Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 196 (2008) 69–84
www.elsevier.com/locate/entcs

Higher-Order Proof Construction Based on First-Order Narrowing

Fredrik Lindblad[1]

Dept. of Computer Science and Engineering
Chalmers University of Technology / Göteborg University
Gothenburg, Sweden

Abstract

We present the idea of using a proof checking algorithm for the purpose of automated proof construction. This is achieved by applying narrowing search on a proof checker expressed in a functional programming language. We focus on higher-order formalisms, such as logical frameworks, whereas the narrowing techniques we employ are first-order. An obvious advantage of this approach is that a single representation of the semantics can in principle be used for both proof checking and proof construction. The correctness of the search algorithm is consequently more or less trivially provided. The question is whether this representation of the search procedure allows a performance plausible for practical use. In order to achieve this, we add some features to the general narrowing search. We also present some small modifications which can be applied on a proof checker and which further improve the performance. We claim that the resulting proof search procedure is efficient enough for application in an interactive environment, where automation is used mostly on small subproofs.

Keywords: Higher-Order Proof Construction, Narrowing, Logical Frameworks, Type Theory

1 Introduction

Narrowing is the study of efficiently evaluating declarative programs in the presence of unknown data and non-deterministic functions. A narrowing strategy which is complete for inductively sequential term rewrite systems[2] can be used to turn a decidable predicate expressed in a standard functional programming language into a search procedure for its members. Given e.g. a predicate deciding whether a list of natural numbers is sorted,

$$\texttt{sorted} : [\texttt{Nat}] \to \texttt{Bool},$$

narrowing can be used to construct sorted lists of numbers.

We propose to analogously apply this idea on a proof checking algorithm in order to get a proof search algorithm for free, so as to speak. Given a proof checking

[1] Email: fredrik.lindblad@cs.chalmers.se

doi:10.1016/j.entcs.2007.09.018

algorithm relating propositions and proofs,

$$\texttt{proofCheck} : \texttt{Prop} \to \texttt{Proof} \to \texttt{Bool},$$

we can fix the proposition and apply narrowing search for an unknown proof. The general search procedure will then try to construct proofs of the given proposition.

One approach to develop a proof construction tool for a given higher-order formalism is to add *meta variables* (or *logical variables*) to the abstract syntax of the proof language. These are used as place holders for unknown data. When searching for a proof, a single meta variable is created at the start, indicating that the entire proof term is initially unknown. The meta variable is then instantiated step-by-step, adding new meta variables representing unknown sub-terms. While instantiating, one keeps track of the semantics of the formalism, back-tracking whenever the current partially instantiated proof turns out to be incorrect.

This approach entails the need of making parts of the functionality, which is essentially shared by the proof checker and proof search, aware of meta variables. E.g. the evaluation of terms has to be able to return a partially evaluated term if it encounters an uninstantiated meta variable. Likewise, the term comparison needs to be able to answer "maybe equal". To give a third example, substitutions must be postponed when encountering a meta variable. Otherwise the instantiation of meta variables and evaluation of terms do not commute. This requires some extra book-keeping, such as introducing *explicit substitutions* to the proof term syntax. Apart from this the proof search algorithm is in a sense a dual representation of the semantics, which can cause inconsistency problems between the checking and search algorithms. Also, the search algorithm is typically larger and more intricate than the checking algorithm. One issue that complicates the implementation of a search algorithm is the need of a refined mechanism for deciding the order in which the sub-terms of a proof are instantiated. In our experience, one ends up craving for a way to control the execution and search branching on the meta level, in order to deal with the fact that a meta variable can be encountered in a large number of different places in the algorithm. Controlling the execution and search branching on the meta level is exactly what narrowing does.

There would be several advantages of the proposed way of attaining proof search from a proof checker. Instead of implementing an often intricate and tedious proof search algorithm, the proof checker, which presumably already exists, is reused. Apart from saving work, this entails that there is little potential for inconsistency. In other words, given that the narrowing search procedure is sound and complete, the same properties are inherited by the proof search. Thus the correctness of the search is in principal directly provided. Another advantage would be that meta variables are handled by the narrowing algorithm. Hence, there is no need to add meta variables to the term syntax, and functions like evaluation and comparison do not need to be aware of them. Also, since the narrowing is first-order, the search algorithm is pretty simple. In this work narrowing is used to construct higher-order proofs. To make this possible the terms are represented in a first-order abstract syntax.

The potential drawback of the approach which could overshadow all these benefits is of course that the resulting search procedure, although working in theory, is not efficient enough for practical use. We have investigated this by implementing a narrowing search algorithm and a proof checker. Our aim has been to achieve a proof construction tool which automates the construction of proofs which are rather small. It is supposed to be useful in an interactive proof construction environment as an aid for filling in not too complex sub-terms in a proof. It is not meant to compete with advanced algorithms for constructing higher-order proof terms. Instead, the intended point of the approach is to add automation to a formal system for higher-order logic at a low cost.

The formalism we have chosen is a logical framework with dependent types, and recursive data-types and function definitions. Hence the proof checker is in fact a type checker, and will we from now on use the terminology of the Curry-Howard correspondence, i.e. refer to proofs as terms and propositions as types. We have implemented a type checker for the formalism in Haskell. Applying the narrowing search on this type checker indeed does not at first give a proof construction tool which could be used in practise. However, our experiments indicate that, by adding a couple of general features to the narrowing search, as well as introducing some small modifications to the type checker, the performance is substantially improved. We will present the most crucial (in our experience) of these general features and modifications. A central idea of the work is that the same code should in principle be able to serve as a description for both checking and searching. Hence, the modifications introduced to the type checking algorithm should preserve its meaning as a Haskell program.

2 Related Work

Our underlying search procedure is based on narrowing. A survey of various narrowing strategies is found in [2]. A notion of parallel evaluation was presented by Antoy et. al. in [3]. We have used a slightly different notion of parallel evaluation, which is described in [10]. Higher-order term construction using first-order narrowing was investigated by Antoy and Tolmach[5]. Our work is related to this in the sense that we have also looked at using first-order narrowing to construct higher-order terms. The difference is that Antoy and Tolmach focused on constructing terms in the declarative language itself, whereas we encode a new language in a first-order data-type and search for objects of that data-type.

Algorithms for term construction in type theory has been studied by e.g. Dowek[8] and Strecker[14]. Strecker's work is far-reaching, but does not cover systems which have defined recursive constants.

Finding efficient strategies for automatically constructing proof terms is an extensively studied area. The concept of *uniform proofs* largely reduces redundancy in proof search and is implemented in e.g. the formal system Twelf [11,12]. *Focused derivations* is a development of this which further improves performance by detecting chains of construction steps for which the search can proceed deterministically

[1]. *Tabling* is another technique for narrowing down search space [13]. It is based on memoizing subproblems in order to avoid searching for the same proof more than once. Our approach cannot compete with these advanced proof construction strategies. Nonetheless we do add some restrictions to the type checker in section 5 which remove certain kinds of redundancy. More refined restrictions could probably be introduced, and a tabling mechanism could possibly be added to the general search procedure. However, such advanced features are outside the scope of this investigation.

In the area of getting term construction "for free" a couple of contributions should be mentioned. Augustsson's tool Djinn[6] converts a Haskell type to a first-order proposition and sends it off to a theorem prover. The result is then interpreted as a λ-term. Tammet and Smith have presented a set of optimized encodings for a fragment of type theory into first-order logic[15]. Here, too, an external theorem prover is used. The fragment includes inductive proofs, but much information is lost in the translation, so even rather trivial problems are hard for the tool to find. Related is also Cheney's experiment on using logic programming to generate terms for the purpose of testing a type checker[7].

3 Basic Approach

The approach of the work is to apply narrowing on a rewrite system which is possible to use as a type checker by executing it as a Haskell program. This means the program should not contain any non-deterministic functions. Hence, the narrowing search does to begin with not need to handle more general systems than inductively sequential term rewrite systems (TRSs). However, since we will introduce the notion of parallel conjunctions to improve performance, we will consider the slightly larger class of *weakly orthogonal* TRSs[2]. These include programs where function definitions may overlap, but only in such a way that overlapping definitions yield the same result. Instead of using an already existing implementation of functional logic programming, we decided to implement our own narrowing algorithm for weakly orthogonal TRSs. The reason for this was to be able to experiment with a couple of features which are not supported by existing implementations. These features are presented in the section 4. Our implementation is based on a variation of lazy needed narrowing strategy [4], which is the most common strategy and is the basis of e.g. Curry [9]. The term syntax of a logical framework are essentially recursively defined. When applying narrowing for an unknown term, its instantiation can in general continue indefinitely. In order to deal with this, our narrowing search is based on measuring the size of the generated term and exploring the potentially infinite search space by iterated deepening. Our implementation reads the *ghc-core* format, which does not contain pattern matching, only case expressions. Thus the narrowing search need not include a stage which constructs the definitional trees of a rewrite system. This is taken care of by the ghc compiler. The blocking position of an expression is decided by traversing the trees of case expressions which define the functions.

```
data Type = El Term
          | Set

data Term = App Var [Term]
          | Lam Var Term
          | Pi Var Type Type

tc :: Ctx -> Type -> Term -> Bool
tc ctx etp trm = case trm of
 App v as -> case tiv ctx v of
  Just ftp -> case tis ctx ftp as of
   Just itp -> eqtp ctx etp itp
   Nothing -> False
  Nothing -> False
 Lam v b -> case hntp ctx etp of
  El (Pi v' itp otp) ->
   tc (extctx v itp ctx)
    (subtp v' (App v []) otp) b
  _ -> False
 Pi v itp otp -> case etp of
  Set -> typ ctx itp &&
         typ (extctx v itp ctx) otp
  _ -> False

tiv :: Ctx -> Var -> Maybe Type

tis :: Ctx -> Type -> [Term] -> Maybe Type
tis ctx tp as = case as of
 [] -> Just tp
 (a:as') -> case hntp ctx tp of
  El (Pi v itp otp) -> if tc ctx itp a then
    tis ctx (subtp v a otp) as'
   else
    Nothing
  otherwise -> Nothing

typ :: Ctx -> Type -> Bool
typ ctx tp = case tp of
 El trm -> tc ctx Set trm
 Set -> True

hntp :: Ctx -> Type -> Type
hntp ctx tp = case tp of
 El trm -> El (hn ctx trm)
 Set -> Set

hn :: Ctx -> Term -> Term
subtp :: Var -> Term -> Type -> Type
eqtp :: Ctx -> Type -> Type -> Bool
empctx :: Ctx
extctx :: Var -> Type -> Ctx -> Ctx
```

Fig. 1. Fragment of a type checker for a logical framework with dependent types

3.1 A simple type checker

Next we will present the core parts of a type checker for a logical framework with dependent types. It will be referred to in this section and later on in conjunction with a few simple examples in order to make clear the basic mechanisms of the approach. The type checker is presented in figure 1. Types and terms are represented by `Type` and `Term` respectively. A term is either an application of a variable on a list of arguments, a λ-abstraction or a dependent function arrow. Later on we will also use a concrete syntax for the terms whenever the thereby obscured details are not important. We will write $(x\ t\ \ldots\ t)$ and $\lambda x \to t$ for applications and abstractions respectively. Functions arrows will in general be denoted by $(x : t) \to t$, while non-dependent functions will be written $t \to t$. The `El` construction will be omitted in the concrete syntax, and terms and types will not be explicitly distinguished. The representation of variables, `Var`, and contexts, `Ctx`, are left abstract. A context is assumed to contain type declarations of local variables and global constants, as well as the reduction rules for defined global constants. The function `tc` decides the correctness of a term with respect to a given context and type. When checking applications `tis` is used to traverse the list of arguments. The correctness of a type is decided by `typ` and `hntp` reduces a type to head normal form in case it is a term. For the remaining functions only the type signature is given. The type of a variable is looked up in the context by `tiv`. The function `hn` reduces a term to head normal form, and `subtp` substitutes a variable for a term in a type. The definitional equality between two types is decided by `eqtp`. Finally, `empctx` represents the empty context and `extctx` extends a context with a new local variable type declaration.

3.2 Example of a proof search

In order to illustrate the basic mechanism of applying narrowing search on the presented type checker let us look at the proposition $A \wedge (A \to B) \to B$. This can

be encoded as the type $(A : \texttt{Set}) \to (B : \texttt{Set}) \to A \to (A \to B) \to B$, or in our abstract syntax:

$$\begin{aligned}\texttt{goalt} \equiv \texttt{El}\ (\texttt{Pi}\ A\ \texttt{Set}\ (\texttt{El}\ (\texttt{Pi}\ B\ \texttt{Set}\ (\texttt{El}\ (\texttt{Pi}\ x\ (\texttt{El}\ (\texttt{App}\ A\ []))\\ (\texttt{El}\ (\texttt{Pi}\ y\ (\texttt{El}\ (\texttt{Pi}\ z\ (\texttt{El}\ (\texttt{App}\ A\ []))\ (\texttt{El}\ (\texttt{App}\ B\ [])))))\\ (\texttt{El}\ (\texttt{App}\ B\ [])))))))))\end{aligned}$$

In order to find a term of this type using the proposed approach we should apply narrowing on the expression

$$\texttt{tc}\ \texttt{empctx}\ \texttt{goalt}\ ?_1,$$

where $?_1$ is a meta variable serving as a placeholder for the yet unknown term.

With a lazy narrowing strategy the expression for which the search is performed is evaluated in a lazy evaluation order, i.e. from the outside and in. Whenever an uninstantiated meta variable is encountered it is said to be in the *blocking position*. The meta variable in the blocking position is chosen for refinement. It is non-deterministically refined to one of the constructors in its type. For each constructor, fresh meta variables are inserted at the argument positions. Then the evaluation of the expression proceeds until a new blocking meta variable is encountered or a value is reached. If the value is `True` a solution has been found. If it is `False` a dead-end has been reached and the search back-tracks.

Proceeding with the example, we start evaluating the given expression lazily. Since `tc` does a case distinction on the third argument, we need to know the head constructor of the term. But the term is $?_1$, which thus blocks further evaluation. There are three possible refinements of $?_1$. The refinement $?_1 := \texttt{App}\ ?_2\ ?_3$ yields an expression with $(\texttt{tiv}\ \texttt{empctx}\ ?_2)$ surrounding the potentially blocking position. Assuming that `tiv` returns `Nothing` for the empty context regardless of its second argument, the evaluation proceeds without further refinement with the result being `False`, which means no solution. The refinement $?_1 := \texttt{Pi}\ ?_2\ ?_3\ ?_4$ immediately yields `False` since the type is not `Set`. Finally, the refinement $?_1 := \texttt{Lam}\ ?_2\ ?_3$ results in the expression

$$\texttt{tc}\ (\texttt{extctx}\ ?_2\ \texttt{Set}\ \texttt{empctx})\ (\texttt{subtp}\ A\ (\texttt{App}\ ?_2\ [])\ (\texttt{Pi}\ B\ \ldots))\ ?_3,$$

where $?_3$ is the blocking meta variable.

The search proceeds similarly until all four λ-abstractions have been constructed. The difference is that the context is no longer empty, which means that `App` is not immediately rejected. However, before all λ-abstractions are introduced, the refinement to `App` will eventually fail and the search will back-track. At the resulting state, the term under construction has been instantiated to

$$\texttt{Lam}\ ?_2\ (\texttt{Lam}\ ?_4\ (\texttt{Lam}\ ?_6\ (\texttt{Lam}\ ?_8\ ?_9)))$$

and the expression is essentially

$$\texttt{tc}\ \ldots\ (\texttt{El}\ (\texttt{App}\ B\ []))\ ?_9.$$

Consider the refinement $?_9 :=$ `App` $?_{10}$ $?_{11}$. Then $?_{10}$ is blocking the evaluation. If $?_{10}$ is set to be equal to $?_8$, `tiv` should return

`Just` (`El` (`Pi` z (`El` (`App` A [])) (`El` (`App` B [])))).

The values of $?_8$ and $?_{10}$ are not important, as long as they are equal. Let us choose them to be y. Now $?_{11}$ becomes the blocking meta variable. The refinement $?_{11} :=$ [] results in the comparison between the expected type, B, and the type inferred by `tiv`, namely $A \to B$. These are not equal and hence the expression evaluates to `False`. The refinement $?_{11} :=?_{12} :?_{13}$ makes $?_{12}$ block. Refining $?_{12}$ to `App` $?_{14}$ $?_{15}$, setting $?_{14}$ to be equal to $?_6$ and refining $?_{15}$ to [] results in comparing the expected and inferred type for the inner application (which has no arguments). The types are both A, so the search proceeds. The common value of $?_6$ and $?_{14}$ is also arbitrary, as long as it is different from y. Let us choose it to be x. Looking at the definition of `tis` the blocking meta variable is now $?_{13}$. After refining $?_{13}$ to [], the expected and inferred type for the outer application is compared. They are both B, so the expression finally evaluates to `True`, indicating that we have a solution. The term is composed by all current refinements which expands to

`Lam` $?_2$ (`Lam` $?_4$ (`Lam` x (`Lam` y (`App` y ((`App` x []) : [])))))).

This term represents a proof for the given problem.

3.3 Comments on the suitability of using narrowing

The basic idea of narrowing search is that, by interleaving instantiation and evaluation, and choosing the order of instantiation in a clever way, the data in many cases do not need to be fully instantiated before the predicate is known to return `False`. Every time this happens, all the data instances that are specializations of the current partial instantiation can be skipped. Thereby the search space is reduced. Predicates which are suitable for narrowing are typically to large extent defined by recursion on the structure of the unknown data. Looking at the example above, type checking does seem to be of this kind.

However, the example could very well be formalized in a system without dependent types. In a logical framework with dependent types, argument types and the output type may depend on a previous argument value in applications. This leads to some complications which were not exposed in the example. In the following two sections these complications will be discussed along with suggestions for how to deal with them.

One complication which did appear in the example was the treatment of variables. When constructing variable occurrences, the description of the search was rather irregular, stating that two variables should be the same rather than instantiating a single meta variable. Also, the variables which were never used were left uninstantiated. These problems are however easy to avoid. When representing variable occurrences by de Bruijn indices, no arbitrary choices need to be made, and no uninstantiated variables at binding position will appear. Furthermore, if recursively defined numbers are chosen to represent the indices, the narrowing algorithm will itself limit the search to the set of possible indices for a given context.

4 Features of the General Search Procedure

This section describes the two main non standard features of the general search algorithm we have investigated. As stated above, the search procedure targets weakly orthogonal TRSs. In order to deal this class of rewrite systems efficiently, a concept of *parallel evaluation* has been proposed[3]. In [10] a somewhat different notion of parallel evaluation is presented, which is used in our implementation. Using this feature for the purpose of parallel conjunction is discussed in section 4.1. Section 4.2 introduces the idea of *subproblem separation* which is an attempt to overcome the performance loss which follows from using parallel conjunction in some cases.

4.1 Parallel Conjunction

Let us now look at a simple example which does involve dependent types. Assume that we have the situation

$$?_1 : P\ M,$$

where $P : X \to \texttt{Set}$ is a variable and M is a term of the correct type. Also assume that $h : (x : X) \to P\ x$ is in scope. Constructing a proof by involving h proceeds by the refinement

$$?_1 := h\ ?_2.$$

The type checker presented in the previous section is devised to check the correctness of the arguments in an application from left to right, and at the end check the equality between the expected and the inferred type. In this example that amounts to first checking $?_2 : X$ and then checking $P\ M = P\ ?_2$. This is a natural choice since it means that terms are always type checked before they are used in computations. However, for the purpose of proof search, type checking before equality checking is not desirable, since the latter is in general more restrictive. Type checking the argument, $?_2$, first means that all terms with the correct type are constructed before their equality to M is decided.

To amend this inefficiency there are two rather straightforward ways to go. One is to reverse the order in which the constraints are checked in an application, as discussed in [8]. The other is to check the conditions in parallel. Both these approaches build on the fact that the result of type checking an argument is not needed for type checking the remaining arguments or for the final equality check. However, both of them also introduce the hazard that a term which has not been constructed in a type correct way exposes partiality in the term reduction functions.

In our implementation we chose the second of these approaches with the motivation that checking restrictions in parallel during the narrowing search should in general give a smaller search space than checking them sequentially, regardless of the order. There are also situations where parallel checking is useful, and where the best order is not as clear as in the case of type checking an application. The following example illustrates this:

$$\exists\ X\ (\lambda x \to P\ x \wedge Q\ x)$$

Constructing a proof of this proposition will yield an intermediate state with the constraints

$$?_1 : X, \quad ?_2 : P\ ?_1, \quad ?_3 : Q\ ?_1.$$

The type judgments of $?_2$ and of $?_3$ both put restrictions on $?_1$. By instantiating the meta variables and taking all the type constraints into account in parallel, a more narrow search space can be achieved.

As mentioned, the hazard of giving conjunctions a parallel meaning is that an intentional partiality in the right conjunct is exposed. The implications are different for non-definedness and for non-termination partiality. Non-definedness can be easily handled by treating a undefinedness error in the right conjunct of parallel conjunction as `False`.

Non-termination is more delicate. A term can be non-terminating in two ways. It is either essentially type correct but recursive in a non-terminating way, or it is not type correct and non-terminating, like the Ω-term. In our experiments we have not implemented a termination checker. Instead, we have allowed recursion only via given elimination rules. We have not experienced any problems caused by nontermination. A formal characterization of when and why this is provably safe would be desirable, but we have had to leave this as future work. An informal motivation of why we have not encountered any problems is that nonterminating terms are either not constructible in the syntax or have to be constructed via intermediate steps which are not type correct.

With the needed narrowing strategy, there is always a unique meta variable to branch the search on. However, in the presence of parallel conjunction, or parallel evaluation in general[10], there is no longer a single meta variable blocking the evaluation. The order in which to instantiate meta variables must hence be further specified. A natural choice is to store them in a collection and extract them in either a queue or a stack manner. In our experience the queue is in general, but not always, the better choice regarding performance[10]. The order of instantiating blocking meta variables can also be controlled in more refined ways, which we will come back to in section 5.2.

4.2 Subproblem Separation

Parallel conjunction was introduced in order to check several properties in parallel during the incremental instantiation of a term. This seems to be beneficial in various situations where the properties constrain the same part of the term. However, for problems where there are sub-terms with no dependencies in between, it seems undesirable to interleave the search for their solutions. Consider the situation

$$?_1 : P \wedge Q,$$

where P and Q have no meta variable occurrences. Interleaving the construction of a proof of P and a proof of Q is unnecessary and should lead to a larger search space than if the to subproofs were constructed separately. This is a drawback of switching to parallel conjunction as discussed in sec 4.1. Moreover the situation is very common. It can appear whenever attempting to prove a proposition by case

distinction, such as in proofs by induction.

A solution to this drawback could be to add a feature of *subproblem separation* to the general narrowing search. One part of the mechanism would be to detect the presence on independent subproblems, i.e. unconnected graphs where parallel conjuncts and meta variables constitute the vertices and the edges represent meta variable occurrences in the conjuncts. When such a partitioning has been detected, a local search for each subproblem is spawned. When performing the search for several independent subproblems backtracking one of them should not affect the other ones, and when the search space is exhausted for one of them, itself and all of its sibling subproblems are cancelled. We have implemented this feature, but it should be considered a prototype.

A rather artificial example will illustrate the reduction in search space which can be gained using subproblem separation. Let the initial problem be

$$?_1 : P$$

and let the following hypotheses be in scope:

$$g_1 : P_1 \to P_2 \to P$$

$$g_2 : P_3 \to P_4 \to P$$

$$h_{ij} : P_{ij} \to P_i, \quad \text{for } 1 \leq i \leq 4, 1 \leq j \leq k_i$$

There are two alternative ways to prove P, namely by applying either g_1 or g_2 on two arguments. In both cases, the construction of the two arguments are independent of each other. The subproblems are in turn matched by unary applications invoking the hypotheses h_{ij}. A complete proof is not possible to construct in the given context, but that is not important. We will merely measure the size of the search space by counting the number of leaves in the spanned tree. Assuming parallel conjunction is used, the instantiation of the first sub-term will be interleaved with the instantiation of the second. The number of leaves in the search tree thus amounts to $k_1 \cdot k_2 + k_3 \cdot k_4$. Now consider the case where the subproblem separation feature is present. When the term has been instantiated to either $g_1\ ?_2\ ?_3$ or $g_2\ ?_2\ ?_3$, the independence between the construction of $?_2$ and $?_3$ is detected. Assuming that the execution alternates between the two separate subproblems in a fair way, the number of leaves is $2 \cdot min(k_1, k_2) + 2 \cdot min(k_3, k_4)$. Hence the size of the search space is linear in k_i instead of quadratic. Subproblem separation could enable the proof search to scale considerably better.

5 Optimizations of the Type Checker

The features presented in the previous section, parallel conjunction and subproblem separation, do not result in a proof search which is of practical use. In order to improve the performance, we have also experimented with some modifications of the type checker. In this section we will discuss a few such modifications which have proved to be important in our experiments.

5.1 Type Checking Restrictions

Most logical frameworks allow writing essentially the same proof in many different ways. By restricting the type checker to accept fewer terms for a given type, a reduction of the search space can be accomplished. One restriction is to only allow normal terms. It can be easily achieved by adding a side condition checking that no sub-term is reducible. Imposing this restriction does not compromise the completeness of the search. Another possibility could be to add restrictions which allow only uniform proofs.

One can of course also come up with a large number of restrictions, which do limit the completeness. These can be seen as representations of different heuristics. One example is to restrict induction so that no generalizations may take place. This clearly makes the search incomplete, but also contributes a lot to performance.

5.2 Meta Variable Prioritization

As mentioned in section 4.1, the search is based on keeping a queue of blocking meta variables and instantiating them one at a time. By slightly annotating the code of the type checker, one can introduce a notion of priority which refines the scheduling of the meta variable instantiation. Controlling the order of instantiation this way does not affect the completeness, apart from the possibility that an otherwise finite search space could become infinite.

The following simple prioritization has made a great performance improvement in our experiments:

- *high* – equality constraints
- *medium* – type checking constraints for proof terms
- *low* – type checking constraints for non-proof terms and when the type is unknown

By *proof term* we mean sub-terms which correspond to a proof step, i.e. whose inhabitation is not trivial. Non-proof terms are the rest, i.e. terms which are typically trivially inhabited and appear in some equality constraint.

The idea behind this prioritization is that equality constraints are in general more restrictive than type checking constraints. The reason for postponing the type-checking of non-proof terms is that it may be the case that the term does not yet appear in an equality constraint, although it will after further instantiating some proof terms. If the instantiation of the non-proof term is initiated before an equality constraint add further restrictions, the search is quite arbitrary.

In order to be able to prioritize proof and non-proof terms differently, there must be a way to tell them apart. One option is to distinguish between dependent and non-dependent function types. Application arguments which stem from dependent function types are treated as non-proof terms and those from non-dependent function types as proof terms. We have chosen this approach in our implementation.

5.3 Special Treatment of Equality Constraints

Checking equality constraints in a dependently typed system is typically implemented by first reducing the left and right hand sides to head normal form, and then comparing the heads and recursively repeating the procedure for the sub-terms. Assume we apply narrowing on a type checker implemented like this. If one of the compared terms is a meta variable, it is necessary for the search algorithm to guess among all global constants, reduce the term and see if it equals the opposite side. If we have e.g. have the equality constraint

$$?_1 = \texttt{add}\ T\ U,$$

the search must guess $?_1$ to be $\texttt{add}\ ?_2\ ?_3$. This is of course a possible solution, but we would like to make better use of the information that is given in the initial type defining the problem. The meta variable on one side should be able to mimic the term on the other side, just like in unification.

The basic approach to achieve this is to compare the terms without first reducing them. Of course, never reducing the terms when comparing for equality is not a real option. That would render the system too weak. A possible option is to mix reduction with first-order unification, but it is not so clear how this should be done. One way to do this is to check for each term whether it is a meta variable (is currently unknown) and only reduce terms which are not. In order to implement this, a primitive function to test whether a term is a meta variable must be added to the general system.

However, the comparison will not be complete. Assume e.g. we have the global definitions

$$\texttt{p a} \equiv \texttt{q} \qquad \texttt{p b} \equiv \texttt{r}$$

for the constant $\texttt{p}$, and the equality constraints

$$?_1[\texttt{a}/x] = \texttt{q} \qquad ?_1[\texttt{b}/x] = \texttt{r},$$

where the brackets represent postponed variable substitutions. Then the solution, $?_1 := \texttt{p}\ x$, will not be found using the approach above. However, the definition of $\texttt{p}$ is rather artificial. This way of allowing information to migrate from one side to the other in equality constraints seems to be sufficient in most practical cases.

6 Experiments

We have performed some experiments with the presented approach. The implementation consists of a general narrowing search system and a type checker for a logical framework written in Haskell. The narrowing system is constituted by a compiler of Haskell programs and a run-time system implementing the search algorithm. The narrowing algorithm accepts weakly orthogonal TRSs by implementing the parallel evaluation discussed in section 4.1 and presented in more detail in [10]. It also includes a prototype of the subproblem separation feature presented in section 4.2. The type checker is based on the one presented in section 3, but modified to enable parallel conjunction and the performance enhancing features presented

in section 5. The implemented logical framework has recursive global function and data definitions.

The examples we have run the resulting proof construction tool on mainly focus on the way instantiation is scheduled when constructing proof terms containing type arguments. Performance-wise the approach cannot compete with more refined proof search methods like focused derivations or higher-order tabling (see section 2). The redundancy quickly becomes overwhelming for propositional problems and problems involving equality reasoning. In the following subsections a couple of examples are presented, illustrating how the presented modifications of the type checker influence the search. In connection with these examples some further remarks on the limitations of the approach are made.

6.1 The scheduling of instantiations

An inductive proof containing some kind of generalization serves well as an illustration of the order of instantiation which is imposed by the prioritization presented in section 5.2. The example is to construct a proof of the proposition stating that a given function, `sort`, always returns a sorted list.

$$?_1 : (xs : \texttt{List Nat}) \to \texttt{sorted}\ (\texttt{sort}\ xs)$$

In order to save space we will omit the definitions of `sorted` and `sort`, but merely state that `sort` implements insertion sort. The reader thus cannot confirm that the proof is correct. Nevertheless, the mechanisms of the proof search should be clear. The function `sort` is defined in terms of $\texttt{sort}'$ which, in turn, uses `insert`. The names `List` and `Nat` refer to the standard recursive definitions of lists and natural numbers. The proof search is provided an elimination constant for lists,

$$\texttt{elimList} : (X : \texttt{Set}) \to (xs : \texttt{List}\ X) \to (P : \texttt{List}\ X \to \texttt{Set}) \to$$
$$P\ \texttt{nil} \to ((z : X) \to (zs : \texttt{List}\ X) \to P\ zs \to P\ (\texttt{cons}\ z\ zs)) \to P\ xs,$$

and the lemma

$$\texttt{lem} : (x : \texttt{Nat}) \to (xs : \texttt{List Nat}) \to \texttt{sorted}\ xs \to \texttt{sorted}\ (\texttt{insert}\ x\ xs).$$

The sub-term `sort` xs in the given problem normalizes to $\texttt{sort}'\ xs\ \texttt{nil}$. A solution to the problem is

$$\begin{aligned}
?_1 := \lambda x \to \texttt{elimList}\ &\underbrace{\texttt{Nat}}_{B}\ \underbrace{x}_{A} \\
&(\lambda y \to (z : \underbrace{\texttt{List Nat}}_{B}) \to \underbrace{\texttt{sorted}\ z}_{E} \to \underbrace{\texttt{sorted}\ (\texttt{sort}'\ y\ z)}_{A}) \\
&\underbrace{(\lambda y \to \lambda z \to z)}_{C} \\
&\underbrace{(\lambda y \to \lambda z \to \lambda w \to \lambda t \to \lambda u \to w\ (\texttt{insert}\ y\ t)\ (\texttt{lem}\ y\ t\ u))}_{D} \\
&\underbrace{\texttt{nil}}_{A}\ \underbrace{\texttt{tt}}_{F}
\end{aligned}$$

where `tt` is the proof of the trivial problem, proving `sorted nil`.

Our implementation constructs the proof term above in the following way. First the λ-abstraction and application of `elimList` with the correct number of arguments are constructed. This also includes constructing the λ-abstraction and the two function arrows in the third argument of the application. A that stage there are a number of type checking constraints and one equality constraint involving the sub-terms marked A. The prioritization presented in section 5.2 makes the instantiation address the equality constraint first. This results in constructing the sub-terms A, followed by the terms marked B. Next, the construction of C and D are interleaved. When the elimination of z has been introduced in C, the resulting equality constraint triggers the construction of the extra hypothesis E. After that the type of F is becomes known, so its construction commences. During the final stage of the search the construction of F and the remaining of D proceeds as two independent subproblems, since they have no uninstantiated meta variables in common.

By employing parallel conjunction combined with the instantiation prioritization, the implementation can find the proof above within a second on a normal desktop computer. Without these features, the construction of this proof term by narrowing search is quite intractable.

6.2 Subproblem separation

To exemplify the improvement gained by introducing subproblem separation, which was discussed in section 4.2, we take the following example:

$$?_1 : (a : \texttt{Nat}) \to (b : \texttt{Nat}) \to \texttt{eq}\ a\ b \to \texttt{eq}\ b\ a$$

It involves a recursively defined equality relation over natural numbers. The solution of the problem includes nested induction, one at the top level and another induction in each of the base and step cases. The problem does not seem very difficult. But in spite of this it is a challenge to our implementation. The example makes it quite clear that more restricted induction is desirable in situations in which no strengthening of the induction hypothesis is required. However, when turning on the subproblem separation feature the base and step cases of the inductions can be solved independently. This makes the search space forty times smaller. The search only becomes a few times faster since our implementation of the feature is rather inefficient. However, we think that the implementation could be considerably improved. We also believe that efficient subproblem separation is essential if one is interested in making the whole approach tractable for more complex problems than those which have been presented here.

7 Conclusions

We have presented the idea of applying first-order narrowing on a type checker for a higher-order formalism in order to achieve proof construction. In order to make the resulting search procedure viable for practical use, we have refined the approach

by adding a couple of non-standard features of the narrowing, as well as a number of small and rather general modifications of the type checker. We have made an implementation to get some empirical evidence of the usability of the approach. The preliminary conclusion is that the approach could be useful in situations where low cost is important rather the high performance. The experiments are however too limited to be able to give a more solid conclusion/

For future work we are keen to further investigate the usefulness of the approach by more thouroughly running examples and comparing to other systems. One of the most crucial points in order to improve our system seems to be the subproblem separation feature. We would also like to investigate the addition of more restrictions to the type checker in order to impose heuristics for e.g. equality reasoning. An interesting direction is to look at using the approach for the purpose of generating functions. We have looked into this to some extent, and been able to synthesize e.g. insertion sort. But the main obstacle seems to be that, although we restrict the search to terminating functions, a lot of very inefficient candidates are still generated. This could be amended by adding some notion of function complexity to the type system.

References

[1] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[2] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[3] S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. Fourteenth International Conference on Logic Programming*, pages 138–152, Leuven, Belgium, July 1997. MIT Press.

[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[5] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722, pages 335–350, Tsukuba, Japan, 11 1999. Springer LNCS.

[6] Lennart Augustsson. Djinn, a theorem prover in haskell, for haskell. `http://www.augustsson.net/Darcs/Djinn/`

[7] James Cheney. `http://homepages.inf.ed.ac.uk/jcheney/publications/wmm06-draft.pdf`

[8] Gilles Dowek. A complete proof synthesis method for the cube of type systems. *J. Logic and Computation*, 3(3):287–315, 1993.

[9] M. Hanus and P. Réty. Demand-driven search in functional logic programs. Research report rr-lifo-98-08, Univ. Orléans, 1998.

[10] Fredrik Lindblad. Property directed generation of first-order test data. Presented at TFP 2007 and submitted for publication.

[11] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[12] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[13] Brigitte Pientka. Tabling for higher-order logic programming. In *CADE*, pages 54–68, 2005.

[14] Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Fakultät für Informatik, Universität Ulm, 1999.

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