normalizes to the data type $\top$. A data type construction refinement thus completes this branch.

$$\frac{\Delta_3, \Gamma_4 \vdash ?_4 : \mathtt{0} \overset{Nat}{==} \mathtt{0}}{\Delta_3, \Gamma_4 \vdash ?_4 \rightsquigarrow \mathtt{tt} : \mathtt{0} \overset{Nat}{==} \mathtt{0}}$$

The target type of the inductive step is precisely the type of the elimination judgement corresponding to the induction hypothesis.

$$\Delta_3, \Gamma_5 \vdash \ f'\ b' : b' \overset{Nat}{==} b' + \mathtt{0}$$

This gives the following elimination refinement:

$$\frac{\Delta_3, \Gamma_5 \vdash ?_5 : b' \overset{Nat}{==} b' + \mathtt{0}}{\Delta_3, \Gamma_5 \vdash ?_5 \rightsquigarrow (f'\ b') : b' \overset{Nat}{==} b' + \mathtt{0}}$$

This concluded the ground case of the outermost case analysis. We now move to the inductive step of this, i.e. the problem of $?_3$. One way to get a short proof is to invoke the elimination rule for equalities. The target type is

$$\mathtt{s}\ (a' + b) \overset{Nat}{==} b + \mathtt{s}\ a'$$

and induction hypothesis is

$$f\ a'\ b : a' + b \overset{Nat}{==} b + a'$$

The induction hypothesis can be used to replace the subexpression $a' + b$ on the left hand side. To rewrite the equality this way we need to use transitivity and substitutivity.

$$\frac{\Delta_2, \Gamma_3 \vdash ?_3 : \mathtt{s}\ (a' + b) \overset{Nat}{==} b + \mathtt{s}\ a'}{\begin{array}{l} \Delta_4, \Gamma_3 \vdash ?_3 \rightsquigarrow \\ \left(\begin{array}{l} \mathit{Nat.tran}\ (\mathtt{s}\ (a'+b))\ (\mathtt{s}\ (b+a'))\ (b + \mathtt{s}\ a') \\ \quad (\mathit{RespEq}\ \mathit{Nat}_D\ \mathit{Nat}\ (\lambda(x : |\mathit{Nat}|) \rightarrow \mathtt{s}\ x)\ (a'+b)\ (b+a')\ (f\ a'\ b)) \\ \quad ?_6 \end{array} \right) \\ \quad : \mathtt{s}\ (a' + b) \overset{Nat}{==} b + \mathtt{s}\ a' \end{array}}$$

where

$$\Delta_4 \ \equiv \ \Delta_2(\Gamma_3 \vdash ?_6 : \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a')$$

To solve $?_6$ we must again do case analysis on b.

$$\frac{\Delta_4, \Gamma_3 \vdash ?_6 : \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a'}{\begin{array}{l} \Delta_5, \Gamma_3 \vdash ?_6 \rightsquigarrow \\ \left(\begin{array}{l} \textbf{let}\ f'\ (b, a' : |Nat|) : \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a' = \\ \qquad \textbf{case}\ b\ \{ \\ \qquad\quad \mathtt{0} \rightarrow ?_7; \\ \qquad\quad \mathtt{s}\ b' \rightarrow ?_8 \\ \qquad \} \\ \textbf{in}\ \ f'\ b\ a' \end{array}\right) \\ \quad : \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a' \end{array}}$$

where

$$\Delta_5 \ \equiv\ \Delta_4(\Gamma_7 \vdash ?_7 : a' \overset{Nat}{==} a',\ \Gamma_8 \vdash ?_8 : \mathtt{s}\ (b' + a') \overset{Nat}{==} b' + \mathtt{s}\ a')$$

$$\Gamma_7 \ \equiv\ \Gamma_3(f' : (b, a' : |Nat|) \rightarrow \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a',\ b, a' : |Nat|)$$

$$\Gamma_8 \ \equiv\ \Gamma_3(f' : (b, a' : |Nat|) \rightarrow \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a',\ b, a', b' : |Nat|)$$

Reflexivity solves the ground case.

$$\frac{\Delta_5, \Gamma_7 \vdash ?_7 : a' \overset{Nat}{==} a'}{\Delta_5, \Gamma_7 \vdash ?_7 \rightsquigarrow (Nat.ref\ a') : a' \overset{Nat}{==} a'}$$

The target type of the inductive step again is simply populated by the induction hypothesis.

$$\frac{\Delta_5, \Gamma_8 \vdash ?_8 : \mathtt{s}\ (b' + a') \overset{Nat}{==} b' + \mathtt{s}\ a'}{\Delta_5, \Gamma_8 \vdash ?_8 \rightsquigarrow (f'\ b'\ a') : \mathtt{s}\ (b' + a') \overset{Nat}{==} b' + \mathtt{s}\ a'}$$

This concludes the proof.

The proof term can now be constructed by successively replacing $?_1$ through $?_8$ in ? with the refined values.

$$
\begin{array}{l}
\lambda(a,\ b : |Nat|) \rightarrow \\
\quad \mathbf{let}\ f\ (a,\ b : |Nat|)\ :\ a + b \overset{Nat}{==} b + a \\
\qquad = \ \mathbf{case}\ a\ \{ \\
\qquad\quad \mathtt{0} \rightarrow \\
\qquad\qquad \mathbf{let}\ f'\ (b : |Nat|)\ :\ b \overset{Nat}{==} b + \mathtt{0} \\
\qquad\qquad\quad = \ \mathbf{case}\ b\ \{ \\
\qquad\qquad\qquad \mathtt{0} \rightarrow \mathtt{tt}; \\
\qquad\qquad\qquad \mathtt{s}\ b' \rightarrow f'\ b'; \} \\
\qquad\qquad \mathbf{in}\ \ f'\ b; \\
\qquad\quad \mathtt{s}\ a' \rightarrow \\
\qquad\qquad Nat.tran\ (\mathtt{s}\ (a' + b))\ (\mathtt{s}\ (b + a'))\ (b + \mathtt{s}\ a') \\
\qquad\qquad\quad (RespEq\ Nat_D\ Nat\ (\lambda(x : |Nat|) \rightarrow \mathtt{s}\ x)\ (a' + b)\ (b + a')\ (f\ a'\ b)) \\
\qquad\qquad\quad (\mathbf{let}\ f'\ (b,\ a' : |Nat|)\ :\ \mathtt{s}\ (b + a') \overset{Nat}{==} b + \mathtt{s}\ a' \\
\qquad\qquad\qquad = \ \mathbf{case}\ b\ \{ \\
\qquad\qquad\qquad\quad \mathtt{0} \rightarrow Nat.ref\ a'; \\
\qquad\qquad\qquad\quad \mathtt{s}\ b' \rightarrow f'\ b'\ a'; \} \\
\qquad\qquad\quad \mathbf{in}\ \ f'\ b\ a'); \} \\
\quad \mathbf{in}\ \ f\ a\ b
\end{array}
$$

In order to make the automatic search reach exactly this solution, one of the search restrictions must be relaxed. When refining $?_3$ we chose to use the induction hypothesis in an elimination refinement. In the subproblem we then did a case analysis. There is however a search restriction forcing all case analyses to appear immediately after the λ-abstractions. Not disabling the restriction also leads to solution but it is not as straightforward as the one above.

The program is able to introduce the equality chain such as the one above only because there is some object in the context which leads from either the left or the right hand side to a middle value. It can then solve the second equality by itself. The program can never come up with a middle value on its own, in the sense that at least one of the two subproofs of transitivity must be found immediately. An equality chain can consist of many steps, if time issues are not considered, but one proof at most can be non-immediate. It may however appear at any place in the chain, since search is performed both from the left and the right.

5.2 Reversing lists

This subsection presents a proof dealing with lists. It relies on doing a generalization at one point. The problem is to show that reversing a list twice is the identity function. This will be done for an arbitrary list element type, specified by its Datoid, X. To make the formulae more readable a couple of abbreviations will be used. The Setoid of the set of lists with $|X|$ as their elements, $\mathit{List}\ (\mathit{D2S}\ X)$, will be abbreviated to '$[X]$'. The corresponding Datoid will be written '$[X]_D$'.
Our problem is the following:

$$(X : \mathit{Datoid}) \rightarrow (xs : |[X]|) \rightarrow \mathit{rev}\ X\ (\mathit{rev}\ X\ xs) \stackrel{[X]}{==} xs$$

To specify the set of the list elements is not enough. We need the Datoid to be able to fully manipulate the equalities. The reversing function is defined in terms of concatenation.

$$\begin{array}{l}
\mathit{rev}\ (X : \mathit{Datoid})\ (xs : |[X]|) : |[X]| \\
\qquad \begin{array}{rcl}
\mathit{rev}\ X\ \texttt{[]} & = & \texttt{[]} \\
\mathit{rev}\ X\ (x :: xs') & = & \mathit{rev}\ X\ xs' +\!\!+ (x :: \texttt{[]})
\end{array}
\end{array}$$

$$\begin{array}{l}
(+\!\!+)\ (X : \mathit{Datoid})\ (xs, ys : |[X]|) : |[X]| \\
\qquad \begin{array}{rcl}
(+\!\!+)\ X\ \texttt{[]}\ ys & = & ys \\
(+\!\!+)\ X\ (x :: xs')\ ys & = & x :: (xs' +\!\!+ ys)
\end{array}
\end{array}$$

Concatenation of two lists, xs and ys, is written '$xs +\!\!+ ys$' where the argument X is omitted.

The description of this problem will not be as comprehensive as in the previous subsection. The complete proof term is directly presented and then there are a few comments on how it was reached. The proof term found by the program is presented on the next page.

$$
\begin{array}{l}
\lambda(X : \mathit{Datoid}) \to \lambda(\mathit{xs} : |[X]|) \to \\
\quad \textbf{let } f\ (\mathit{xs} : |[X]|)\ :\ \mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}) \overset{[X]}{==} \mathit{xs} \\
\qquad =\ \textbf{case } \mathit{xs}\ \{ \\
\qquad\qquad \texttt{[]} \to \texttt{tt}; \\
\qquad\qquad x :: \mathit{xs}' \to \\
\qquad\qquad\quad [X].\mathit{tran} \\
\qquad\qquad\qquad (\mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}' +\!\!+ (x :: \texttt{[]}))) \\
\qquad\qquad\qquad (x :: \mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}')) \\
\qquad\qquad\qquad (x :: \mathit{xs}') \\
\qquad\qquad\qquad (\textbf{let } g\ (\mathit{ys} : |[X]|) \\
\qquad\qquad\qquad\quad :\ \mathit{rev}\ X\ (\mathit{ys} +\!\!+ (x :: \texttt{[]})) \overset{[X]}{==} x :: \mathit{rev}\ X\ \mathit{ys} \\
\qquad\qquad\qquad\quad =\ \textbf{let } f'\ (\mathit{ys} : |[X]|) \\
\qquad\qquad\qquad\qquad\quad :\ \mathit{rev}\ X\ (\mathit{ys} +\!\!+ (x :: \texttt{[]})) \overset{[X]}{==} x :: \mathit{rev}\ X\ \mathit{ys} \\
\qquad\qquad\qquad\qquad\quad =\ \textbf{case } \mathit{ys}\ \{ \\
\qquad\qquad\qquad\qquad\qquad \texttt{[]} \to [X].\mathit{ref}\ (x :: \texttt{[]}); \\
\qquad\qquad\qquad\qquad\qquad y :: \mathit{ys}' \to \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{RespEq}\ [X]_D\ [X] \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda(x : |[X]|) \to (x +\!\!+ (y :: \texttt{[]}))) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{rev}\ X\ (\mathit{ys}' +\!\!+ (x :: \texttt{[]}))) \\
\qquad\qquad\qquad\qquad\qquad\qquad (x :: \mathit{rev}\ X\ \mathit{ys}') \\
\qquad\qquad\qquad\qquad\qquad\qquad (f'\ \mathit{ys}'); \} \\
\qquad\qquad\qquad\quad \textbf{in}\ \ f'\ \mathit{ys} \\
\qquad\qquad\qquad \textbf{in}\ \ g\ (\mathit{rev}\ X\ \mathit{xs}')) \\
\qquad\qquad\qquad (\mathit{RespEq}\ [X]_D\ [X] \\
\qquad\qquad\qquad\quad (\lambda(x' : |[X]|) \to x :: x') \\
\qquad\qquad\qquad\quad (\mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}')) \\
\qquad\qquad\qquad\quad \mathit{xs}' \\
\qquad\qquad\qquad\quad (f\ \mathit{xs}')); \} \\
\quad \textbf{in}\ \ f\ \mathit{xs}
\end{array}
$$

The proof is by induction on *xs*. The ground case is trivial. The inductive step has the target type

$$\mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}' +\!\!+ (x :: \texttt{[]})) == x :: \mathit{xs}'$$

and the induction hypothesis is

$$f\ \mathit{xs}' : \mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}') == \mathit{xs}'$$

The solution begins with an equality chain which is generated by invoking the induction hypothesis to replace xs' with $\mathit{rev}\ X\ (\mathit{rev}\ X\ \mathit{xs}')$ in the RHS.

This renders the subproblem

$$rev\ X\ \ (rev\ X\ \ xs' {+\!\!+} (x :: [\,]\,)) == x :: rev\ X\ \ (rev\ X\ \ xs')$$

Here the subexpression $rev\ X\ xs'$ appears twice and therefore the more general problem

$$rev\ X\ \ (ys {+\!\!+} (x :: [\,]\,)) == x :: rev\ X\ \ ys$$

is attacked, in addition to the original one. The generalized problem is solved by induction on ys. For the empty list we have

$$x :: [\,]\ == x :: [\,]$$

which is satisfied using reflexivity of $[X]$. For the inductive step the target type is

$$rev\ X\ \ (ys' {+\!\!+} (x :: [\,]\,)) {+\!\!+} (y :: [\,]\,) == x :: (rev\ X\ \ ys' {+\!\!+} (y :: [\,]\,))$$

The type of the inductive call is

$$rev\ X\ \ (ys' {+\!\!+} (x :: [\,]\,)) == x :: rev\ X\ \ ys'$$

This is used from left to right on the LHS which yields the expression

$$(x :: rev\ X\ \ ys') {+\!\!+} (y :: [\,]\,)$$

This however simplifies exactly to the RHS, so the proof is complete.

5.3 Quicksort

This is a case study of a more complex problem. It gives an example of the intended use of the program. The problem is to prove the correctness of the quicksort algorithm. The program cannot solve it as one entity and thus the problem must be divided into smaller pieces by hand. The proof consists of a number of lemmas followed by the main result. The formulation of this collection of subproblems has been taken from an external source. It is presented in [8].

The quicksort algorithm is informally defined in the following way:

$$\begin{array}{lll} qsort & [\,] & = [\,] \\ qsort & (x :: xs) & = qsort\ (filter\ (\leq x)\ \ xs) {+\!\!+} [x] {+\!\!+} qsort\ (filter\ (> x)\ \ xs) \end{array}$$

This function does however not adhere to structural recursion. To do this, a dedicated recursor must be added. The tool will only produce inductive

proofs based on structural recursion so here it must be done. The natural choice is the length of the list, since both recursive calls are made with lists strictly shorter than the submitted one.
To formulate the problem we need a number of definitions. First a few functions on lists.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{(+\!\!+)\ (X : \mathit{Setoid})\ (\mathit{xs}, \mathit{ys} : |\mathit{List}\ X|) : |\mathit{List}\ X|} \\
(+\!\!+)\ X\ \texttt{[]}\ \mathit{ys} & = & \mathit{ys} \\
(+\!\!+)\ X\ (x :: \mathit{xs}')\ \mathit{ys} & = & x :: (\mathit{xs}' +\!\!+ \mathit{ys})
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathit{filter}\ (X : \mathit{Setoid})\ (f : |X| \to |\mathit{Bool}|)\ (\mathit{xs} : |\mathit{List}\ X|) : |\mathit{List}\ X|} \\
\mathit{filter}\ X\ f\ \texttt{[]} & = & \texttt{[]} \\
\mathit{filter}\ X\ f\ (x :: \mathit{xs}') & = & \textbf{case}\ (f\ x)\ \{\texttt{t} \to x :: \mathit{filter}\ X\ f\ \mathit{xs}';\ \texttt{f} \to \mathit{filter}\ X\ f\ \mathit{xs}'\}
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathit{length}\ (X : \mathit{Setoid})\ (\mathit{xs} : |\mathit{List}\ X|) : |\mathit{Nat}|} \\
\mathit{length}\ X\ \texttt{[]} & = & \texttt{0} \\
\mathit{length}\ X\ (x :: \mathit{xs}') & = & \texttt{s}\ (\mathit{length}\ X\ \mathit{xs}')
\end{array}
$$

As in the previous subsection, the argument X of concatenation is omitted. The first argument of all functions will in this subsection be omitted except in the function's definition, in order to make the expression more compact. The first argument in all cases is the *Setoid*, *Datoid* or *Ord* of the list elements or of $|\mathit{Nat}|$, so it carries no essential information.
The correctness was in [8] proved for lists of natural numbers, but we choose to do it for any ordered set. Apart from making the result more general it also restricts search space, since no refinements are provoked by the inner structure of the elements. The following signature is used to host the properties of ordered sets.

$$
\begin{array}{l}
\mathit{Ord} : * \\
\quad = \textbf{sig}\ \{ \\
\qquad X : \mathit{Datoid}; \\
\qquad \mathit{leq} : |X| \to |X| \to |\mathit{Bool}|; \\
\qquad \mathit{tran} : (x, y, z : |X|) \to T\ (\mathit{leq}\ x\ y) \to T\ (\mathit{leq}\ y\ z) \to T\ (\mathit{leq}\ x\ z); \\
\qquad \mathit{ref} : (x : |X|) \to T\ (\mathit{leq}\ x\ x); \\
\qquad \mathit{asym} : (x, y : |X|) \to T\ (\mathit{not}\ (\mathit{leq}\ x\ y)) \to T\ (\mathit{leq}\ y\ x) \\
\quad\ \ \}
\end{array}
$$

Instead of taking merely a set, a Datoid is required. We will use '$|X|$' to denote the set when X is an *Ord*-structure, just as for Setoids and Datoids. There will be some abbreviations for the order relation.

$$\mathit{leq}\ (X : \mathit{Ord})\ (x, y : |X|) : |\mathit{Bool}| = X.\mathit{leq}\ x\ y$$

$$gr\ (X : Ord)\ (x, y : |X|) \to |Bool| = not\ (X.leq\ x\ y)$$
$$(\leq)\ (X : Ord)\ (x, y : |X|) : * = T\ (leq\ X\ x\ y)$$
$$(>)\ (X : Ord)\ (x, y : |X|) : * = T\ (gr\ X\ x\ y)$$

The functions $\leq$ and $>$ are invoked as infix operators omitting the argument of X.

An instance of Ord for natural numbers is declared. This is employed to ascertain properties about the primitive recursor.

$$\begin{aligned}
&Nat_O : Ord\\
&\quad = \textbf{struct}\ \{\\
&\qquad\qquad X = Nat_D;\\
&\qquad\qquad leq = \ldots;\\
&\qquad\qquad tran = \ldots;\\
&\qquad\qquad asym = \ldots\\
&\qquad \}
\end{aligned}$$

To formulate the modified quicksort algorithm we also need to establish a simple property of *filter*.

$$\begin{aligned}
&lemList\ (X : Setoid)\ (f : |X| \to |Bool|)\ (xs : |List\ X|)\\
&\quad : length\ (filter\ f\ xs) \leq length\ xs
\end{aligned}$$

Now we can define the function $qsort'$ which, apart from a list, takes a natural number and a proof that the length of the list does not exceed this number. It uses *lemList* to construct the corresponding proofs for the two recursive calls. The definition of $qsort'$ is found on the next page.

$$
\begin{array}{l}
\mathit{qsort'}\ (X : \mathit{Ord})\ (n : |\mathit{Nat}|)\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X.X)|) \\
\quad (p : \mathit{length}\ \mathit{xs} \leq n)\ :\ |\mathit{List}\ (\mathit{D2S}\ X.X)|
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{qsort'}\ X\ n\ \texttt{[]} & = & \texttt{[]} \\
\mathit{qsort'}\ X\ \texttt{0}\ (x :: \mathit{xs'}) & = & \textbf{case}\ p\ \{\} \\
\mathit{qsort'}\ X\ (\texttt{s}\ n')\ (x :: \mathit{xs'}) & = & \mathit{qsort'}\ X\ n'\ (\mathit{filter}\ (\mathit{gr}\ x)\ \mathit{xs'}) \\
& & \quad (\mathit{Nat}_O.\mathit{tran} \\
& & \qquad (\mathit{length}\ (\mathit{filter}\ (\mathit{gr}\ x)\ \mathit{xs'})) \\
& & \qquad (\mathit{length}\ \mathit{xs'}) \\
& & \qquad n' \\
& & \qquad (\mathit{lemList}\ (\mathit{gr}\ x)\ \mathit{xs'}) \\
& & \qquad p) \\
& & +\!\!+ \\
& & (x :: \texttt{[]}) \\
& & +\!\!+ \\
& & \mathit{qsort'}\ X\ n'\ (\mathit{filter}\ (\mathit{leq}\ x)\ \mathit{xs'}) \\
& & \quad (\mathit{Nat}_O.\mathit{tran} \\
& & \qquad (\mathit{length}\ (\mathit{filter}\ (\mathit{leq}\ x)\ \mathit{xs'})) \\
& & \qquad (\mathit{length}\ \mathit{xs'}) \\
& & \qquad n' \\
& & \qquad (\mathit{lemList}\ (\mathit{leq}\ x)\ \mathit{xs'}) \\
& & \qquad p)
\end{array}
$$

Now we can define the original function in terms of $\mathit{qsort'}$. As n we simply take the length of the list. Reflexivity provides the required proof.

$$
\begin{array}{l}
\mathit{qsort}\ (X : \mathit{Ord})\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X.X)|) : |\mathit{List}\ (\mathit{D2S}\ X.X)| \\
\quad = \mathit{qsort'}\ (\mathit{length}\ \mathit{xs}\ (\mathit{Nat}_O.\mathit{ref}\ (\mathit{length}\ \mathit{xs}))
\end{array}
$$

The correctness of the algorithm is established by proving that the output list is sorted and is a permutation of the input list.
The sortedness property of a list can be defined like this:

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathit{Sorted}\ (X : \mathit{Ord})\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X.X)|)\ :\ *} \\
\mathit{Sorted}\ X\ \texttt{[]} & = & \top \\
\mathit{Sorted}\ X\ (x :: \texttt{[]}) & = & \top \\
\mathit{Sorted}\ X\ (x :: x' :: \mathit{xs}) & = & (x \leq x') \wedge \mathit{Sorted}\ X\ (x' :: \mathit{xs})
\end{array}
$$

We define permutation by demanding that for each member of the element set, the number of occurrences is the same in both lists.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathit{occs}\ (X : \mathit{Datoid})\ (x : |\mathit{List}\ (\mathit{D2S}\ X)|)\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X)|)\ :\ |\mathit{Nat}|} \\
\mathit{occs}\ X\ x\ \texttt{[]} & = & \texttt{0} \\
\mathit{occs}\ X\ x\ (x' :: \mathit{xs}) & = & \textbf{case}\ (X.\mathit{eq}\ x\ x')\ \{\texttt{t} \to \texttt{s}\ (\mathit{occs}\ X\ x\ \mathit{xs});\ \texttt{f} \to \mathit{occs}\ X\ x\ \mathit{xs}\}
\end{array}
$$

$$Perm\ (X : Datoid)\ (xs, ys : |List\ (D2S\ X)|)\ :\ *$$
$$= (x : |X|) \to occs\ x\ xs \overset{Nat}{==} occs\ x\ ys$$

In addition, the assertion that an element appears in a list is used in some lemmas.

$$\begin{aligned} &Mem\ (X : Setoid)\ (x : |X|)\ (xs : |List\ X|)\ :\ * \\ &Mem\ X\ x\ \texttt{[]} \ = \ \bot \\ &Mem\ X\ x\ (x' :: xs) \ = \ (x \overset{X}{==} x') \lor Mem\ X\ x\ xs \end{aligned}$$

Finally we declare the collection of lemmas, the main theorem stating the correctness of *qsort′* and a corollary bringing the result back to *qsort*.

$$lem_1\ (X : Ord)\ (x, y : |X|)\ :\ (leq\ x\ y \overset{Bool}{==} \texttt{t}) \to x \leq y$$

$$\begin{aligned} &lem_2\ (X : Datoid)\ (p : |X| \to |Bool|)\ (xs : |List\ (D2S\ X)|)\ (x : |X|) \\ &\quad :\ Mem\ x\ (filter\ p\ xs) \to (p\ x \overset{Bool}{==} \texttt{t}) \end{aligned}$$

$$\begin{aligned} &lem_3\ (X : Ord)\ (xs : |List\ (D2S\ X.X)|)\ (x, y : |X|) \\ &\quad :\ Mem\ y\ (filter\ (leq\ x)\ xs) \to (x \leq y) \end{aligned}$$

$$\begin{aligned} &lem_4\ (X : Datoid)\ (x : |X|)\ (xs : |List\ (D2S\ X)|) \\ &\quad :\ Mem\ x\ xs \leftrightarrow \neg(occs\ x\ xs \overset{Nat}{==} \texttt{0}) \end{aligned}$$

$$\begin{aligned} &lem_5\ (X : Datoid)\ (xs, ys : |List\ (D2S\ X)|) \\ &\quad :\ Perm\ xs\ ys \to (x : |X|) \to (Mem\ x\ xs \leftrightarrow Mem\ x\ ys) \end{aligned}$$

$$\begin{aligned} &lem_6\ (X : Datoid)\ (x : |X|)\ (xs, ys : |List\ (D2S\ X)|) \\ &\quad :\ occs\ x\ (xs +\!\!+ ys) \overset{Nat}{==} occs\ x\ xs + occs\ x\ ys \end{aligned}$$

$$\begin{aligned} &lem_7\ (X : Datoid)\ (xs, ys, zs : |List\ (D2S\ X)|)\ (x : |X|) \\ &\quad :\ Perm\ zs\ (xs +\!\!+ ys) \to Perm\ (x :: zs)\ (xs +\!\!+ (x :: \texttt{[]}) +\!\!+ ys) \end{aligned}$$

$$\begin{aligned} &lem_8\ (X : Datoid)\ (xs_1, xs_2, ys_1, ys_2 : |List\ (D2S\ X)|) \\ &\quad :\ (Perm\ xs_1\ xs_2 \land Perm\ ys_1\ ys_2) \to Perm\ (xs_1 +\!\!+ ys_1)\ (xs_2 +\!\!+ ys_2) \end{aligned}$$

$$\begin{aligned} &lem_9\ (X : Ord)\ (xs : |List\ (D2S\ X.X)|)\ (x : |X|) \\ &\quad :\ Perm\ xs\ (filter\ (gr\ x)\ xs) +\!\!+ filter\ (leq\ x)\ xs) \end{aligned}$$

$$\begin{aligned} &lem_{10}\ (X : Datoid)\ (xs, ys, zs : |List\ (D2S\ X)|) \\ &\quad :\ Perm\ xs\ ys \to Perm\ ys\ zs \to Perm\ xs\ zs \end{aligned}$$

$$
\begin{array}{l}
\mathit{lem}_{11}\ (X : \mathit{Ord})\ (a : |X|)\ (\mathit{xs}, \mathit{ys} : |\mathit{List}\ (\mathit{D2S}\ X.X)|) \\
\quad :\ \mathit{Sorted}\ \mathit{xs} \rightarrow \mathit{Sorted}\ \mathit{ys} \\
\qquad \rightarrow ((x : |X|) \rightarrow \mathit{Mem}\ x\ \mathit{xs} \rightarrow x \leq a) \rightarrow ((x : |X|) \rightarrow \mathit{Mem}\ x\ \mathit{ys} \rightarrow a \leq x) \\
\qquad \rightarrow \mathit{Sorted}\ (\mathit{xs} +\!\!+ (a :: \mathtt{[]}) +\!\!+ \mathit{ys})
\end{array}
$$

$$
\begin{array}{l}
\mathit{thm}\ (X : \mathit{Ord})\ (n : |\mathit{Nat}|)\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X.X)|)\ (p : \mathit{length}\ \mathit{xs} \leq n) \\
\quad :\ \mathit{Sorted}\ (\mathit{qsort'}\ n\ \mathit{xs}\ p) \wedge \mathit{Perm}\ \mathit{xs}\ (\mathit{qsort'}\ n\ \mathit{xs}\ p)
\end{array}
$$

$$
\begin{array}{l}
\mathit{cor}\ (X : \mathit{Ord})\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X.X)|) \\
\quad :\ \mathit{Sorted}\ (\mathit{qsort}\ \mathit{xs}) \wedge \mathit{Perm}\ \mathit{xs}\ (\mathit{qsort}\ \mathit{xs})
\end{array}
$$

This is the original formulation translated into Agda, except for any ordered set replacing natural numbers as list elements. The question is how much must be changed before the program can solve the problem. The aim has of course been to do as few changes as possible, but we do not claim that the ones presented here are a minimum.

A technical matter is that the functions *filter* and *occs* contain two levels of case expression. This makes the program insufficiently simplify types containing either function, since a case expression is not considered to be a simplified type. Thus the innermost case expression must be replaced with an equivalent application. The solution is to use elimination for $|\mathit{Bool}|$.

$$
\mathit{elimBool}\ (X : *)\ (b : |\mathit{Bool}|)\ (x, y : X) : X = \textbf{case}\ b\ \{\mathtt{t} \rightarrow x;\ \mathtt{f} \rightarrow y\}
$$

We redefine *filter* and *occs* in terms of *elimBool*.

$$
\begin{array}{l}
\mathit{filter}\ (X : \mathit{Setoid})\ (f : |X| \rightarrow |\mathit{Bool}|)\ (\mathit{xs} : |\mathit{List}\ X|) : |\mathit{List}\ X| \\
\begin{array}{rcl}
\mathit{filter}\ X\ f\ \mathtt{[]} & = & \mathtt{[]} \\
\mathit{filter}\ X\ f\ (x :: \mathit{xs'}) & = & \mathit{elimBool}\ (f\ x)\ (x :: \mathit{filter}\ X\ f\ \mathit{xs'})\ (\mathit{filter}\ X\ f\ \mathit{xs'})
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{occs}\ (X : \mathit{Datoid})\ (x : |\mathit{List}\ (\mathit{D2S}\ X)|)\ (\mathit{xs} : |\mathit{List}\ (\mathit{D2S}\ X)|)\ :\ |\mathit{Nat}| \\
\begin{array}{rcl}
\mathit{occs}\ X\ x\ \mathtt{[]} & = & \mathtt{0} \\
\mathit{occs}\ X\ x\ (x' :: \mathit{xs}) & = & \mathit{elimBool}\ (X.\mathit{eq}\ x\ x')\ (\mathtt{s}\ (\mathit{occs}\ X\ x\ \mathit{xs}))\ (\mathit{occs}\ X\ x\ \mathit{xs})
\end{array}
\end{array}
$$

The first lemma, lem_1, seems very trivial, and indeed the proof term is short and easily found by the program. It is therefore tempting to try to remove it, but it turns out that it is a conclusion that the program cannot formulate on its own. Hence the lemma must stay. The usage of the lemma is to turn a proposition into an equality, for which transitivity can be used.

The lemma lem_2 is intended to be used in lem_3, but the program can solve lem_3 without it, so it can be removed. The proof of lem_3 then just depend on lem_1.

The next lemma, lem_4 is solved on its own, lem_5 depends on lem_4 and lem_6

is again solved independently.
At lem_7 the program runs into problems. To begin with, to find a solution the problem must be generalized as follows:

$$\begin{array}{l} lem'_7\ (X : Datoid)\ (xs, ys, zs : |List\ (D2S\ X)|)\ (x, a : |X|)\ (n : |Nat|) \\ \quad :\ (occs\ a\ zs \overset{Nat}{==} occs\ a\ (xs{+\!+}ys) + n) \\ \qquad \to (occs\ a\ (x :: zs) \overset{Nat}{==} occs\ a\ (xs{+\!+}(x :: \texttt{[]}){+\!+}ys) + n) \end{array}$$

Here we have added an arbitrary number, n, in the RHS of both the hypothesis and the conclusion. This particular generalization is not automatically invented by the generalization refinement rule.
By providing the following property of addition, lem'_7 can be solved.

$$lemNat_1\ (a, b : |Nat|)\ :\ a + \texttt{s}\ b \overset{Nat}{==} \texttt{s}\ (a + b)$$

This is also a trivial result, but cannot be proved on the fly when solving lem'_7. With the aid of lem'_7, lem_7 can then be solved if we also add another simple property of addition:

$$lemNat_2\ (a : |Nat|)\ :\ a + \texttt{0} \overset{Nat}{==} a$$

The lemma lem_8 is solved with the help of $lem6$ and lem_9 needs lem_6 and $lemNat_1$. Both lem_{10} and lem_{11} are solved without using previous lemmas.
For the main theorem we run into another problem. It seems that we must produce a result similar to lem_3 for gr instead of leq.

$$\begin{array}{l} lem'_3\ (X : Ord)\ (xs : |List\ (D2S\ X.X)|)\ (x, y : |X|) \\ \quad :\ Mem\ y\ (filter\ (gr\ x)\ xs) \to (y \leq x) \end{array}$$

To solve lem'_3 we in turn must have the counterpart of lem_1:

$$lem'_1\ (X : Ord)\ (x, y : |X|)\ :\ (gr\ x\ y \overset{Bool}{==} \texttt{t}) \to x > y$$

The proofs of these two propositions are similar to those of lem_3 and lem_1.
The main theorem uses several lemmas, namely lem_3, lem'_3, lem_5, lem_7, lem_8, lem_9, lem_{10} and lem_{11}. Finally, the corollary is trivial and depends only on the main theorem.

As you have seen, a few things had to be added in order to solve the whole problem, but most parts of the formulation could be left unchanged. The time it takes to solve some of the problems is just about how long a person can be expected to wait in an interactive system. The most time-consuming lemma, lem'_7, takes about one minute on a normal machine. The average time is a few seconds. This is when the needed lemmas are provided as hints. It is also possible to let the program search among all the global definitions made above the current position. When doing this, the proof search in general takes longer time, but most of the lemmas are still found within reasonable time.

6 Limitations

A number of deficiencies of the program which limits the amount of problems that can be solved have already been mentioned in the previous sections. This section contains a couple of previously unaddressed flaws, followed by a summary.

6.1 Parameter construction

As seen in previous sections, parameter meta variables are bound as a result of unification. In the situations where this is not enough, a special refinement method can do assumptions on the form of a parameter, e.g. suggest that a natural number parameter is equal to zero. These features both more or less only apply to parameters of data type. But there can be other kinds of parameters too, namely structures and functions. This subsection gives some suggestions of how to deal with such parameters.
Structures are not very difficult to handle. Nor does it seem to have any interesting applications. Assume there is parameter meta variable, ?, of the type **sig** $\{x : X,\ y : Y\}$. If the pattern $?.x$ appears in some type, the meta variable could be bound to the value **struct** $\{x = ?_x,\ y = ?_y\}$, introducing two new meta variables. This assumption can never be wrong.

Function parameters on the other hand appear frequently and are often essential. Many propositions say that, given a function, f, and an argument, x, something holds concerning the application $f\ x$. In order to invoke such a proposition, the program has to pick an f and an x such that $f\ x$ equals a certain expression, M. This could be done by enclosing M in a λ-abstraction, replacing a subexpression of the correct type with a function argument and assigning x to the subexpression. You could even try every combination of several occurrences of the function argument. This would make the program capable of solving many problems relying on such propositions, even if the treatment is merely syntactical, i.e. does not include expression reduction. In practice, though, search will be very cumbersome since a lot of different combinations will arise, especially when M is long or when f takes several arguments. Still, we think that for short expressions, this simple approach would be useful.

Another case involving function meta variables are proofs stating the existence of a function with certain properties. Sometimes it is unnecessary to formulate the problem in terms of function existence. Say for instance that we want to prove that there is a function, f, from X to Y, such that for each $x : X$, $P\ (f\ x)$. It is of course analogous to prove that there for each $x : X$ is a $y : Y$ such that $P\ y$. But, when the proposition specifies relations between different argument patterns, no such reformulation exists. It could for example be the existence of a function with reflexive, commutative or associative properties.

Some problems of this kind could be solved by binding the meta variable to a standard form containing the relevant λ-abstractions and case-expressions for the arguments. If we have $? : |Bool| \to |Bool| \to X$, it would be

$$
\begin{aligned}
? \;=\; & \lambda(b_1, b_2 : |Bool|) \to \textbf{case}\ b_1\ \{\\
& \quad \texttt{t} \to \textbf{case}\ b_2\ \{\texttt{t} \to ?_{\texttt{tt}};\ \texttt{f} \to ?_{\texttt{tf}}\}\\
& \quad \texttt{f} \to \textbf{case}\ b_2\ \{\texttt{t} \to ?_{\texttt{ft}};\ \texttt{f} \to ?_{\texttt{ff}}\}\}
\end{aligned}
$$

For non-recursive arguments and for some function properties this would be enough, but in general a whole different system for representing partial restrictions on functions would be called for. These kind of problems bring us into the huge area of model generation.

6.2 Summary

We now summarize all known limitations considered to be of major importance. First the situations which the program cannot even in principal deal with.

- One important issue is the limited ability to generalize. While specializing is rather straightforward to mechanize, this does not seem to be the case with generalization. In section 8 a few ideas of how to improve generalization abilities are presented.
- A similar problem is the inability to invent intermediate values in an equality chain. This was described in subsection 3.6.
- Generating function terms for meta variables, as discussed in subsection 6.1. The standard example which cannot be handled is to automatically invoke substitutivity for equalities. But also existence proofs oriented towards model generation are impossible for the program to solve.
- The program will only construct proofs containing structural recursion, at most.
- Mutually recursively defined sets are not handled.

- The parameter/proof classification applied on meta variables may not be flexible enough. Problems could arise for instance in some cases where there are compound objects containing both parameter like and proof like elements.

- Certain combinations of the program's features do not mix. One example is that for existence proofs, where the target type will contain a meta variable, the tool is unable to generalize the problem.

- The new Agda language component for defining indexed inductive families, **idata**, is unknown to the program. Inductive families have not at all been subject to investigation in this work.

Secondly, there are a few items concerning the speed of the program.

- Case analysis is generated in a rather arbitrary way. This make problems with several recursive set parameters very time consuming.

- The program has no system for giving a notion of direction in equality chains. This results in blind searching and in effect only proofs with very short chains can be constructed. This was described in subsection 3.6.

- When there are parameter meta variables which can be bound to a large amount of different values and the problem contains several coexisting subproblems, the search tend to be very slow. Possible explanations to this were presented in subsection 4.4.

- Upon choosing between the possible refinements of a problem, the program does not do much to try to predict which one will lead to a solution. There could be some good heuristics to this end which would improve performance.

7 Conclusions

The goal was to construct a tool that can solve fairly simple problems in the Agda/Alfa system. Looking at the result, a rather wide variety of problems can indeed be solved, including some which require inductive proofs and generalizations. Solutions spanning a few lines can normally be found in less than a second, which should be satisfactory for interactive proof construction. If the program is allowed to search for a bit longer, some proofs which cover several pages can be found. This gives an indication of that it could be meaningful to try to adapt the program to better deal with non-trivial problems.
The tool is similar to the interactive mode of the Coq system in many aspects. Several refinement rules have their counterpart among the elementary tactics of Coq and the proof search of this work is much like the auto tactics. Related to Coq, the contributions of this work would be that induction and generalization are included in the automatic search. The experiments show that non-trivial proofs with more than one level of induction can be found automatically within a few seconds, in spite of the fact that the search for the appropriate inductions is more or less blind.
Compared to the results in [10], it seems that searching for inductive proofs can be performed more quickly if it is carried out directly and not after encoding it into first-order logic.
The experiments with simplified types where higher-order unification is essentially avoided have been rather promising. The notion of type equality imposed by this has so far caused very limited trouble.
The approach when it comes to $\vee$-elimination could possible also prove successful, even though there are not yet any results which clearly support this.
The question remains of how generally applicable the tool is, since the number of test examples so far has been rather limited.

8 Future work

Although the program now has the ability to solve a fair amount of simple problems, many issues remain uncovered. This section lists a number of items possible for further development.

- Improve generalization abilities. The program should produce a more comprehensive list of generalizations to a problem. Some results from the ongoing work on automated generalization could perhaps be incorporated. It is however also important to consider the fact that a larger set of suggested generalizations causes search space to grow. To minimize the amount of candidates we suggest the use of testing. Whereas problems originating from the user are supposed to be true, this may not be the case for every generalized problem suggested by the program. By using testing to falsify generalizations we hope that many candidates can be discarded without being searched. As discussed in subsection 4.2, there is in a sense a notion of partial order for the types. A set of generalizations could thus be organized in a lattice. If we restrict ourselves to consider equalities, then the original type would appear in the bottom of it and at the top we would have the extreme generalization, $?_1 = ?_2$. The generalizations all appear in between. The original equality is assumed to be true, while the extreme one is normally false. From the partial ordering we can conclude that every equality below a true one should also be true and above a false equality only other false ones can be found. Somewhere in each path of the lattice, the state must go from true to false. Thus you need not test all generalizations; if a false one is encountered, none of the related types which are more general can be true. Otherwise, they could be specialized to prove the proposition found to be false.

- Better ability to choose between refinements in general. Perhaps it is possible to develop some heuristics for giving rough relative ratings of how difficult the subproblems of each refinement are. Some proposals to this are presented in [4]. Apart from heuristics, one idea is that maybe testing can be used for more than generalizations, namely refinements that invoke hypotheses and lemmas. If we need to prove $S\ x$ and there is an object of type $(x : X) \to P\ x \to S\ x$, then invoking it would be considered a refinement. But it may be the case that $S\ x$ holds for other reasons then those which could establish $P\ x$. And if we instead have $(x : X) \to Q\ x \to R\ x \to S\ x$ at our disposal, the problem would be divided into two parts, and it could be the case that for this specific problem the division would lead to false subproblems. These ideas are unsupported by facts or experiments. Still, the program certainly often spends time trying to solve a problem by some lemma, which in the end will not be used. But the question is how often the subproblems in these cases can be tested false and when they are simply not provable or indeed provable, but at a higher cost than by using some other lemma.

- Use rewriting rules or something to that effect to deal with equality chains. This area has been explored with great success, hence the technique of some existing system should be adopted. To use testing would not be meaningful in this case, since equality rewritings transport the problem between equivalent formulations.

- Picking the right variable for induction is another example where testing would certainly not be of any help. Case analysis on a variable may be unnecessary but it will never prohibit solving the problem in the branches. This is reflected by the fact that the subproblems of a case refinement are specializations of the original one and thus can be no more false than this. These thoughts suggest doing case analysis on all present variables right away. That would however lead to a large amount of branches for which the same search is partially repeated. Also, the answer, if not simplified, would be highly unreadable for the user.

- Handle the situation of coexisting subproblems in a better way. An important weakness of the proof search is that is performed depth-first. As the program works at the moment, searching for a proof of the existence of an object which should fulfill several criteria often becomes too tedious. Hopefully, such problems could be solved more efficiently within the frame of this program. One example would be to attack the set of criteria in parallel in order to narrow down the search at an earlier stage.

- Generation of parameters, especially functions, should be developed. The kind of generation which is vital to invoking many propositions seems rather easy to implement. But if only type consistency is considered, the amount of possible choices will often be large. Here again testing may partially solve the problem. Since the functions are generated without any insight, many of them should result in subproblems possible to falsify by testing.

- Improve the interactive search mode. To be of practical use when constructing proofs the user would need to be able to navigate in the search tree more easily. A clickable tree structure describing the search hierarchy would probably do huge difference. Another thing would be to do a graphical interface, by incorporating it in Alfa.

- Make the solutions more readable. Some after-touch is performed on the proof terms, but many more things could be done which are not difficult to implement.

Testing is referred to several times above as a possible way to limit search space. It is uncertain how successful it would be. But the fact that a testing tool for Agda/Alfa has already been developed[5] should make it quite easy to try. Apart from the items above which are all improvements of the program, it would also be interesting to investigate how the implementation could be supported by theoretical results. Even though the general task of solving a problem in dependent type theory is undecidable, there may still be subsets of problems which enjoy certain properties. One interesting topic could be to investigate notions of canonical forms for specifications and proofs in Agda and this way reveal some details of the underlying combinatorial problem of constructing a proof term.

References

[1] Thomas Hallgren, *The Alfa homepage*, `http://www.cs.chalmers.se/~hallgren/Alfa`.

[2] Catarina Coquand, *The Agda homepage*, `http://www.cs.chalmers.se/~catarina/agda`.

[3] B. Nordström, K. Petersson, J. M. Smith, *Programming in Martin-Löf's Type Theory*, Oxford University Press, 1990.

[4] Marcin Benke, *Strategies for interactive proof and program development in Martin-Löf Type Theory*, Workshop Proceedings, IJCAR'2001.

[5] Qiao Haiyan, *Testing and Proving in Dependent Type Theory*, Dissertation, Department of Computing Science, Chalmers University of Technology, 2003.

[6] Peter Dybjer, *Inductive Families*, In Formal Aspects of Computing 6, 1994, pp 440-465.

[7] Coq Development Team, *The Coq Proof Assistant Reference Manual*, `http://coq.inria.fr`.

[8] Simon Thompson, *Type Theory and Functional Programming*, Addison-Wesley, International Computer Science Series, 1991.

[9] C. Coquand and Th. Coquand., *Structured Type Theory*, In Proc. Workshop on Logical Frameworks and Meta-languages (LFM'99), 1999.

[10] Tanel Tammet, Jan M. Smith, *Optimized Encodings of Fragments of Type Theory in First Order Logic*, Journal of Logic and Computation, vol 8:6, pp 713-744, 1998.

[11] Sabine Broda, Luís Damas, *The Formula-Tree Proof Method*, Technical Report DCC-2004-01, DCC - FC & LIACC, Universidade do Porto, January, 2004.

[12] Gilles Dowek, *Higher-Order Unification and Matching*, A. Robinson and A. Voronkov (eds.), Handbook of Automated Reasoning, Elsevier, 2001, pp. 1009-1062.