Generalization and Reuse of Tactic Proofs

Amy Felty and Douglas Howe

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

Abstract. A *tactic proof* is a tree-structured sequent proof where steps may be justified by tactic programs. We describe a prototype of a generic interactive theorem-proving system that supports the construction and manipulation of tactic proofs containing metavariables. The emphasis is on *proof reuse*. Examples of proof reuse are proof by analogy and reconstruction of partial proofs as part of recovering from errors in definitions or in proof strategies. Our reuse operations involve solving higher-order unification problems, and their effectiveness relies on a proof-generalization step that is done after a tactic is applied. The prototype is implemented in λProlog.

1 Introduction

Most interactive theorem proving systems support some notion of "high-level" proof, although this is usually informal. Often it is a textual transcript of the commands that were used to direct the prover. It can also be a composite tactic program formed by combining all the individual tactics used to prove a theorem. Informal or not, in practice such objects serve as explanations of formal proofs and as the basis for useful proof-manipulating operations.

One general way to directly support high-level proof objects is to extend tree-structured sequent proofs so that a step in a sequent proof can be justified either by an inference rule of the implemented logic, or by a program e whose execution produces a partial proof that justifies the inference, providing steps that lead from the antecedent sequents of the inference to the concluding sequent.

Because most interactive proofs are conducted top-down, we restrict the program e to consist of a program e', called a *tactic*, together with an input goal g. We call proofs where steps may be justified by tactics *tactic proofs*. If we also require that g be identical to the conclusion g' of the inference justified by a tactic, then this proof structure is the same as that of Nuprl [1]. We do not make this restriction because it is incompatible with the use of metavariables in proofs.

The focus of this paper is the reuse of tactic proofs. One obvious example of proof reuse is proof by analogy, where an existing proof is adapted to a new theorem. A more common application in interactive systems is in error recovery. During the course of a proof it is often necessary to make a change that affects the part of the proof that has already been built. Such a change might be to the statement of the goal, to a definition the proof depends on, to an induction hypothesis or to a previous step in the proof. The problem is to recover as much of the existing (partial) proof as possible after such a change.

One way to reuse proof steps after a change is to *replay* them. In the setting of tactic proofs, this can be done by reconstructing the proof tree by successively reapplying all the tactics in a top-down fashion.

There are two problems with this approach. First, it can be time consuming. Second, it is not a robust procedure for many of the kinds of local changes typical in error recovery. As a simple example, suppose a step in the proof used a tactic that just repeatedly applied &-introduction until no conjunctions remained in the succedent of the goal sequent. In particular, suppose the tactic refined the goal $\vdash \phi_1 \,\&\, \phi_2 \,\&\, \phi_3$ to the three subgoals $\vdash \phi_1$, $\vdash \phi_2$, and $\vdash \phi_3$.

Now suppose that ϕ_1 has been changed to $\phi_4 \,\&\, \phi_5$. When the replay of the steps in the original proof gets to the above tactic step, the goal is now $\vdash (\phi_4 \,\&\, \phi_5) \,\&\, \phi_2 \,\&\, \phi_3$. Rerunning the tactic produces subgoals $\vdash \phi_4$, $\vdash \phi_5$, $\vdash \phi_2$, and $\vdash \phi_3$. In order to continue replay, we need to know that the subproofs for the last two of the original three subgoals should be used for the last two of the new subgoals. In this simple case it is obvious how to make this association, but in general, when there can be other changes, it is not.

The basic idea underlying our solution to this problem is simple, and involves relaxing the requirement on how a tactic relates to the step it justifies. In the above example, the step is determined by the execution of the tactic: the tactic, when applied to the goal, produces a proof tree whose concluding sequent, or *conclusion*, is the goal, and whose non-axiomatic leaf sequents, or *premises*, are the subgoals. However, the proof produced by the tactic may justify many more inferences than the one that is determined this way. For example, if the proof is the obvious one involving just &-introduction rules, then the identity of the formulas ϕ_i is irrelevant.

We capture this kind of generality by performing proof generalization, making use of metavariables in proofs. Inference rules of the logic are implemented as schemas with metavariables in a manner very similar to what is found in Isabelle [10]. A rule schema justifies any inference which can be obtained by instantiating metavariables in the schema. When a tactic produces a proof using these rule schemas, we can generalize to find a "minimal" proof using these schemas. We then allow this tactic to justify any step corresponding to an instance of this minimal proof.

Consider again the &-introduction example. The generalization of the proof produced by the tactic has conclusion $H \vdash A \,\&\, B \,\&\, C$ and premises $H \vdash A$, $H \vdash B$, and $H \vdash C$. Instantiating H to the empty hypothesis list and A, B, C to ϕ_1, ϕ_2, ϕ_3 yields the original inference. When a conjunct is added as above, the same tactic step can be used by taking A to be $\phi_4 \,\&\, \phi_5$. The number of subgoals remains the same.

This example also shows a tradeoff we have made between generality and intelligibility. After the conjunction is added, in the step where repeated &-introduction is applied, there will be a subgoal where another &-introduction could be applied. In other words, it may appear as though the tactic did not progress as far as it might have, since if we reran the tactic on the new goal there would be four subgoals instead of three. The situation is worse when tactics explicitly refer to components of the sequent. For example, if the tactic is `Elim 1`, which applies &-elimination to the first hypothesis, then the generalized step is simply the &-elimination rule. This means that the proof could be modified so that the eliminated hypothesis is the second one. However, explicit references to hypothesis numbers is typically regarded as bad style since it makes proof replay much less robust. We believe that the &-introduction example is more typical. In any case, the original input goal for the tactic is retained, so when reading a proof it is still possible to precisely understand the connection between a tactic and the step it justifies.

We have implemented a prototype of an interactive theorem proving system based on these ideas. The system includes a command interface for building and viewing tactic proofs. We have also implemented a reuse command that attempts to reuse a proof when the statement of the root goal is changed.

λProlog [9] is both the implementation language and the tactic programming language in our system. Using ML as an implementation language would give us a greater degree of control over unification and metavariable instantiation, but a substantial amount of work would be required to implement λProlog's built-in support for metavariables, variable-binding and backtracking.

The design of our system is generic, in the sense that it can easily accommodate most logics that can be specified in the general style of LF [5]. The exact style of encoding of logics in our system is very similar to [3], except that we have made a commitment to sequent-calculus presentations.

The next section describes the structure of proofs in our system and gives a somewhat abstract account of proof generalization and reuse. This is followed by an example session with our prover, a section giving some implementation details and, finally, a conclusion discussing some extensions and giving a few further comparisons with related work.

2 Proofs

This section gives a simplified account of the proof data-structure. A few of the more important differences between this account and the implementation are given at the end of the section.

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 using λ-terms and tactics are programs in λProlog, a language whose programs are all λ-terms.

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, if there is a substitution σ such that $\sigma(e) = \sigma(e')$, then e and e' have a higher-order unifier. We distinguish a base type and define the set 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 describe the kinds of justifications below. 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. g is the *conclusion* of the step $(g, \overline{g})$, and g_i, $1 \leq i \leq n$, is the i^{th} *premise* of the step. Define $concl(s(j)) = g$ and $prem_i(s(j)) = g_i$, $1 \leq i \leq n$. 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 &-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 $s(j_{prem})$ is $(G, ())$. 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.*

Justifications corresponding to inference rules of the object logic are represented as a set $R \subset A$ of constants called *rule names.* For each $r \in R$ there is an associated step $s(r)$.

The third form of justification is the *tactic justification.* Tactics are represented as a subset T of A. A tactic is a program that takes a goal as argument and, if it produces an output, returns a proof and a substitution. For each goal g and tactic $e \in T$ such that e produces an output on input g, there is a justification $j_{e,g}$.

Let p be the proof returned by e on input g. One possibility for $s(j_{e,g})$ is to take the conclusion and premises of p [4]. Since proofs are closed under instantiation, any instance of this step is sound. However, this is restrictive. If there is a proof p' of which p is an instance, then the step derived from p' is also sound and is more general. So, we define $s(j_{e,g})$ to be a "minimal" p' having p as an instance. We call p' a *generalization* of p.

We can cast the problem of finding this generalization p' as a unification problem. For each node u in p, let j^u be its associated justification. Without loss of generality, we may assume that $s(j^u)$ and $s(j^v)$ have no metavariables in common when $u \neq v$. To find a minimal p', we find a minimal unifier σ such that

$$\sigma(prem_i(s(j^u)) = \sigma(concl(s(j^v))$$

for all nodes u and v of p such that v is the i^{th} child of u. We then obtain p' from p by replacing the goal at each node u by $\sigma(concl(s(j^u)))$.

In general, most-general higher-order unifiers do not exist, and so the definition of $s(j_{e,g})$ is under-specified. The implementation computes generalizations by solving the above unification problems. The exact choice of σ is implementation dependent.

The generalization step in the definition of $s(j_{e,g})$ is exploited in the procedure for proof reuse. This procedure reuses the existing justifications in a proof tree to justify a tree with different goals. The particular reuse procedure we have implemented is almost identical to the generalization procedure. The only substantial difference is that we also require that

$$\sigma(g) = \sigma(concl(s(j^r)))$$

where r is the root of the (sub-) proof we wish to reuse and g is the new root goal we want to reuse the proof for.

Proof trees are typically extended by *refining* a premise with a tactic. Tactic refinement works as follows. Let g be a premise of a proof p, and suppose that the tactic e with argument g returns a proof p' and a substitution σ. Let g' be the conclusion of p' and let $\overline{g}'$ be the sequence of premises of p'. If $g' = \sigma(g)$, then the *refinement of p at g using e* is obtained from $\sigma(p)$ by replacing the premise justification of $\sigma(g)$ by $j_{e,g}$, and adding, as children of $\sigma(g)$, nodes $g'_1, \ldots, g'_n$ each with j_{prem} as its justification.

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 will replace X by 0 in the entire proof, and produce a new premise $\vdash 0 = 0$ as a child of the old premise.

The most important difference between the implementation and the above account of tactics in proofs is that in the implementation, the logic variables of λProlog are used for metavariables. This means that tactics do not need to explicitly return a substitution; instantiation of metavariables is part of the execution of a tactic.

The justifications associated with the nodes of a proof are stored along with the goals in the implemented proof data structure. A tactic justification $j_{e,g}$ is implemented as a tuple containing the text of the program e along with the goal g. Also, to avoid having to recompute the application of e on g every time $s(j_{e,g})$ is needed, we also put into the tuple the *justification tree* that results from erasing the goals from the proof produced by e on input g. The justification tree is all that is needed to perform the generalization step for $s(j_{e,g})$. The tactic and its argument are retained in addition to the justification tree because they are informative to a user.

3 An Example Session

The following session illustrates interaction with our system, including refinement commands, navigation within a proof, as well as the `reuse` command. We attempt to prove the simple formula $(pb \vee qa) \;\&\; (pa \supset ra) \;\&\; (ra \supset qb) \supset \exists x\, qx$ from first-order logic. The example is contrived, though it illustrates the main operations. The tactics used here implement the basic inference rules of a sequent calculus. The lines beginning with "`!:`" are for user input. All metavariables are printed as capital letters.

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

```
prove ((p b or q a) and (p a imp r a) and (r a imp q b)
       imp (exists X\ (q X))).
```

This results in a prompt for input. The user supplies the commands to run the `intro` tactic and then `and_e*_tac` to eliminate the conjunctions in the hypothesis, using the `next` command in between to go to the next premise in the tree.

```
!: tactic intro.
Address:
|- (p b or q a) and (p a imp r a) and (r a imp q b)
    imp exists X\ (q X)
By tactic intro
(p b or q a) and (p a imp r a) and (r a imp q b) |- exists X\ (q X)

!: next.
Address: 1
(p b or q a) and (p a imp r a) and (r a imp q b) |- exists X\ (q X)
By ?

!: tactic and_e*_tac.
Address: 1
(p b or q a) and (p a imp r a) and (r a imp q b) |- exists X\ (q X)
By tactic and_e*_tac
(p b or q a) and (p a imp r a) and (r a imp q b), p b or q a,
(p a imp r a) and (r a imp q b), p a imp r a, r a imp q b
|- exists X\ (q X)
```

After running each 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 the first case), followed by the goal at the node, its justification, following the word `By` and the subgoals of the node.

In displaying the remainder of the session, we leave off the hypotheses that are not important, and sometimes leave out the output of navigation commands. Continuing the proof, the user applies a tactic that applies several inference rules.

```
!: next.
Address: 1 1
p b or q a, p a imp r a, r a imp q b |- exists X\ (q X)
By ?

!: tactic (then or_e_tac (then intro (try close_tac))).
Address: 1 1
p b or q a, p a imp r a, r a imp q b |- exists X\ (q X)
By tactic then or_e_tac (then intro (try close_tac))
p b, p a imp r a, r a imp q b |- q T
```

Here the proof was split into two branches by eliminating the disjunction and the second was completed. Note that introducing an existential quantifier introduces a metavariable T which can be filled in later.

At this point, the user realizes that the original goal had an error. The b in (p b) should have been an a. Our command language includes a `transform` command to transform one proof to another. It takes as an argument the transformation predicate. The `reuse` predicate copies the structure of the current proof to build a proof of the new formula.

```
!: root.
!: transform
     (reuse ((p a or q a) and (p a imp r a) and (r a imp q b))
              imp (exists X\ (q X)).
!: next.
Address: 1 1 1
p a, p a imp r a, r a imp q b |- q T
By ?
```

The operation succeeds and one subgoal remains. The proof is completed with the `(repeat backchain)` tactic.

```
!: tactic backchain.
Address: 1 1 1
p a, p a imp r a, r a imp q b |- q T
By tactic backchain
!: next.
No next premise: Proof Complete!
```

The application of `reuse` in this example works as follows. Let ϕ be the argument given to `reuse`. At the point `reuse` is applied, the current proof has four nodes, u_1, u_2, u_3 and u_4, with u_i the parent of u_{i+1}, $1 \leq i \leq 3$. Call the corresponding justifications j_i. j_1, j_2 and j_3 are tactic justifications, and u_4 is a premise. Let $s(j_i) = (g_i, (h_i))$, $1 \leq i \leq 3$. The reuse procedure uses λProlog's built-in unification to compute a unifier σ solving the equations

$$\phi = g_1, \quad h_1 = g_2, \quad h_2 = g_3, \quad h_3 = G.$$

The new proof has the same justifications, but its goals are, from root to leaf, $\sigma(g_1)$, $\sigma(g_2)$, $\sigma(g_3)$, and $\sigma(G)$.

Reuse employs proof generalization in computing the steps $s(j_i)$. For example, consider j_2. This corresponds to the tactic **and_e*_tac**. The justification tree stored in j_2 consists of a three node tree with rule name *and_e* as the justification at the root and at the only child of the root, followed by j_{prem} at the leaf. This justification will justify any inference using exactly two consecutive applications of the &-elimination rule. The steps for the two rule justifications are $(g_1, (h_1))$ and $(g_2, (h_2))$, where

$$\begin{array}{rcl} g_i & = & H_i, A_i \,\&\, B_i, H'_i \vdash C_i \\ h_i & = & H_i, A_i \,\&\, B_i, A_i, B_i, H'_i \vdash C_i, \end{array}$$

$i = 1, 2$. The generalization procedure attempts to compute a unifier σ for the equation $h_1 = g_2$ (the equation involving the premise is trivial). The result of computing $s(j_2)$ is then $(\sigma(g_1), \sigma(h_2))$.

A complication here is that this unification problem gives rise to unification constraints. λProlog will compute a substitution that is not a unifier, and leave some equations as constraints on further instantiation of metavariables. The main cause of constraints is our representation of hypothesis lists using function composition (as is done in Isabelle) so that higher-order unification can be used to handle metavariables standing for subsequences of hypothesis lists. For example, H_i and H'_i above are of functional type and range over hypothesis lists. These constraints are not a major problem for us because we always immediately use the result of generalization in some more specific context, as in the reuse example above where unification involves the goal ϕ. See Section 4 for the precise form of the step associated with the &-elimination rule.

4 Implementation

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

4.1 λProlog

λProlog is a partial implementation of higher-order hereditary Harrop (hohh) formulas [9] 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 closed formula of the form (pi x_1\ ... (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.

λProlog permits a degree of polymorphism by allowing type declarations to contain type variables (written as capital letters). It is also possible to build new types using type constructors. Two examples used in our implementation are `list` and `pair`, along with the standard constructors `nil` and `::` for lists and `p` for pairs. The map function on lists, for example, is defined as followed.

```
type    map           (A -> B -> o) -> list A -> list B -> o.
map F nil nil.
map F (A::L) (B::M) :- F A B, map F L M.
```

Several non-logical features of λProlog are used in our implementation. We use the cut (`!`) operator to eliminate backtracking points. For example, we implement `var` and `notvar` predicates for variable tests using `!` and `fail`. In addition, we use a primitive `make_abs` described in [4] 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 also freezes a record of any unification constraints on the logic variables occurring in the term. In our `match` procedure the pattern may contain variables with constraints, however the instance may not. An explicit check is included in the implementation, causing `match` to fail when this requirement is not met. To avoid any problems in our implementation, we make the restriction that logic variables in proofs cannot contain constraints.

This is not a severe restriction since, in practice, any constraints that arise get solved before calling the primitive proof construction operations.

4.2 Encoding Logics

As in [3], an object logic is encoded by giving a set of constants with their types to specify the syntax and a set of clauses specifying the inference rules. Similar encodings are also given in [10, 5]. We introduce the types `form`, `seq`, and `prule_name` for formulas, sequents, and primitive rule names, respectively. To represent hypothesis lists of sequents we introduce a type `lform` and an an element constructor `e` of type `form -> (lform -> lform)`. Lists of formulas have type `(lform -> lform)`. The following constants are provided for the specification of object logics.

```
type    |-          (lform -> lform) -> form -> seq.
type    aseq        (A -> seq) -> seq.
type    prule_def   prule_name -> seq -> list seq -> o.
```

The predicate `prule_def` associates a (sequent version of a) step with each rule name. λ-abstracted sequents are also sequents: the constructor `aseq` converts a term `x\(S x)` into a sequent. In a sequent, `aseq x\(S x)`, the bound variable `x` represents a new object level variable whose scope is the sequent `(S x)`.

To specify first-order logic, we introduce the type `tm` for first-order terms, constants `and`, `or`, and `imp` of type `form -> form -> form`, and constants `forall` and `exists` of type `(tm -> form) -> form`. The following `prule_def` clauses specify some of the rules of a sequent calculus for intuitionistic logic.

```
prule_def close (|- (u\ (H1 (e A (H2 u)))) A) nil.
prule_def and_i (|- H (A and B)) ((|- H A)::(|- H B)::nil).
prule_def imp_i (|- H (A imp B)) ((|- (u\ (e A (H u))) B)::nil).
prule_def forall_i (|- H (forall A)) ((aseq (x\ (|- H (A x))))::nil).
prule_def exists_i (|- H (exists A)) ((|- H (A T))::nil).
prule_def and_e (|- (u\ (H1 (e (A and B) (H2 u)))) C)
           ((|- (u\ (H1 (e (A and B) (e A (e B (H2 u)))))) C)::nil).
```

Note the use of λ-abstraction and the `aseq` constructor to represent the eigenvariable condition on the all-introduction rule. In contrast, the exists-introduction rule introduces a new logic variable `T` for the substitution term.

4.3 Proofs and Tactics

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

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

These are intended to form abstract data types and an abstract interface for justifications and proofs. All of our operations for building and modifying proofs do so via the above operations.

The |-, aseq, and prule_def constants defined earlier are also part of the abstract data types. The type goal is the type of goals. Goals are essentially sequents. They also have an additional argument, a list of sequents, which provides more flexibility in using derived rules as is done in Isabelle. We do not use this feature here, so we omit operations which use it. λ-abstracted goals are also goals. The type step and the constructor step implement the steps of Section 2.

The predicate just_step translates a justification to a step. It is defined by case analysis on the three kinds of justifications discussed in Section 2. There is one constructor for each kind of justification. In addition, the type jtree and two constructors are introduced for building justification trees, which form a part of tactic justifications.

```
type  prem     just.
type  prule    prule_name -> just.
type  trule    abs (pair (goal -> proof -> o) goal) -> jtree -> just.
kind  jtree    type.
type  jtree    just -> list jtree -> jtree.
type  ajtree   (A -> jtree) -> jtree.
```

Tactic justifications have three parts. The first is a tactic, which is a predicate of type goal -> proof -> o, and the second is the tactic argument. These two form an "abstracted" pair. Since the logic variables of λProlog are used for metavariables, we cannot simply directly store the tactic argument in the justification because 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. The make_abs operation described earlier is used for this purpose. The third component of a tactic justification is the justification tree. A justification tree is either a pair of a justification and a list of justification trees, or it is a λ-abstracted justification tree where, as in sequents and goals, λ-abstracted variables represent object level variables. In the current implementation, because primitive rules do not take arguments, there can be no metavariables in justification trees.

The implementation of just_step, along with its auxiliary predicates is as follows.

```
just_step prem (step G nil).
just_step (prule N) S :- prule_step N S.
just_step (trule TacAp JT) S :- tactic_step JT S.
prule_step N (step G Gs) :-
   prule_def N S Ss, seq_goal S G, map seq_goal Ss Gs.
type  alist    (A -> list B) -> list B.
tactic_step JT (step G Gs) :- jtree_prems G JT AGs, map_lam AGs Gs.
jtree_prems (agoal G) (ajtree JT) (alist Gs) :-
   pi x\ (jtree_prems (G x) (JT x) (Gs x)).
jtree_prems G (jtree J JTs) (G::nil) :- J = prem, !.
jtree_prems G (jtree J JTs) Prems :-
   just_step J (step G Gs), map3 jtree_prems Gs JTs Gss,
   map map_lam Gss Gss1, flatten Gss1 Prems.
```

The predicate seq_goal is used by prule_step to translate a sequent to a goal. The predicate jtree_prems is used by tactic_step and performs the generalization step described in Section 2 on the input justification tree, but instead of building the proof, retains only the final conclusion and premise goals. Since there may be more than one way to unify the premise of one step with the conclusion of the next, the first solution found by λProlog will be the one chosen, and subsequent unifiers will be enumerated upon backtracking. However, for our example object logic, there will always be a single unifier at each step, though it will have possibly many constraints on it. Subsequent unification or matching with a specific step generally solves these constraints. An alist constructor and two list operations map_lam and flatten are used in these predicates. Abstraction over goals and justification trees is mapped to abstraction over a list of goals. The abstractions are pushed inward to each goal in the list by the map_lam predicate. The flatten predicate is the usual list operation to flatten a list of lists to a single list.

The definition of proofs as well as the one_step_proof predicate and its auxiliary operation goal_prem are defined below.

```
type  proof           goal -> just -> list proof -> proof.
one_step_proof (step G Gs) J (proof G J Ps) :-
   just_step J S, match S (step G Gs), map goal_prem Gs Ps.
goal_prem (bgoal L S) (proof (bgoal L S) prem nil).
goal_prem (agoal G) (aproof P) :- pi x\ (goal_prem (G x) (P x)).
```

Proofs are built using the proof constructor whose arguments are a goal, a justification, and the subproofs at the child nodes. This constructor is hidden. Three ways of building proofs are provided to the user. One is to use aproof to turn an abstracted proof into a proof. The second way to build proofs is to construct a one-step proof from a step and a justification. The query (one_step_proof S J P) computes the step corresponding to the justification J, checks that the step S=(step G Gs) is an instance, 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. For instance, the query:

```
one_step_proof (step (bgoal (|- (u\ u) (forall x\ (q x))))
   ((agoal x\ (bgoal (|- (u\ u) (q x)))))::nil)) (prule forall_i) P
```

returns the following value for P:

```
(proof (bgoal (|- (u\ u) (forall x\ (q x)))) (prule forall_i)
  ((aproof x\ (proof (bgoal (|- (u\ u) (q x))) prem nil))::nil)).
```

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. We do not show its implementation here. It is used in the implementation of then, a combinator for sequencing tactics. Some care was taken with this operation in order to make tactics efficient. In particular, it produces a variant representation of a proof that delays actual computation of the composition. Often 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.

There is one predicate for constructing each of the three kinds of justifications.

```
prem_just prem.
prule_just R (prule R).
tactic_to_just T G P (trule TacAp JT) :-
   var JT, !, make_abs (p T G) TacAp MVars, T G P, proof_jtree P JT.
proof_jtree (aproof P) (ajtree JT) :- !,
   pi x\ (proof_jtree (P x) (JT x)).
proof_jtree (proof G J Ps) (jtree J JTs) :- map proof_jtree Ps JTs.
```

The tactic_to_just operation first uses make_abs to freeze the metavariables in the tactic and goal. Then, the tactic is run on the the goal, returning the tactic's proof from which the corresponding justification tree is extracted using the proof_jtree operation.

This completes the description of the implementation of the operations defined by the abstract interface. These operations are used to implement tactics and tacticals for refinement. The first two below are the operations for both tactic and primitive rule operations. Their implementations are fairly simple. We omit the definition of prems which computes the premises of a proof.

```
type  tactic_refine  (goal -> proof -> o) -> goal -> proof -> o.
type  prule_refine   prule_name -> goal -> proof -> o.
tactic_refine T G P :- tactic_to_just T G TPf J,
  prems TPf Gs, one_step_proof (step G Gs) J P.
prule_refine R G P :- prule_just R J,
  just_step J (step G Gs), one_step_proof (step G Gs) J P.
```

Tacticals are implemented similarly to those in [3]. The main difference is the use of compose_proofs in the implementation of then to provide a way to compose our proof structures via the operations in the abstract interface.

```
maptac T (agoal G) (aproof P) :- !, pi x\ (maptac T (G x) (P x)).
maptac T G P :- T G P.
then T1 T2 G P :- T1 G P1, prems P1 Gs,
  map (maptac T2) Gs Ps, compose_proofs P1 Ps P.
```

Tactics are secure in the sense that proofs are built and checked only via the primitive operations of the abstract interface. A simple example is the implementation of `close_tac`.

```
close_tac G P :- prule_refine close G P.
```

The `reuse` operation illustrated in Section 3 has the following simple implementation, whose core is the use of the `jtree_proof` operation.

```
reuse A P NewP :- proof_jtree P JT,
    jtree_proof (bgoal (u\ u) (|- (u\ u) A)) JT NewP.
jtree_proof (agoal G) (ajtree JT) (aproof P) :- !,
    pi x\ (jtree_proof (G x) (JT x) (P x)).
jtree_proof G (jtree J JTs) P :- prem_just J, !, goal_prem G P.
jtree_proof G (jtree J JTs) P :-
    just_step J (step G Gs), one_step_proof (step G Gs) J TopP,
    map3 jtree_proof Gs JTs Ps, compose_proofs TopP Ps P.
```

The implementation of `jtree_proof` is similar to `jtree_prems` above except that it takes an input goal and builds a proof instead of simply computing a list of premises. The input goal is necessary in order to avoid unification constraints that might otherwise be generated by `just_step` in the third clause above. This operation can be viewed as simultaneously performing generalization (via `just_step`) and specialization of the conclusion of each step to a given input goal.

5 Discussion

Although the programs in this paper make extensive use of many of the higher-order features of λProlog, such features are used in a fairly limited way. For example, quantification over both functions and predicates is at most second-order. Operationally, the unification problems that arise in executing these programs are similar in nature to those encountered in [3] and are all fairly simple. The most significant departure is the `just_step` operation which results in unifiers with possibly many constraints. However, as mentioned, these constraints generally are immediately solved by subsequent unifications.

The reuse operation was illustrated on an example that involved a minor change in the original sequent, allowing us to reuse the entire proof. Several other useful operations can be implemented directly using proof generalization via the `jtree_proof` operation and minor modifications. For example, proof generalization provides a simple operation to backup or undo proofs during refinement. In particular, a user can designate a subproof to be deleted, and `jtree_proof` with the original root goal as input can be used to construct a new proof. Any instantiations done by steps in the deleted subtree will not be present in the new proof. More complicated operations can be defined by combining generalization with replay.

Our implementation generalizes a system we described in [4]. That system supported a variety of undo procedures provided that proofs were constructed by a given collection of operations including refinement by tactics, pruning, and instantiation and reinstantiation of metavariables in proofs. By a simple modification, our `reuse` operation can be added to this collection. To do so, additional data must be stored in tactic justifications and `reuse` must be modified to update this data as it builds a new proof.

Isabelle [10] and Coq [2] have metavariables and support tactic-style theorem-proving, but refinement trees are implicit, and operations on these trees are limited. This also applies to KIV [6], even though it explicitly supports a form of refinement trees. In contrast to ALF [8] 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.

References

1. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
2. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide. Technical Report 154, INRIA, 1993.
3. A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.
4. A. Felty and D. Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In *Twelfth International Conference on Automated Deduction.* Springer-Verlag Lecture Notes in Computer Science, June 1994.
5. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.
6. 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.
7. C. Horn. *The Oyster Proof Development System.* University of Edinburgh, 1988.
8. 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.
9. 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.
10. L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.