Tactic Theorem Proving with Refinement-Tree Proofs and Metavariables*

Amy Felty and Douglas Howe

AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, USA

Abstract. This paper describes a prototype of a programmable interactive theorem-proving system. The main new feature of this system is that it supports the construction and manipulation of tree-structured proofs that can contain both metavariables and derived rules that are computed by tactic programs. The proof structure encapsulates the top-down refinement process of proof construction typical of most interactive theorem provers. Our prototype has been implemented in the logic programming language λProlog, from which we inherit a general kind of higher-order metavariable. Backing up, or undoing, of proof construction steps is supported by solving unification and matching constraints.

1 Introduction

Interactive proof construction typically proceeds top-down, starting with the statement of the theorem to be proven, and then successively refining goals into subgoals. This pattern is characteristic of most interactive systems, although there are large differences in the kinds of refinement that can be performed and in the underlying ideas of proof and state of the system. Our focus is on refinement by *tactics*. Loosely speaking, a tactic is a program that reduces a goal (typically a sequent) to a sequence of subgoals such that the goal can be inferred from the subgoals.

The set of refinements used to construct a proof has a natural tree structure. A system that directly supports this structure can provide a number of useful operations for building, manipulating and reading proofs. For example, subproofs of interest, or the steps leading up to a particular subgoal, can be quickly located via user-directed navigation of the tree. Bad proof strategies can be corrected by locating the first bad step in the tree, then removing just the part of the argument that depended on the step. The inferences in a proof can be replayed when, for example, a definition is changed, allowing one to recover all the portions not affected by the change. Proofs can be cut-and-pasted, and used for reasoning by analogy. [2] contains more examples of such operations.

Metavariables are supported in a number of existing systems [8, 3, 13, 10, 9, 4]. There are many compelling examples of their usefulness in interactive proofs. A common example is "existential introduction". Removing the existential quantifier in the goal $\vdash \exists\, x.\ p(x)$ requires a witness. We can use a metavariable X to stand for the witness term, refining the goal to the subgoal $\vdash p(X)$. We can instantiate

* To appear in *Proceedings of the 12th International Conference on Automated Deduction.*

X to a specific term later in the proof, at which time it will be replaced wherever it occurs. The point of postponing choice of the witness is that the details of the proof of $p(X)$ may make it possible to automatically find an instantiation of X, or at least make it easier for the user to determine it. For example, $p(X)$ may be an equation that can be manipulated to present an explicit solution $X = t$ for X.

There are many similar examples. Metavariables can be used to stand for programs to be synthesized from constructive proofs [11, 13], for induction hypotheses (or program invariants, as in [8]) that can be strengthened as proof progresses, and for "don't care" arguments to functions, such as type arguments or domain-membership evidence for constructive partial functions, that can be automatically deduced in a uniform way. Also, various procedures for automating reasoning make essential use of metavariables, and some of these procedures can be more usefully integrated with an interactive system if the system supports metavariables.

We have designed and implemented a theorem-proving system for constructing and manipulating refinement-tree proofs with metavariables. Proofs are trees of goals where each node g has associated with it a "justification" which specifies how the goal g can be inferred from its children. This justification can be a rule (name), in which case g and the children of g must be the conclusion and premises, respectively, of an instance of the rule, or it can be a representation T of a tactic.

A fundamental problem is to explain how T relates to g and its children $g_1, \ldots, g_n$. If we ignore metavariables, we can take T to be a program which, when applied to g, produces $g_1, \ldots, g_n$. If we allow metavariables, then this explanation no longer suffices. Suppose that after this tactic refinement was done, some metavariable instantiations were made, resulting in instances g', $g'_1, \ldots, g'_n$ of g and $g_1, \ldots, g_n$. What should be the relationship of T to g', $g'_1, \ldots, g'_n$?

One possibility is to restrict the tactic language so that we are guaranteed that the relationship above continues to hold: T applied to g' produces subgoals $g'_1, \ldots, g'_n$. We reject this possibility for two main reasons. First, although a *pure* logic programming language would preserve this relationship, we want to allow the use of a more practical programming language for tactic programming in order to enable users to achieve a high degree of automatic support for proof construction. For example, we would like to be able to accommodate the full λProlog language including its non-logical operators, and also the functional language ML, which has been proven well-suited to tactic programming.

Second, this property to be guaranteed is not respected by many commonplace and useful tactics. For example, suppose that we have a goal $X + 0 > 0 \vdash \phi(X)$, where X is a metavariable we intend to instantiate in another part of the proof, and the refinement step we wish to make is to simplify the goal using a tactic `Simplify`, say, that repeatedly applies common arithmetic rewrite rules. Assume the subgoal produced by `Simplify` is $X > 0 \vdash \phi(X)$. If, later during the proof, X gets instantiated to $1 + 0$, then the subgoal becomes $1 + 0 > 0 \vdash \phi(1 + 0)$, yet if `Simplify` were re-executed on the new goal, it would produce the subgoal $1 > 0 \vdash \phi(1)$. This example illustrates another problem with using a pure logic programming language for tactics: we do not want simplification to instantiate X, but instead to treat it as a constant. In general, we want some tactics to instantiate

metavariables, and others to treat them as constants.

One of the design goals for our system is programmability. We want to support programming of a wide range of procedures and tools for automating proof construction. We have therefore designed a simple proof structure with a small set of simple basic operations for constructing proofs. In particular, we have tried to avoid including in a proof information pertaining to the history of its construction, such as a record of metavariable instantiations. Section 2 gives a slightly simplified account of our approach to proofs. Section 5 gives some implementation details.

An important operation in interactive theorem provers is "undoing" parts of a proof attempt. Metavariables cause serious complications for this operation.

Without metavariables, subproofs P_1 and P_2 with no nodes in common are independent, and can be treated as completely separate tasks. However, metavariables can introduce dependencies between the subproofs. Suppose that a step in building P_1 involves the instantiation of a logic variable X that is mentioned in P_2. The instantiation replaces all occurrences of X in P_2, and subsequent steps in building P_2 may take advantage of this change. If we now decide that P_1 was a bad proof attempt, we could just remove it from the main proof and start a new subproof. But this would not completely undo the effect of the bad subproof P_1. We might also want to find and remove the parts of P_2 that depended on X's instantiation, and, in addition, uninstantiate any of the variables instantiated by refinements within these parts.

One kind of "undo" is directly inherited from our implementation language λProlog (see [4] for details). This is "chronological undo", which undoes proof modifications in the order they were performed. This is a useful form of undo, especially for quick local backups, and is the only form of undo in all other systems with metavariables except ALF [10]. However, it is unsatisfactory in general since it will often remove parts of the proof which are independent of the targeted parts.

We have designed a number of undo procedures that improve on chronological undo and have implemented one of them. These procedures are based on solving sequences of higher-order matching and unification problems. In section 3 we give an example session with our system that illustrates some of the problems and issues for designing undo operations for this kind of proof. Section 4 describes the undo procedure, and Section 5 gives some details of its implementation.

Our prototype has been implemented in the higher-order logic programming language λProlog [12], from which we inherit a general kind of higher-order metavariable, similar to what is found in Isabelle [13]. Our proof structure is inspired by Nuprl [2, 1]. The system is generic, in the sense that it can easily accommodate most logics that can be specified in the general style of LF [7] and Isabelle [13]. The exact style of encoding of logics in our system is very similar to [4], except that we have made a commitment to sequent-calculus presentations.

Although we have based our prototype on λProlog, the ideas are applicable to tactic systems based on ML such as LCF [6], Nuprl [2] and HOL [5]. Implementing the ideas for such systems would be straightforward, although it would be much more work because ML lacks λProlog's built-in support for metavariables, bound variables and unification.

The conclusion of the paper has a few additional comparisons with related work and a brief discussion of future work.

2 Proofs

This section gives a simplified account of our proof data-structure. Two of the principle constituents of proofs are *goals* and *tactics*. Goals are intended to be sequents in the logic being implemented (the "object logic"), and tactics are programs from some programming language. To make the presentation simple, and to keep the description here close to what has been implemented, we assume that both goals and tactics are represented as terms in the simply-typed λ-calculus. In our implementation, object logics are encoded as λ-terms and tactics are programs in λProlog, a language whose programs are all λ-terms.

There are a number of differences between the implementation and what is described here, mostly for reasons of efficiency and ease of tactic programming. Some of the more important differences are given at the end of the section.

Let Λ be the set of terms of the typed λ-calculus over some set of base types and some set of constants. We identify $\alpha\beta\eta$-equal terms. Thus, when we say that there is a substitution σ such that $\sigma(e) = \sigma(e')$, we mean that e and e' have a higher-order unifier. We distinguish a base type and define the set G of *goals* to be the set of all terms in Λ of this type.

Metavariables in this setting are simply the free variables of a goal. We will use capital letters to stand for metavariables. Ordinary variables in our representation are bound by λ-abstractions. See the implementation section for details on this representation.

A *proof* is a tree of goals where each node has associated with it a *justification*. The justification says how the goal can be inferred from its children. We will specify what the justifications are below, but for now, assume that for any justification j there is an associated pair $s(j) = (g, \overline{g})$, called a *step*, where g is a goal and $\overline{g}$ is a sequence $g_1, \ldots, g_n$ $(n \geq 0)$ of goals. We place the following restriction on proofs. Let g be a node in the tree, let j be its associated justification, and let $\overline{g}$ be the sequence of children of g (in left-to-right order). Write $s(j) = (g', \overline{g}')$. We require that there be a substitution σ such that $\sigma(g') = g$ and $\sigma(\overline{g}') = \overline{g}$ (using the obvious extension of substitution to sequences of goals).

Thus $s(j)$ can be thought of as a rule schema, with premises $\overline{g}'$ and conclusion g', and the valid instances of the schema are obtained by substituting for metavariables. For example, one of the allowed justifications in our implementation of first-order logic is the constant *and_i* (for "and-introduction"), and

$$s(\mathit{and\_i}) \quad = \quad (H \vdash A \,\&\, B,\ \ (H \vdash A,\ H \vdash B)),$$

which corresponds to the rule schema

$$\frac{H \vdash A \quad H \vdash B}{H \vdash A \,\&\, B.}$$

Note that proof trees are preserved under *instantiation:* if σ is a substitution, p is a proof, and $\sigma(p)$ is obtained by replacing every goal g in p by $\sigma(g)$ (and keeping the same associated justifications), then $\sigma(p)$ is a proof.

There are three kinds of justifications. One corresponds to primitive rules of the object logic and one to tactics. There is also a justification j_{prem}, where

$$s(j_{prem}) = (G, ()).$$

This corresponds to a trivial "rule" which infers any goal from no premises. Goals in a proof tree that have j_{prem} as their justification are called *premises* of the proof. Thus proof trees represent incomplete proofs in the object logic. The root goal of a proof is called its *conclusion.*

An important operation on proofs is *refinement*, where a justification is used to extend a proof tree at a premise. In particular, let g be a premise of a proof p, let j be a justification with $s(j) = (g', (g'_1, \ldots, g'_n))$, and let σ and σ' be substitutions such that $\sigma(g) = \sigma'(g')$.Then the *refinement of p at g using j, σ and σ'* is obtained from $\sigma(p)$ by giving j to the premise $\sigma(g)$ as a new justification, and adding children $\sigma'(g'_1), \ldots, \sigma'(g'_n)$, each with j_{prem} as its justification. For example, if g is $\vdash \phi \;\&\; \psi$, σ' substitutes ϕ for A and ψ for B, and $\sigma = \emptyset$, then refining at g using *and_i*, σ and σ' adds children $\vdash \phi$ and $\vdash \psi$ to g.

Note that the problem of finding some σ and σ', given g and j, can be cast as a higher-order unification problem. Let τ be a substitution that renames metavariables in g' such that $\tau(g')$ and g have no metavariables in common. Then σ, σ' exist if and only if g and $\tau(g')$ are unifiable.

Justifications corresponding to inference rules of the object logic are represented as a subset R of Λ. (Typically $r \in R$ will be a constant.) For each $r \in R$ there is an associated step $s(r)$. The justifications corresponding to tactics have a number of components. These will be explained by considering the typical kind of refinement: extending a proof by applying a tactic to a premise.

Tactics are represented as a subset T of Λ. Applying a tactic $e \in T$ to a premise g of a proof p is a single operation for a user of our theorem-prover, but it actually consists of first obtaining a justification and substitution from e and g, and then using these to perform a refinement step. More precisely, first the tactic e is evaluated with argument g, producing as its value a step $s' = (g', \overline{g}')$ and a substitution σ such that $\sigma(g) = g'$. Then the *tactic justification* $j = (e, g, \sigma, s')$ is formed, where $s(j)$ is defined to be s'. g is called the *tactic argument* of j. Finally, we refine p at g using j, σ and $\emptyset$. Thus, applying e to g produces a substitution to be applied to p, new children $\overline{g}'$ for the premise being refined, and a justification for the refinement.

For example, if g is $\vdash X = 0$ and if e is a tactic that instantiates X with 0, we might have $\sigma(X) = 0$, $\vdash 0 = 0$ for g', and $(\vdash 0 = 0)$ for $\overline{g}'$. The act of refining by j and σ will replace X by 0 in the entire proof, and produce a new premise $\vdash 0 = 0$ as a child of the old premise.

Note that occurrences of tactic justifications in a proof need not have arisen from the process just described, since the constraint on a justification j is only in terms of the step $s(j)$ associated with it.

There are several reasons for including more information in a tactic justification than just its step: e and g are informative to a user, since they determine the step; e

is needed for "replaying" a proof; and g and σ are required for the undo procedure. These last two points are discussed in the section on undo. The step is stored in the justification for reasons of efficiency, so that it does not have to be recomputed from e and g every time it is needed.

Some of the more important differences between the implementation and what is described above are as follows. In the account here, free variables correspond to metavariables, and free variables of the object logic are assumed to be handled by λ-abstraction in goals. In the implementation, these abstractions are handled more conveniently by distributing them throughout proofs, so there is only one λ-abstraction introduced whenever a new free variable of the object logic is needed.

In the implementation, the logic variables of λProlog are used for metavariables. This means that when, for example, we form a tactic justification, we cannot simply directly store the tactic argument in the justification, since then subsequent instantiations of metavariables in the proof might change it. So, when a tactic justification is created, the components have their logic variables "abstracted out", *i.e.* bound by λ-abstractions, to prevent them from being instantiated by further proof operations. More is said about this abstraction operation in Section 5.1.

Tactics work somewhat differently in the implementation. Since we are using a logic-programming language, and logic variables are used to implement metavariables, tactics do not need to explicitly return a substitution. To guarantee soundness, tactics produce proofs as results, from which the step is obtained as the conclusion and premises of the proof. These proofs can again contain tactic justifications. This introduces a circularity, but this is not hard to deal with (see [1] for one approach).

3 An Example Session

The following session illustrates interaction with our system, including refinement commands, navigation within a proof, and undo. We prove a simple formula from first-order logic. The tactics used here implement the basic inference rules of a sequent calculus. The lines beginning with "`!:`" indicate user input. All other text is output from the system (we have added some whitespace and renamed a few variables). All metavariables are printed as capital letters.

We begin the session by entering the following query to λProlog.

```
prove (exists x\ (exists y\ ( (q x imp q y) and
        (q x imp (q a or q b)) and (q a imp q b imp q x) and
        (q a imp exists z\ (q z)) and (q b imp q x)))).
```

This results in a prompt for input. The user supplies the command to run the tactic `(repeat intro)` which repeatedly applies some of the introduction rules for connectives occurring in formulas on the right in the sequent, resulting here in five subgoals.

```
!: tactic (repeat intro).
Address:
|- exists x\ (exists y\ ((q x imp q y) and
         (q x imp q a or q b) and (q a imp q b imp q x) and
```

```
                (q a imp exists z\ (q z)) and (q b imp q x)))
By tactic repeat intro
q X |- q Y
q X |- q a or q b
q b, q a |- q X
q a |- q Z
q b |- q X
```

After running the tactic, the node of the proof is redisplayed. The output above shows the address of the node of the proof being displayed (a list of integers, empty in this case), followed by the goal at the node, its justification following the word By, and the subgoals of the node.

The user then solves these subgoals in left-to-right order using the next command to go to the next premise node in the tree.

```
!: next.
Address: 1      q X |- q Y          By ?
!: tactic (hyp 1).
Address: 1      q Y |- q Y          By tactic hyp 1
!: next.
Address: 2      q Y |- q a or q b   By ?
!: tactic (then or_i1_tac (hyp 1)).
Address: 2      q a |- q a or q b   By tactic then or_i1_tac (hyp 1)
!: next.
Address: 3      q b, q a |- q a     By ?
!: tactic (hyp 2).
Address: 3      q b, q a |- q a     By tactic hyp 2
!: next.
Address: 4      q a |- q Z          By ?
!: tactic (hyp 1).
Address: 4      q a |- q a          By tactic hyp 1
!: next.
Address: 5      q b |- q a          By ?
```

The hyp tactic completes a proof when the formula on the right unifies with the formula on the left at the position indicated by an integer argument. In the first, second, and fourth subgoals above, this operation causes instantiation of metavariables: X to Y, Y to a, and Z to a, respectively. Note that or_i1_tac in subgoal 2 chooses the first disjunct of the conclusion which, when followed by hyp 1, through the use of the tactic combinator then, forces the instantiation of Y to a.

Now no further progress can be made. Looking at subgoal 5 as it originally appeared above, the user realizes that X must be instantiated to b if this subgoal is to be provable. By comparing to the 5 subgoals above, we see that it was subgoal 2 that forced the instantiation of a. This can be verified by moving through the tree to the second child of the root, and then using the show_tactic_arg command to examine the goal which was the original argument to the tactic at that node. Entering the uninst command there undoes the instantiation of the variables in this sequent and any nodes that depended on them, in this case just Y. We omit the printing of the root node.

```
!: root.      !: down 2.
Address: 2       q a |- q a or q b   By tactic then or_i1_tac (hyp 1)
!: show_tactic_arg.
q Y |- q a or q b
!: uninst.
Address: 2       q Y1 |- q a or q b  By ?
```

By going back to the root and printing the proof, we see the refinements at subgoals 2 and 3 have been undone, since they both depended on the instantiation of `Y` to `a`. The proof of the first subgoal which originally unified two logic variables remains, as does the proof of subgoal 4.

```
!: root.      !: show_proof.
Goal: |- exists x\ [...]   Justification: By tactic repeat intro
  Goal: q Y1 |- q Y1          Justification: By tactic hyp 1
  Goal: q Y1 |- q a or q b    Justification: By ?
  Goal: q b, q a |- q Y1      Justification: By ?
  Goal: q a |- q a            Justification: By tactic hyp 1
  Goal: q b |- q Y1           Justification: By ?
```

The remaining subgoals can be completed by choosing the second disjunct in subgoal 2 forcing the instantiation of metavariable `Y1` to `b`.

4 Undoing Proofs

In the implemented undo procedure (`uninst`), we want to back the proof up enough so that for some node, called the *undo node*, the goal that appeared at that node before a tactic refinement was applied becomes the goal at that node in the new proof. The procedure removes any parts of the proof which would force metavariables in the undo node to be instantiated. Furthermore, it undoes any instantiations forced by the removed subproofs.

For our undo procedure, we assume that all non-premise nodes in a proof have tactic justifications. Below, we will speak of a node's associated tactic, tactic argument, substitution, and step, corresponding to the four components of its justification. In order to not have to worry about renaming of free variables, we assume (without loss of generality) that justifications from different nodes in a proof have no free variables in common. However, it is not necessary to modify a proof to meet this criteria in order to run the undo procedure. Instead, at each step, fresh variables could be introduced to rename the components of a justification.

The procedure works by finding an ordering on the nodes in the proof that corresponds to a possible method of building the tree by using alternating instantiations and tactic-refinement steps. The ordering is used to track "when" variables become instantiated in order to determine which branches to prune. To reconstruct the new tree, the appropriate branches are pruned from the original tree. Then, the same procedure used to find the ordering on the original tree is used to construct the new tree. We describe this core procedure and discuss how it is used in each of the two phases of the implemented `uninst` operation.

The main operation of the core procedure is a step-by-step reconstruction of the proof. At each step, we have the partially reconstructed tree p with a sequence of premises $g_1, \ldots, g_n$ called the *fringe*, that collects the premises that must still be expanded to obtain the complete tree. In addition, we have a mapping from fringe elements to the subproofs $p_1, \ldots, p_n$ in p rooted at the corresponding locations in the original proof. It is these subproofs that must be processed in order to complete the reconstruction. At any point, if we build a "justification tree" by taking the associated justifications of each node in $p, p_1, \ldots, p_n$ and attaching those for $p_1, \ldots, p_n$ at the appropriate nodes, we obtain the same justification tree as from the original proof.

At each step, we choose for "expansion" an element g_i of the fringe that satisfies the following two requirements. First, the tactic argument at the root node r of p_i must be an instance of g_i. Let σ be the matching substitution. Second, for every other fringe element g' mapped to subproof q, the tactic argument at the root of q must be an instance of $\sigma(g')$. Let j be the justification at node r and let τ be the substitution component of j. We refine p at g_i using j, $\sigma \circ \tau$ and the empty substitution. Thus, $\tau(\sigma(g_i))$ now replaces g_i in the new proof. We then replace g_i in the fringe by the new premises added by the refinement. The mapping is extended by mapping these elements to the corresponding subproofs rooted at the children of r.

A proof p' is obtained by *pruning* from p if it results from removing some subproofs from p (leaving new premises). p' is a *re-instantiation* of p if the two proofs have the same justification tree. Using an inductive argument, it can be shown that the set of proofs for which the above reconstruction procedure can be applied is closed under pruning, instantiation, re-instantiation, and refinement by tactics. An important invariant during reconstruction of any proof built only from these operations is that there will always be some fringe node that meets the two requirements for expansion. The ordering of nodes by relative time that they were expanded by tactic refinement, for example, can be used in reconstruction of the proof, and each successive node will meet the requirements. There may be other possible orderings, all of which correspond to possible orderings in which the tree could have been constructed. We call such an ordering a *refinement ordering*.

The proof reconstruction procedure is repeated twice in the `uninst` operation. The first time, we keep track of two kinds of information: a set of variables $\mathcal{V}$ that must remain uninstantiated and a set of addresses $\mathcal{A}$ of nodes that roots of subtrees that must be pruned. Initially, $\mathcal{V}$ is the set containing all of the free variables in the tactic argument of the undo node, and $\mathcal{A}$ is empty. At each step, after a fringe node g is chosen for expansion, $\mathcal{V}$ is updated. In particular, the match substitution σ is checked to see if it maps any of the variables in $\mathcal{V}$ to new variables. These variables must also be added to $\mathcal{V}$. After each refinement step, $\mathcal{A}$ is updated. Any nodes containing metavariables in $\mathcal{V}$ that get instantiated by this refinement step must be marked for deletion. More precisely, let τ be the substitution at the root node of the proof that g is mapped to. Let $\mathcal{V}'$ be the set of all variables X in $\mathcal{V}$ such that either $\tau(\sigma(X))$ is not a variable or there is a Y in $\mathcal{V}$ with $X \neq Y$ and $\tau(\sigma(X)) = \tau(\sigma(Y))$. If $\mathcal{V}'$ is non-empty, then the address of the node just expanded

is added to $\mathcal{A}$. In addition, the address of any fringe node that contains a variable in $\mathcal{V}'$ is also added to $\mathcal{A}$. Note that no addresses will get added to $\mathcal{A}$ until the point in reconstruction after the undo node is expanded.

After the reconstruction is complete, the subproofs indicated by $\mathcal{A}$ are pruned, resulting in a new proof q. Now, the reconstruction procedure is repeated on q. This time $\mathcal{V}$ and $\mathcal{A}$ are ignored. The tree produced by this phase is a minimally instantiated version of the justification tree. It is this proof that is returned from the `uninst` operation.

A very slight modification of this procedure gives us an operation that allows the user to point at specific variables in one or more tactic arguments and ask that they remain uninstantiated in the remaining tree. This can be achieved by initializing $\mathcal{V}$ to be the selected variables only. Multiple variables from different nodes can be processed simultaneously by including them all in $\mathcal{V}$.

A slightly more complicated operation is to request that all of the variables instantiated by a particular tactic refinement are backed up along with all branches that saw these instantiations. Here the substitution at the justification of the undo node is used. Any variables that are mapped to anything other than themselves should be put into $\mathcal{V}$. This operation can be extended to include all variables instantiated in a particular subproof. Here, in addition to any variable from the substitution at the undo node, any variable from the substitution at any of its descendants that is mapped to something other than itself must be included in $\mathcal{V}$.

The operations discussed so far prune any branches that depend on instantiation of the selected variables. Another option is to attempt to replay them using the tactics in justifications. Such replay can be accommodated by introducing a new *bad* justification. Then instead of pruning, a tactic justification is changed to a bad one which retains the tactic and the structure of the tree below it. A final phase of the procedure would then attempt to replay as much of the bad proof by re-executing the tactics, only pruning when execution fails. Alternatively, the "bad" subproofs could be left to the user.

All of these undo operations work on proofs closed under tactic refinement, pruning, instantiation, and re-instantiation. We can extend the proof reconstruction procedure to handle a larger class of proofs including those pieced together through unification. However, the notion of tracking "when" a variable is instantiated no longer works in the same way. In particular there is not necessarily a refinement ordering because any two proofs pieced together may have been constructed independently. We define a heuristic for determining an order. This procedure is a simple modification of the above procedure. In the case when no node in the fringe meets the two requirements for expansion, we find a node g in the fringe maximizing a particular measure. Let g' be the conclusion of the step at the root r of the subproof that g is mapped to and let σ be a substitution such that $\sigma(g) = \sigma(g')$. That is, instead of matching with the tactic argument, we require unification with the step conclusion. This unification problem is in fact solvable for all nodes in the fringe. We now need a measure to determine when one ordering of nodes is better than another. We can use a measure that favors instantiations that are done via tactic refinement over those done simply to match up the goal of a node with the

corresponding premise of a step in the parent node. The measure should have the property that any refinement ordering maximizes it. In the worst case, this heuristic reduces to reconstructing the tree by repeatedly solving unification problems in arbitrary order. In the best case, it finds an ordering corresponding to the order in which the proof could have been constructed for those trees where such an ordering exists.

5 Implementation

This section describes the implementation of our system. It starts with a brief account of the implementation language.

5.1 λProlog

λProlog is a partial implementation of higher-order hereditary Harrop (hohh) formulas [12] which extend positive Horn clauses in essentially two ways. First, they allow implication and universal quantification in the bodies of clauses, in addition to conjunctions, disjunctions, and existentially quantified formulas. In this paper, we only consider the extension to universal quantification. Second, they replace first-order terms with the more expressive simply typed λ-terms and allow quantification over predicate and function symbols. The application of λ-terms is handled by β-conversion, while the unification of λ-terms is handled by higher-order unification.

The terms of the language are the terms of Λ where the set of base types includes at least the type symbol `o`, which denotes the type of logic programming propositions. In this section, we adopt the syntax of λProlog. Free variables are represented by tokens with an upper case initial letter and constants are represented by tokens with a lower case initial letter. Bound variables can begin with either an upper or lower case letter. λ-abstraction is represented using backslash as an infix symbol.

Logical connectives and quantifiers are introduced into λ-terms by introducing suitable constants with their types. In particular, we introduce constants for conjunction (`,`), disjunctions (`;`), and (reverse) implication (`:-`) having type `o -> o -> o`. The constant for universal quantification (`pi`) is given type `(A -> o) -> o` for each type replacing the "type variable" `A`. A function symbol whose target type is `o`, other than a logical constant, will be considered a *predicate*. A λ-term of type `o` such that the head of its $\beta\eta$-long form is not a logical constant will be called an *atomic formula*. A *goal* is a formula that does not contain implication. A *clause* is a formula of the form (`pi` x_1`\` $\ldots$ (`pi` x_n`\(`A`:-` G`))) where $G$ is a goal formula and $A$ is an atomic formula with a constant as its head. In presenting clauses, we leave off outermost universal quantifiers, and write (`(`$A$`:-` $G$`)`).

Search in λProlog is similar to that in Prolog. Universal quantification in goals (`pi x\`G) is implemented by introducing a new parameter `c` and trying to prove [`c/x`]G. Unification is restricted so that if G contains logic variables, the new constant `c` will not appear in the terms eventually instantiated for those logic variables.

Several non-logical features of λProlog are used in our implementation. We use the cut (`!`) operator to eliminate backtracking points. In addition, we have implemented a new primitive `make_abs` which takes any term and replaces all logic

variables with λ-bindings at the top-level. It has type `A -> abs A -> list mvar -> o` where `abs` is a type constructor introduced for this purpose and the third argument is a list containing all of the logic variables in the order they occurred in the term. We use this operation to "freeze" the degree of instantiation of a term as well as to implement a `match` procedure. In order to correctly freeze a term, this operation must also freeze a record of any unification constraints on the logic variables occurring in the term. The current implementation does not do so. However, we have verified that our implementation does not generate constraints. We make the restriction that any programmer defined tactics also cannot generate constraints.

λProlog allows type constructors for building types. In addition to `abs`, we use `pair` and `list` in our implementation.

5.2 Proofs and Tactics

Below are the basic types and operations for our implementation of proofs.

```
goal            type.
agoal           (A -> goal) -> goal.
step            type.
step            goal -> list goal -> step.
prule_name      type.
prule_def       prule_name -> seq -> list seq -> o.
proof           type.
just            type.
prem_just       just -> o.
prule_just      prule_name -> just -> o.
tactic_to_just  (goal -> proof -> o) -> goal -> proof -> just -> o.
one_step_proof  step -> just -> proof -> o.
compose_proofs  proof -> list proof -> proof -> o.
aproof          (A -> proof) -> proof.
```

These are intended to form abstract data types for justifications and proofs. We have omitted several destructors for these types. All of our operations for building and modifying proofs do so via the above operations.

The type `goal` is the type of goals. Goals are essentially sequents. They also have some additional structure which we plan to exploit in future work. We represent hypothesis lists of sequents using function composition (as is done in Isabelle) so that higher-order unification can be used to deal with metavariables standing for subsequences of hypothesis lists. λ-abstracted sequents are also goals: the constructor `agoal` converts a term `x\(G x)` into a goal. The type `step` and the constructor `step` implement the steps of Section 2.

The object logic is assumed to be specified by a type `prule_name` whose members are the primitive rule names, and a predicate `prule_def` that associates a (sequent version of a) step with each rule name. For example, the following clauses specify the rules for and-introduction and all-introduction.

```
prule_def and_i (|- H (A and B)) ((|- H A)::(|- H B)::nil).
prule_def forall_i (|- H (forall A)) ((aseq (x\ (|- H (A x))))::nil).
```

Here |- is the constructor for basic sequents, and aseq constructs abstracted sequents.

There are three ways of building proofs. One is to use aproof to turn an abstracted proof into a proof. In a proof (aproof x\(P x)), the bound variable x represents a new object level variable whose scope is the proof (P x). The second way to build proofs is to construct a one-step proof from a step and a justification. (one_step_proof S J P) computes the step corresponding to the justification J, checks that it matches the step S=(step G Gs), then produces a proof whose root has goal G and justification J, and whose children are premises with goals from the list Gs. The premises may use the aproof constructor. This would be the case if, for example, J were the justification for the rule forall_i.

The final way to construct proofs is with compose_proofs which attaches the members of a list of proofs at the premises of another proof. This is used in the implementation of then, a combinator for sequencing tactics. Tactics are predicates of type goal -> proof -> o. Some care was taken with the composition operation in order to make tactics efficient. In particular, it produces a variant representation of a proof that delays actual computation of the composition. Usually the actual composition never needs to be performed, and when it does, it will usually be in the context of other delayed compositions, and grouped compositions can be handled much more efficiently.

The type just is an abstract type of justifications. prule_just constructs primitive rule justifications. Tactic justifications have four parts made up of two "abstracted" pairs with types (abs (pair (goal -> proof -> o) goal)) and (abs (pair (list mvar) step)), respectively. The four parts of this datatype implement the four parts of the tactic justification: the tactic, tactic argument, substitution, and step. Since metavariables are implemented directly using the logic variables of λProlog, and since we do not want variables in any of these components to be further instantiated, we use the the make_abs operation described earlier to "freeze" them. When a copy is needed, it is made by applying these abstractions to new logic variables. Instead of a set of variable/instance pairs, the substitution is represented as a list of terms such that the length of this list is the same as the length of the binder of the first pair. A substitution is applied by taking the list of new variables used to make a copy of the first abstraction and matching it against this stored list.

tactic_to_just takes a tactic T, runs it on a goal, returning the tactic's proof and the corresponding justification. Below is the main clause of its implementation. (The test that the last argument is a variable to ensure one-way behaviour is omitted here).

```
tactic_to_just T G P (trule AbsTacAp AbsSigmaStep)
:- make_abs (p T G) AbsTacAp Subs,
   T G P,
   concl P NewG,
   prems P Gs,
   make_abs (p Subs (step NewG Gs)) AbsSigmaStep Bazola.
```

trule is the hidden constructor for tactic justifications, p is a pairing constructor,

and `concl` and `prems` compute the conclusion and premises of a proof.

Although it is unlikely, it is possible for a user of our system to build objects of type `proof` that are not proofs. For maximum security, we would need to include some further run-time checks. In a language like ML, such security could be obtained through the type system.

5.3 Undo: the `uninst` Command

The implementation of `uninst` follows the description in Section 4 with a few optimizations. One such optimization comes from using λProlog's built-in unification for our match procedure. When checking the first requirement of a fringe element and determining the match substitution σ, for example, the match procedure can directly apply the result of the match to the metavariables of the goal, automatically propagating σ to the new proof.

At the refinement step of proof reconstruction, the application of a substitution is also propagated in the new proof by logic variable instantiation. Note that, in the refinement step, instead of using the substitution in the justification, we could simply match the goal to be refined with the conclusion of the step. In fact, we need not record substitutions at all in tactic justifications. However, they serve as an optimization, allowing the propagation of the exact substitution that was originally done by executing the tactic. In addition, because matching uses λProlog unification, by avoiding matching we also avoid generating unification constraints.

6 Discussion

Isabelle [13] and Coq [3] have metavariables and support tactic-style theorem-proving, but refinement trees are implicit. Operations on these trees are limited, and, in particular, undo is chronological. This also applies to KIV [8], even though it explicitly supports a form of refinement trees. In contrast to ALF [10] and Coq, our system only supports simple types for metavariables. If the object logic has a richer type system, then types must be represented explicitly, for example as predicates in the object logic. ALF supports dependency-directed undo, but proofs are λ-terms, not refinement trees.

Plans for future work include: improving the way types are handled; designing and implementing further undo operations that handle arbitrary proofs, *e.g.* proofs that are pieced together using unification; adapting Nuprl's scheme for compact storage of proofs in files; and implementing our ideas for Nuprl.

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. B. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic and Computer Science*, pages 95–107. IEEE Computer Society, June 1990.
2. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

3. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide. Technical Report 134, INRIA, December 1991.
4. A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
5. M. Gordon. A proof generating system for higher-order logic. In *Proceedings of the Hardware Verification Workshop*, 1989.
6. M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
7. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.
8. M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In M. Stickel, editor, *Tenth Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 117–131. Springer-Verlag, 1990.
9. C. Horn. *The Oyster Proof Development System*. University of Edinburgh, 1988.
10. L. Magnussan. Refinement and local undo in the interactive proof editor ALF. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, 1993.
11. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *Transactions on Programming Languages and Systems*, 2:90–121, 1980.
12. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
13. L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.