Lightweight Lemmas in λProlog: Extended Version[1]

Princeton University CS-TR-607-99, October 1999

Andrew W. Appel
Bell Labs and Princeton University
appel@princeton.edu

Amy P. Felty
Bell Labs, 600 Mountain Avenue, Murray Hill, NJ 07974, USA
felty@research.bell-labs.com

Abstract

λProlog is known to be well-suited for expressing and implementing logics and inference systems. We show that lemmas and definitions in such logics can be implemented with a great economy of expression. We encode a polymorphic higher-order logic using the ML-style polymorphism of λProlog. The terms of the metalanguage (λProlog) can be used to express the statement of a lemma, and metalanguage type-checking can directly type-check the lemma. But to allow polymorphic lemmas requires either more general polymorphism at the meta-level or a less concise encoding of the object logic. We discuss both the Terzo and Teyjus implementations of λProlog as well as related systems such as Elf.

1 Introduction

It has long been the goal of mathematicians to minimize the set of assumptions and axioms in their systems. Implementers of theorem provers use this principle: they use a logic with as few inference rules as possible, and prove lemmas outside the core logic in preference to adding new inference rules. In applications of logic to computer security – such as *proof-carrying code* [18] and distributed authentication frameworks [1] – the implementation of the core logic is inside the trusted code base (TCB), while proofs need not be in the TCB because they can be checked.

Two aspects of the core logic are in the TCB: a set of logical connectives and inference rules, and a program in some underlying programming language that implements proof checking – that is, interpreting the inference rules and matching them against a theorem and its proof.

Definitions and lemmas are essential in constructing proofs of reasonable size and clarity. A proof system should have machinery for checking lemmas,

[1] A shorter version of this paper is to appear in 16th International Conference on Logic Programming, November 1999.

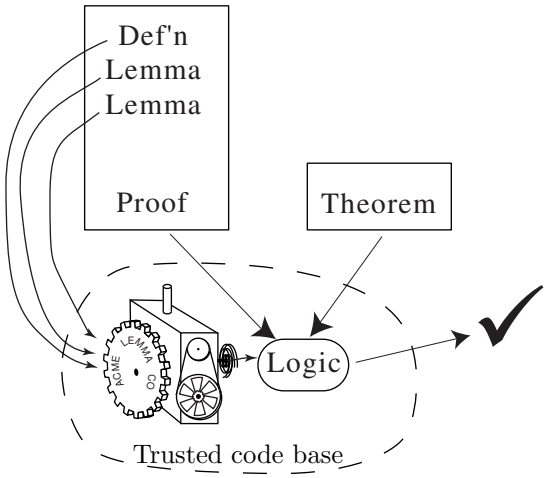

Figure 1: Lemma machinery is inside the TCB.

and applying lemmas and definitions, in the checking of proofs. This machinery also is within the TCB; see Figure 1. Many theorem provers support definitions and lemmas and provide a variety of advanced features designed to help with tasks such as organizing definitions and lemmas into libraries, keeping track of dependencies, and providing modularization; in our work we are particularly concerned with separating that part of the machinery necessary for proof checking (i.e., in the TCB) from the programming-environment support that is used in proof development. In this paper we will demonstrate a definition/lemma implementation that is about two dozen lines of code.

The λProlog language [14] has several features that allow concise and clean implementation of logics, proof checkers, and theorem provers [6]. We use λProlog, but many of our ideas should also be applicable in logical frameworks such as Elf/Twelf [20, 23]. An important purpose of this paper is to show which language features allow a small TCB and efficient representation of proofs. We will discuss *higher-order abstract syntax*, *dynamically constructed clauses*, *dynamically constructed goals*, *meta-level formulas as terms*, *prenex* and *non-prenex polymorphism*, *type abbreviations*, *arithmetic*, and *implementation of λ-terms*.

2 A core logic

The clauses we present use the syntax of the Terzo implementation of λProlog [27]. λProlog is a higher-order logic programming language which extends Prolog in essentially two ways. First, it replaces first-order terms with the more expressive simply-typed λ-terms; λProlog implementations generally extend simple types to include ML-style prenex polymorphism [5, 15], which we use in our implementation. Second, it permits implication and universal quantification (over objects of any type) in goal formulas.

We introduce types and constants using `kind` and `type` declarations, respectively. For example, a new primitive type t and a new constant f of type $t \to t \to t$ are declared as follows:

```
kind   t       type.
type   f       t -> t -> t.
```

Capital letters in type declarations denote type variables and are used in polymorphic types. In program goals and clauses, λ-abstraction is written using backslash `\` as an infix operator. Capitalized tokens not bound by λ-abstraction denote free variables. All other unbound tokens denote constants. Universal quantification is written using the constant `pi` in conjunction with a λ-abstraction (e.g., `pi X\` represents universal quantification over variable `X`). The symbols *comma* and `=>` represent conjunction and implication. The symbol `:-` denotes the converse of `=>` and is used to write the top-level implication in clauses. The type `o` is the type of clauses and goals of λProlog. We usually omit universal quantifiers at the top level in definite clauses, and assume implicit quantification over all free variables.

We will use a running example based on a sequent calculus for a higher-order logic. We call this the *object logic* to distinguish it from the *metalogic* implemented by λProlog. We implement a proof checker for this logic that is similar to the one described by Felty [6]. We introduce two primitive types: `form` for object-level formulas and `pf` for proofs in the object logic. We introduce constants for the object-level connectives, such as `and` and `imp` of type `form` $\to$ `form` $\to$ `form`, and `forall` of type `(A ->  form)` $\to$ `form`. We also have `eq` of type `A` $\to$ `A` $\to$ `form` to represent equality at any type. We use infix notation for the binary connectives. The constant `forall` takes a functional argument, and thus object-level binding of variables by quantifiers is defined in terms of meta-level λ-abstraction. An example of its use is

```
forall (X\ forall (Y\ (eq X Y) imp (eq Y X)))
```

The parser uses the usual rule for the syntactic extent of a lambda, so this expression is equivalent to

```
forall X\ forall Y\ eq X Y imp eq Y X
```

This use of higher-order data structures is called *higher-order abstract syntax* [22]; with it, we don't need to describe the mechanics of substitution explicitly in the object logic [6]. Programs 2 and 3 implement a proof checker for our object logic.

To implement assumptions (that is, formulas to the left of the sequent arrow) we use implication. The goal A `=>` B adds clause A to the λProlog clause database, evaluates B, and then (upon either the success or failure of B) removes A from the clause database. It is a dynamically scoped version of Prolog's `assert` and `retract`. For example, suppose we use `(imp_r initial)` to prove `((eq x y) imp (eq x y))`; then λProlog will execute the (instantiated) body of the `imp_r` clause:

```
kind  form     type.
kind  pf       type.

type  eq       A → A → form.
type  and      form → form → form.          infixl  and     7.
type  imp      form → form → form.          infixr  imp     8.
type  forall   (A → form) → form.

type  proves   pf → form → o.               infix   proves  5.
type  assume   form → o.

type  initial  pf.
type  and_l    form → form → pf → pf.
type  and_r    pf → pf → pf.
type  imp_r    pf → pf.
type  imp_l    form → form → pf → pf → pf.
type  forall_r (A → pf) → pf.
type  forall_l (A → form) → A → pf → pf.
type  cut      pf → pf → form → pf.
type  congr    A → A → (A → form) → pf → pf → pf.
type  refl     pf.
```

Program 2: Type declarations for core logic.

```
(assume (eq x y))  =>  (initial proves (eq x y))
```

This adds `(assume (eq x y))` to the database; then the subgoal

```
initial proves (eq x y)
```

generates a subgoal `(assume (eq x y))` which matches our dynamically added clause.

We have used λProlog's ML-style prenex polymorphism to reduce the number of inference rules in the TCB. Instead of a different `forall` constructor at each type – and a corresponding pair of inference rules – we have a single polymorphic `forall` constructor. Our full core logic (not shown in this paper) uses a base type `exp` of machine integers, and a type `exp → exp` of functions, so if we desire quantification both at expressions and at predicates (let alone functions at several types) we have already saved one constructor and two inference rules.

We have also used polymorphism to define a general congruence rule on the `eq` operator, from which many other desirable facts (transitivity and symmetry of equality, congruence at specific functions) may be proved as lemmas.

Theorem 1 shows the use of our core logic to check a simple proof.

It is important to show that our encoding of higher-order logic in λProlog is *adequate*. To do so, we must show that a formula has a sequent proof if and only if its representation as a term of type `form` has a proof term that can

$\dfrac{}{A, \Gamma \vdash A}$	`initial proves A :- assume A.`
$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$	`(and_r Q1 Q2) proves (A and B) :-` `  Q1 proves A, Q2 proves B.`
$\dfrac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}$	`(imp_r Q) proves (A imp B) :-` `  (assume A) => (Q proves B).`

$\dfrac{A, B, \Gamma \vdash C}{(A \wedge B), \Gamma \vdash C}$	`(and_l A B Q) proves C :-`
	`  assume (A and B),`
	`  (assume A) => (assume B) => (Q proves C).`

$\dfrac{\Gamma \vdash A \qquad B, \Gamma \vdash C}{(A \to B), \Gamma \vdash C}$	`(imp_l A B Q1 Q2) proves C :-`
	`  assume (A imp B), Q1 proves A,`
	`  (assume B) => (Q2 proves C).`

$\dfrac{\Gamma \vdash A(y) \quad \text{for any } y \text{ not in conclusion}}{\Gamma \vdash \forall x\ A(x)}$	`(forall_r Q) proves (forall A) :-` `  pi y\ ((Q y) proves (A y)).`

$\dfrac{A(T), \Gamma \vdash C}{\forall x\ A(x), \Gamma \vdash C}$	`(forall_l A T Q) proves C :-`
	`  assume (forall A),`
	`  (assume (A T)) => (Q proves C).`

$\dfrac{\Gamma \vdash A \qquad A, \Gamma \vdash C}{\Gamma \vdash C}$	`(cut Q1 Q2 A) proves C :-`
	`  Q1 proves A,`
	`  (assume A) => (Q2 proves C).`

$\dfrac{\Gamma \vdash X = Z \qquad \Gamma \vdash H(Z)}{\Gamma \vdash H(X)}$	`(congr X Z H Q P) proves (H X) :-` `  Q proves (eq X Z), P proves (H Z).`
$\dfrac{}{\Gamma \vdash X = X}$	`refl proves (eq X X).`

Program 3: Inference rules of core logic.

```
 (forall_r I\ forall_r J\ forall_r K\
  (imp_r (and_l (eq J I) (eq J K)
         (congr I J (X\ (eq X K))
          (congr J I (eq I) initial refl) initial))))
proves
 (forall I\ forall J\ forall K\ (eq J I and eq J K) imp eq I K).
```

Theorem 1. $\forall I\ \forall J\ \forall K\ (J = I \wedge J = K) \to I = K$.

```
type  lemma  (A → o) → A → (A → pf) → pf.

(lemma Inference Proof Rest) proves C :-
 pi Name\ (valid_clause (Inference Name),
           Inference Proof,
           (Inference Name) => ((Rest Name) proves C)).
```

Program 4: The `lemma` proof constructor.

be checked using the inference rules of Program 3. Proving such a theorem should be straightforward. In particular, since we have encoded our logic using prenex polymorphism, we can expand out instantiated copies of all of the polymorphic expressions in terms of type `pf`; the expanded proof terms will then map directly to sequent proof trees.

3 Lemmas

In mathematics the use of lemmas can make a proof more readable by structuring the proof, especially when the lemma corresponds to some intuitive property. For automated proof checking (in contrast to automated or traditional theorem proving) this use of lemmas is not essential, because the computer doesn't need to understand the proof in order to check it. But lemmas can also reduce the *size* of a proof (and therefore the time required for proof checking): when a lemma is used multiple times it acts as a kind of "subroutine." This is particularly important in applications like proof-carrying code where proofs are transmitted over networks to clients who check them.

The heart of our lemma mechanism is the clause shown in Program 4. The proof constructor `lemma` takes three arguments: (1) a derived inference rule `Inference` (of type `A → o`) parameterized by a proof constructor (of type `A`), (2) a term of type `A` representing a proof of the lemma built from core-logic proof constructors (or using other lemmas), and (3) a proof of the main theorem `C` that is parameterized by a proof constructor (of type `A`).

For example, we can prove a lemma about the symmetry of equality; the proof uses congruence and reflexivity of equality:

```
pi A\ pi B\ pi P\ (P proves (eq B A) =>
                   ((congr B A (eq A) P refl) proves (eq A B))).
```

This theorem can be checked as a successful λProlog query to Programs 2 and 3: for an arbitrary `P`, add `(P proves (eq B A))` to the logic, then check the proof of congruence using this fact. The syntax F `=>` G means exactly the same as G `:-` F, so we could just as well write this query as:

```
pi A\ pi B\ pi P\ ((congr B A (eq A) P refl) proves (eq A B) :-
                   P proves (eq B A)).
```

```
(lemma
  (Symmx\ pi A\ pi B\ pi P\
     (Symmx A B P) proves (eq A B) :- P proves (eq B A))
  (A\B\P\(congr B A (eq A) P refl))
  (symmx\ (forall_r I\ forall_r J\ imp_r (symmx J I initial))))
proves (forall I\ forall J\ eq I J imp eq J I).
```

Theorem 2. $\forall I\, \forall J\, (I = J \rightarrow J = I)$.

Now, suppose we abstract the proof (roughly, `congr B A (eq A) P refl`) from this query:

```
(Inference = (PCon\ pi A\ pi B\ pi P\
              (PCon A B P) proves (eq A B) :- P proves (eq B A)),
 Proof = (A\B\P\ congr B A (eq A) P refl),
 Query = (Inference Proof),
 Query)
```

The solution of this query proceeds in four steps: the variable `Inference` is unified with a λ-term; `Proof` is unified with a λ-term; `Query` is unified with the application of `Inference` to `Proof` (which is a term β-equivalent to the query of the previous paragraph), and finally `Query` is solved as a goal (checking the proof of the lemma).

Once we know that the lemma is valid, we make a new λProlog atom `symmx` to stand for its proof, and we prove some other theorem in a context where the clause `(Inference symmx)` is in the clause database; remember that `(Inference symmx)` is β-equivalent to

```
pi A\ pi B\ pi P\ ((symmx A B P) proves (eq A B) :-
                   P proves (eq B A)).
```

This looks remarkably like an inference rule! With this clause in the database, we can use the new proof constructor `symmx` just as if it were primitive.

To "make a new atom" we simply `pi`-bind it. This leads to the recipe for lemmas shown in Program 4 above: first execute `(Inference Proof)` as a query, to check the proof of the lemma itself; then `pi`-bind `Name`, and run `Rest` (which is parameterized on the lemma proof constructor) applied to `Name`. Theorem 2 illustrates the use of the `symmx` lemma. The `symmx` proof constructor is a bit unwieldy, since it requires `A` and `B` as arguments. We can imagine writing a primitive inference rule

```
(symm P) proves (eq A B) :- P proves (eq B A).
```

using the principle that the proof checker doesn't need to be told `A` and `B`, since they can be found in the formula to be proved.

Therefore we add three new proof constructors – `elam`, `extract`, and `extractGoal` – as shown in Program 5. These can be used in the following stereotyped way to extract components of the formula to be proved. First

```
type elam          (A → pf) → pf.
type extract       form → pf → pf.
type extractGoal   o → pf → pf.

(elam Q) proves B :- (Q A) proves B.
(extract B P) proves B :-  P proves B.
(extractGoal G P) proves B :- valid_clause G, G, P proves B.
```

Program 5: Proof constructors for implicit arguments of lemmas.

```
(lemma
  (Symm\ pi A\ pi B\ pi P\
    (Symm P) proves (eq A B) :- P proves (eq B A))
  (P\ elam A\ elam B\ extract (eq A B) (congr B A (eq A) P refl))
  (symm\ (forall_r I\ forall_r J\ imp_r (symm initial))))
proves (forall I\ forall J\ eq I J imp eq J I).
```

Theorem 3. $\forall I\, \forall J\, (I = J \rightarrow J = I)$.

bind variables with `elam`, then match the target formula with `extract`. Theorem 3 is a modification of Theorem 2 that makes use of these constructors. The `extractGoal` asks the checker to run λProlog code to help construct the proof. Of course, if we want proof checking to be finite we must restrict what kinds of λProlog code can be run, and this is accomplished by `valid_clause` (see below). The proof of lemma `def_l` in Section 4 is an example of `extractGoal`.

Of course, we can use one lemma in the proof of another, as shown by Theorem 4.

Since the type of `(Inference Proof)` is `o`, the lemma `Inference` might conceivably contain any λProlog clause at all, including those that do input/output. Such λProlog code cannot lead to unsoundness – if the resulting proof checks, it is still valid. But there are some contexts where we wish to restrict the kind of program that can be run inside a proof. For example, in a proof-carrying-code system, the code consumer might not want the proof to execute λProlog code that accesses private local resources.

To limit the kind and amount of execution possible in the executable part of a lemma, we introduce the `valid_clause` predicate of type `o → o` (Program 6). A clause is valid if it contains `pi`, comma, `:-`, `=>`, `proves`, `assume`, and nothing else. Of course, a `proves` clause contains subexpressions of type `pf` and `form`, and an `assume` clause has a subexpression of type `form`, so all the constants in proofs and formulas are also permitted. Absent from this list are λProlog input/output (such as `print`) and the semicolon (backtracking search).

In principle, we do not need lemmas at all. Instead of the symmetry lemma, we can prove `(forall A\ forall B\ (eq B A imp eq A B))` and then cut it into the proof of a theorem using the ordinary `cut` of sequent

```
(lemma
  (Symm\ pi X\ pi Y\ pi P\
    (proves (Symm P) (eq X Y) :- proves P (eq Y X)))
  (P\ elam X\ elam Y\
    (extract (eq X Y) (congr Y X (eq X) P refl)))
  symm\

(lemma
  (Trans\ pi X\ pi Y\ pi Z\ pi P1\ pi P2\
    (proves (Trans Z P1 P2) (eq X Y) :-
     proves P1 (eq X Z), proves P2 (eq Z Y)))
  (Z\ P1\ P2\ elam X\ elam Y\
    (extract (eq X Y) (congr Y Z (eq X) (symm P2) P1)))
  trans\

(lemma
  (And_imp\ pi A\ pi B\ pi C\ pi Q\
    (proves (And_imp Q) ((A and B) imp C) :-
     proves Q (A imp B imp C)))
  (Q\ elam A\ elam B\ elam C\ (extract ((A and B) imp C)
   (imp_r (and_l A B
    (cut Q (imp_l A (B imp C) initial
            (imp_l B C initial initial)) (A imp B imp C))))))
  and_imp\

  (forall_r I\ forall_r J\ forall_r K\
   (and_imp (imp_r (imp_r (trans J (symm initial)
                                initial)))))))))))

proves
(forall I\ forall J\ forall K\ (eq J I and eq J K) imp eq I K)
```

Theorem 4. $\forall I\, \forall J\, \forall K\, (J = I \wedge J = K) \rightarrow I = K$.

```
valid_clause (pi C) :-   pi X\ valid_clause (C X).
valid_clause (A,B) :-    valid_clause A, valid_clause B.
valid_clause (A :- B) :- valid_clause A, valid_clause B.
valid_clause (A => B) :- valid_clause A, valid_clause B.
valid_clause (P proves A).
valid_clause (assume A).
```

Program 6: Valid clauses.

calculus. To make use of this fact requires two `forall_l`'s and an `imp_l`. This approach adds undesirable complexity to proofs.

It should be possible to directly extend soundness and adequacy to the system with lemmas by showing that it is possible to replace any lemma with a cut-in formula in the way we have discussed for `symm`.

4 Definitions

Definitions are another important mechanism for structuring proofs to increase clarity and reduce size. If some property (of a base-type object, or of a higher-order object such as a predicate) can be expressed as a logical formula, then we can make an abbreviation to stand for that formula.

For example, we can express the fact that f is an associative function by the formula $\forall X\ \forall Y\ \forall Z\ f\,X\,(f\,Y\,Z) = f\,(f\,X\,Y)\,Z$. Putting this formula in λProlog notation and abstracting over f, we get the predicate:

```
F\ forall X\ forall Y\ forall Z\ eq (F X (F Y Z)) (F (F X Y) Z)
```

A definition is just an association of some name with this predicate:

```
eq assoc
 (F\ forall X\ forall Y\ forall Z\ eq (F X (F Y Z)) (F (F X Y) Z))
```

To use definitions in proofs we introduce three new proof rules: (1) `define` to bind a λ-term to a name, (2) `def_r` to replace a formula on the right of a sequent arrow with the definition that stands for it (or viewed in terms of backward sequent proof, to replace a defined name with the term it stands for), and (3) `def_l` to expand a definition on the left of a sequent arrow during backward proof. All three of these proof constructors are just lemmas provable in our system using congruence of equality, as Program 7 shows. Theorem 5 shows a proof using definitions.

To check a proof `(define Formula (Name\ (RestProof Name)))` the system interprets the `pi D` within the `define` lemma to create a new atom `D` to stand for the `Name`. It then adds `(assume(eq D Formula))` to the clause database. Finally it substitutes `D` for `Name` within `RestProof` and checks the resulting proof. If there are occurrences of `(def_r D)` or `(def_l D)` within `(RestProof D)` then they will match the newly added clause.

To check that `(def_r assoc (A\ A f) P)` is a proof of the formula `(assoc f)` the prover checks that `(A\ A f)(assoc)` matches `(assoc f)` and that `(assume (eq assoc Body))` is in the assumptions for some formula, predicate, or function `Body`. Then it applies `(A\ A f)` to `Body`, obtaining the subgoal `(Body f)`, of which `P` is required to be a proof.

To check that `(def_l assoc (A\ A f) P)` proves some formula `D`, the checker first reduces `(A\ A f)(assoc)` to `(assoc f)`, and then checks that `(assume (assoc f))` is among the assumptions in the λProlog database.

```
(lemma
  (Define\ pi F\ pi P\ pi B\
    ((Define F P) proves B :-
     pi D\ (assume (eq d F) => (P D) proves B)))
  (F\P\ (cut refl (P F) (eq F F)))
  define\

(lemma
  (Def_r\ pi Name\ pi B\ pi F\ pi P\
    ((Def_r Name B P) proves (B Name) :-
     assume (eq Name F), P proves (B F)))
  (Name\B\P\ elam F\ (extract (B Name)
   (extractGoal (assume (eq Name F)) (congr Name F B initial P))))
  def_r\

(lemma
  (Def_l\ pi Name\ pi B\ pi D\ pi F\ pi Q\
    ((Def_l Name B Q) proves D :- assume (B Name),
     assume (eq Name F), (assume (B F) => Q proves D)))
  (Name\B\Q\ elam F\ (extractGoal (assume (eq Name F))
   (cut (congr F Name B (symm initial) initial) Q (B F))))
  def_l\ ...
```

Program 7: Machinery for definitions.

Then it verifies that `(assume (eq assoc Body))` is in the assumption database for some `Body`. Finally the checker introduces `(assume (Body f))` into the assumptions and verifies that, under that assumption, `Q` proves `D`.

If there is more than one formula of the form `(eq assoc _)` among the assumptions, then either the checker will backtrack or – if cuts are introduced into the checker to prohibit backtracking – the checker will fail. It is up to the author of the proof to avoid this situation. However, such backtracking or failure cannot lead to unsoundness, that is, to invalid proofs being accepted.

5 Dynamically constructed clauses and goals

Our technique allows lemmas and definitions to be contained *within* the proof. We do not need to install new "global" lemmas and definitions into the proof checker. The dynamic scoping also means that the lemmas of one proof cannot interfere with the lemmas of another, even if they have the same names. This machinery uses several interesting features of λProlog:

Metalevel formulas as terms. As we have seen, the `symm` lemma

```
(Symm\ pi A\ pi B\ pi P\
  (Symm P) proves eq A B :- P proves eq B A)
```

```
(lemma ... symm\
(lemma ... trans\
(lemma ... define\
(lemma ... def_l\
(lemma ... def_r\

(define
  (f\ (forall a\ forall b\ forall c\
       (eq (f a (f b c)) (f (f a b) c))))
  assoc\

(lemma
  (Assoc_inst\ pi F\ pi A\ pi B\ pi C\
    ((Assoc_inst F) proves
     (eq (F A (F B C)) (F (F A B) C)) :- assume (assoc F)))
  (F\ elam A\ elam B\ elam C\
   (extract (eq (F A (F B C)) (F (F A B) C))
    (def_l assoc (Assoc\ (Assoc F))
     (forall_l (a\ forall b\ forall c\
                 (eq (F a (F b c)) (F (F a b) c)) A)
      (forall_l (b\ forall c\
                  (eq (F A (F b c)) (F (F A b) c)) B)
       (forall_l (c\ (eq (F A (F B c)) (F (F A B) c)) C)
        initial))))))
  assoc_inst\

  (forall_r f\ (imp_r (forall_r a\
   (cut (def_r assoc (Assoc\ (Assoc f)) initial)
        (trans (f (f a a) (f a a)) (assoc_inst f) (assoc_inst f))
        (assoc f)))))))))))
proves
  (forall f\
   ((forall a\ forall b\ forall c\
     (eq (f a (f b c)) (f (f a b) c))) imp
      (forall a\ eq (f a (f a (f a a))) (f (f (f a a) a) a)))).
```

Theorem 5. $(\forall a, b, c\ fa(fbc) = f(fab)c) \rightarrow \forall a\ fa(fa(faa)) = f(f(faa)a)a$.

occurs inside the proofs as an argument to the `lemma` constructor and so is just a data structure (parameterized by `Symm`); it does not "execute" anything, in spite of the fact that it contains the λProlog connectives `:-` and `pi`. This gives us the freedom to write lemmas using the same syntax as we use for writing primitive inference rules.

Dynamically constructed goals. When the clause from Program 4 for the `lemma` proof constructor checks the proof of a lemma by executing the goal `(Inference Proof)`, we are executing a goal that is built from a run-time-constructed data structure. `Inference` will be instantiated with terms such as the one above representing the `symm` lemma. It is only when such a term is applied to its proof and thus appears in "goal position" that it becomes the current subgoal on the execution stack.

Dynamically constructed clauses. When, having successfully checked the proof of a lemma, the `lemma` clause executes

```
(Inference Name) => ((Rest Name) proves C))
```

it is adding a dynamically constructed clause to the λProlog database.

The Teyjus system does not allow `=>` or `:-` to appear in arguments of predicates. It also does not allow variables to appear at the head of the left of an implication. These restrictions come from the theory underlying λProlog [12]; without this restriction, a runtime check is needed to insure that every dynamically created goal is an acceptable one. We now show that it is possible to relax the requirements on dynamically constructed clauses and goals to accommodate Teyjus's restrictions.

We can avoid putting `:-` inside arguments of predicates by writing the lemma as

```
(Symm\ pi A\ pi B\ pi P\
  (Symm P) proves (eq A B) <<== P proves (eq B A))
```

where `<<==` is a new infix operator of type $\mathtt{o} \to \mathtt{o}$. But this, in turn, means that the clause for checking lemmas cannot add `(Inference Name)` as a new clause, since `<<==` has no operational meaning. Instead, Program 8 contains a modified `lemma` clause that adds the clause `(cl (Inference Name))` where `cl` is a new atomic predicate of type $\mathtt{o} \to \mathtt{o}$. The rest of Program 8 implements an interpreter to handle clauses of the form `(cl A)` and goals of the form `(A <== B)` and `(A ==>> B)`. The use of `cl` is the only modification to the `lemma` clause. The new clause for the `proves` predicate is used for checking nodes in a proof representing lemma applications and illustrates the use of the new atomic clauses. The `(cl Cl)` subgoal looks up the lemmas that have been added one at a time and tries them out via the `backchain` predicate. This predicate processes the clauses in a manner similar to the

```
type    ==>>         o -> o -> o.
infixr  ==>>         4.
type    <<==         o -> o -> o.
infixl  <<==         0.
type    cl           o -> o.
type    backchain    o -> o -> o.

(lemma Inference Proof Rest) proves C :-
 pi Name\ (valid_clause (Inference Name),
           Inference Proof,
           cl (Inference Name) => ((Rest Name) proves C)).

P proves A :- cl Cl, backchain (P proves A) Cl.

backchain G G.
backchain G (pi D) :- backchain G (D X).
backchain G (A,B) :-  backchain G A; backchain G B.
backchain G (H <<== G1) :-  backchain G H, G1.
backchain G (G1 ==>> H) :-  backchain G H, G1.

(D ==>> G) :- (cl D) => G.
(G <<== D) :- (cl D) => G.
```

Program 8: An interpreter for dynamic clauses.

λProlog language itself. The remaining two clauses are needed in both checking lemmas and in checking the rest of the proof for interpreting the new implication operators when they occur at the top level of a goal.

Handling new constants for `:-` and `=>` is easy enough operationally. However, it is an inconvenience for the user, who must use different syntax in lemmas than in inference rules.

If the metalanguage prohibits all terms having `o` in their types as arguments to a predicate, we can go further and introduce a new constant for each metalevel connective, and extend the interpreter to handle them all. Such an implementation makes it possible to implement lemmas nearly as directly as in Section 3 even if the metalanguage does not allow metalevel formulas in terms at all. Note that when metalevel formulas are not allowed, there is no possibility for dynamically created goals or clauses.

To write a full interpreter, we introduce a new type `goal` and connectives which build terms of this type. In particular, we now give `<<==` and `==>>` the type `goal` → `goal` → `goal`. We also introduce a new constant `^^` for conjunction having the same type as the implication constructors. Finally, we introduce `all` for universal quantification having type `(A` → `goal)` → `goal`. In addition, we change the type of `backchain` to `goal` → `goal` → `o`, and modify the clauses for comma and `pi` to use the new constants. In the clauses for `<<==` and `==>>` in Program 8, note that the goal `G1` which appears as an argument inside the head of the clause also appears as a goal in the body of

```
kind   goal          type.

type    ==>>         goal -> goal -> goal.
infixr  ==>>         4.
type    <<==         goal -> goal -> goal.
infixl  <<==         0.
type    ^^           goal -> goal -> goal.
infixl  ^^           3.
type    all          (A -> goal) -> goal.

type    cl           goal -> o.
type    backchain    goal -> goal -> o.
type    solveg       goal -> o.

type    proves       pf -> form -> goal.
type    assume       form -> goal.
type    valid_clause goal -> goal.

solveg (all G) :- pi x\ (solveg (G x)).
solveg (G1 ^^ G2) :- solveg G1, solveg G2.
solveg (D ==>> G) :- (cl D) => solveg G.
solveg (G <<== D) :- (cl D) => solveg G.
solveg G :- cl D, backchain G D.

backchain G G.
backchain G (all D) :- backchain G (D X).
backchain G (A ^^ B) :- backchain G A; backchain G B.
backchain G (H <<== G1) :- backchain G H, solveg G1.
backchain G (G1 ==>> H) :- backchain G H, solveg G1.
```

Program 9: A full interpreter.

the clause. In the full interpreter, we cannot do this. `G1` no longer has type `o`; it has type `goal` and is constructed using the new connectives. Instead, we replace `G1` with `(solveg G1)` and implement the `solveg` predicate to handle the solving of goals. The new code for `solveg` and the modified code for `backchain` is in Program 9. In order to use this interpreter to solve goals of the form `(P proves A)`, the `proves` predicate must be a constructor for terms of type `goal`, and the meta-level goal presented to λProlog must have the form `(solveg (P proves A))`. Similarly, inference rules must also be represented as objects of type `goal` and wrapped inside `cl` to form λProlog clauses. Several examples of clauses for inference rules are given in Program 10 to illustrate. The last clause is the new clause for handling lemmas. Note that in this version, `valid_clause` constructs objects of type `goal`; thus all the clauses for `valid_clause` must also be wrapped in `cl`.

```
cl (proves initial A <<== assume A).
cl (proves (and_r Q1 Q2) (A and B) <<==
    proves Q1 A ^^ proves Q2 B).
cl (proves (imp_r Q) (A imp B) <<==
    (assume A) ==>> (proves Q B)).
cl (proves (forall_r Q) (forall A) <<==
    all y\ (proves (Q y) (A y))).
cl (proves (lemma Inference LemmaProof RestProof) C <<==
  all Name\
   (valid_clause (Inference Name) ^^
    Inference LemmaProof ^^
    (Inference Name) ==>> (proves (RestProof Name) C))).
```

Program 10: Clauses used by the full interpreter.

```
(lemma (Symm\ pi A\ pi B\ pi P\
         (Symm P) proves (eq A B) :- P proves (eq B A))
       (P\ elam A\ elam B\
         (extract (eq A B) (congr B A (eq A) P refl)))
       (symm\ (forall_r f\ forall_r g\ forall_r x\
         (imp_r (imp_r (and_r (symm initial) (symm initial)))))))
proves
(forall f\ forall g\ forall x\
  (eq f g) imp (eq (f x) x) imp ((eq g f) and (eq x (f x)))).
```

Theorem 6. $\forall f, g, x. f = g \rightarrow f(x) = x \rightarrow (g = f \wedge x = f(x))$.

6 Meta-level types

In the encoding we have presented, ML-style prenex polymorphism is used in the `forall_r` and `congr` rules of Program 3 and in implementing lemmas as shown in Program 4. We now discuss the limitations of prenex polymorphism for implementing lemmas which are themselves polymorphic; and we discuss ways to overcome these limitations both at the meta-level and at the object level. The `symm` lemma is naturally polymorphic: it should express the idea that $a = 3 \rightarrow 3 = a$ (at type int) just as well as $f = \lambda x.3 \rightarrow (\lambda x.3) = f$ (at type int $\rightarrow$ int). But Theorem 6, which uses `symm` at two different types, fails to type-check in our implementation. When the λProlog type-checker first encounters `symm` as a λ-bound variable, it creates an uninstantiated type metavariable to hold its type. The first use of `symm` unifies this metavariable type variable with the type `T` of `x`, and then the use of `symm` at type `T` $\rightarrow$ `T` fails to match. Prohibiting λ-bound variables from being polymorphic is the essence of prenex polymorphism. On the other hand, the proof of Theorem 3 type-checks because `symm` is used at only one type. We can fix Theorem 6 by including two copies of the `symm` lemma inside the proof and using each one only at one type. This problem was in fact already hidden inside the proof of Theorem 5. In this proof, the `symm` lemma is used in the proofs

of both `trans` and `def_l`, `def_l` is used in the proof of `assoc_inst`, and both `trans` and `assoc_inst` are used in the proof of the theorem. Tracing through the types, we see that `symm` is used as some type `A` via `trans`, and also at a different type `(A → A → A)` via `assoc_inst`. The (hidden) reason that this proof checks is that it contains a proof of `def_l` that is different from the one given in Program 7 and in fact doesn't use `symm` at all.

We can generalize the prenex polymorphism of the metalanguage by removing the restriction that all type variables are bound at the outermost level and allow such binding to occur anywhere in a type, to obtain the second-order λ-calculus. We start by making the bindings clear in our current version by annotating terms with fully explicit bindings and quantification. The result will not be λProlog code, as type quantification and type binding are not supported in that language. So we will use the standard λProlog `pi` and `\` to quantify and abstract term variables; but we'll use Π and Λ to quantify and abstract type variables, and use *italics* for type arguments and other nonstandard constructs.

```
type congr     ΠT. T → T → (T → form) → pf → pf → pf.
type forall_r  ΠT. (T → pf) → pf.

ΠT. pi X: T\ pi Z: T\ pi H: T → form\ pi Q: pf\ pi P: pf\
  (congr T X Z H Q P) proves (H X) :-
    Q proves (eq T X Z), P proves (H Z).

ΠT. pi A: T → form\ pi Q: T → pf\
  (forall_r T Q) proves (forall T A) :- pi Y:T\ (Q Y proves A Y).
```

Every type quantifier is at the outermost level of its clause; the ML-style prenex polymorphism of λProlog can typecheck this program. However, we run into trouble when we try to write a polymorphic lemma. The lemma itself is prenex polymorphic, but the lemma definer is not.

Figure 11 is pseudo-λProlog in which type quantifiers and type bindings are shown explicitly. The line marked *here* contains a λ-term, λ`Symm`.body, in which the type of `Symm` is $\Pi T.$`pf` → `pf`. Requiring a function argument to be polymorphic is an example of non-prenex polymorphism, which is permitted in second-order λ-calculus but not in an ML-style type system.

Polymorphic definitions (using `define`) run into the same problems and also require non-prenex polymorphism. For example, the given proof of Theorem 7 cannot be checked in a prenex framework. Thus prenex polymorphism is sufficient for polymorphic inference rules; non-prenex polymorphism is necessary to directly extend the encoding of our logic to allow polymorphic lemmas, although one can scrape by with monomorphic lemmas by always duplicating each lemma at several different types within the same proof.

Proofs in a nonprenex polymorphic calculus cannot, in general, be expanded out to monomorphic proofs. Therefore we cannot prove adequacy with respect to Church's higher-order logic. Instead, we can view the (nonprenex polymorphic) object logic as a sublogic of the calculus of construc-

```
type  lemma ΠT. (T → o) → T → (T → pf) → pf.

(lemma T Inference Proof Rest) proves C :-
 pi Name:T\ (valid_clause (Inference Name),
             Inference Proof,
             (Inference Name) => ((Rest Name) proves C)).
(lemma  T
        (Symm: ΠT. pf → pf \        ← here!
          ΠT. pi A:T\ pi B:T\ pi P:pf\
            (Symm T P) proves (eq T A B) :- P proves (eq T B A))
        (ΛT. P:pf\ elam A:T\ elam B:T\
          (extract (eq T A B) (congr T B A (eq T A) P refl)))
        (symm\ (forall_r I:int\ forall_r J:int\
          (imp_r (symm int initial)))))
proves (forall I\ forall J\ (eq int I J) imp (eq int J I)).
```

Figure 11: Explicitly typed version of Theorem 3.

tions [4], and prove the soundness and adequacy of our system with respect to that logic (although we have not done such a proof).

The prenex-polymorphic λProlog language can represent only a restricted set of λ-terms, sufficient for polymorphic inference rules but not polymorphic lemmas and definitions. Perhaps the problem lies in using a statically typed metalanguage. Lamport and Paulson [10] have argued that types are not necessary to a logical metalanguage; the errors that would be caught by a static type system will always be caught eventually because invalid theorems simply won't prove, and sometimes the types just get in the way.

Types, however, play an essential role in λProlog [15]. For example, they are necessary for unification. There is a sublanguage called L_λ [11] in which unification doesn't require types, but it is too weak (with restrictions on β-equivalence) to encode our logic without extra rules to perform substitution explicitly.

Regardless of whether the metalanguage is typed, our logic must certainly be typed. Just as untyped set theory is unsound (with paradoxes about sets that contain themselves), the untyped version of our higher-order logic is also unsound. The proof is simple: in untyped λProlog we could represent the fixed-point function Y `= (F\ (X\ F(X X))(X\ F(X X)))`, with the theorem $\forall f.\, Yf = f(Yf)$. By applying Y to `(X\ X imp false)` we can prove $\exists x.\, x = (x \to \text{false})$ from which anything can be proved.

Therefore, if we build our system in an untyped logical framework then our checker would have to include an implementation of static polymorphic typechecking of object-logic terms. The machinery for typechecking the object logic – written out as λProlog inference rules – would be about as large as the proof-checking machinery shown in Figure 3; it is this machinery that we avoid by using a statically typed metalanguage.

There are also several ways to encode our polymorphic logic and allow for

```
(lemma ... symm\
(lemma ... define\
(lemma ... def_l\
(lemma ... def_r\
(define ... assoc\

(forall_r f (forall_r g (imp_r (imp_r (imp_r
   (cut (def_r assoc (Assoc Assoc f) initial)
     (cut (def_r assoc (Assoc Assoc g) initial)
      (and_r
       (def_l assoc (Assoc Assoc f) initial)
       (def_l assoc (Assoc Aassoc g) initial))
      (assoc g))
     (assoc f))))))))))))

proves
```

$$(\forall f, g\ (gff = f)) \rightarrow$$
$$(\forall a, b, c\ (fa(fbc) = f(fab)c)) \rightarrow (\forall a, b, c\ (ga(gbc) = g(gab)c)) \rightarrow$$
$$(\forall a, b, c\ (fa(fbc) = f(fab)c) \wedge \forall a, b, c\ (ga(gbc) = g(gab)c))$$

Theorem 7.

polymorphic lemmas without changing the metalanguage. One possibility is to encode object-level types as meta-level terms. The following encoding of the `congr` rule illustrates this approach.

```
kind tp     type.
kind tm     type.
type arrow  tp → tp → tp.
type form   tp.
type eq     tp → tm → tm → tm.
type congr  tp → pf → pf → (tm → tm) → tm → tm → pf.

congr T Q P H X Z proves H X :-
  typecheck X T, typecheck Z T, Q proves (eq T X Z), P proves H Z.
```

This encoding also requires the addition of explicit `app` and `abs` constructors, primitive rules for β- and η-reduction, and typechecking clauses for terms of types `exp` and `form`, but not `pf`. To illustrate, the new constructors and corresponding type checking clauses are given below.

```
type app  tp → tm → tm → tm.
type lam  (tm → tm) → tm.
typecheck (app T1 F X) T2 :-
  typecheck F (arrow T1 T2),  typecheck X T1.
typecheck (lam F) (arrow T1 T2) :-
  pi X\ (typecheck X T1 => typecheck (F X) T2).
```

This encoding loses some economy of expression because of the extra constructors needed for the encoding, and requires a limited amount of typechecking, though not as much as would be required in an untyped framework.

For instance, in addition to typechecking subgoals such as the ones in the `congr` rule, it must also be verified that all the terms in a particular sequent to be proved have type `form`. In this encoding, polymorphism at the meta-level is no longer used to encode formulas, although it is still used for the `lemma` constructor. Lemma polymorphism can also be removed by using an application constructor at the level of proofs, though this would require adding typechecking for proofs also.

Another alternative is to use an encoding similar to one by Harper et al. [7] (for a non-polymorphic higher-order logic) in a metalanguage such as Elf/Twelf [20, 23]. The extra expressiveness of dependent types allows object-level types to be expressed more directly as meta-level types, eliminating the need for any typechecking clauses. This encoding still requires explicit constructors for `app` and `abs` as well as primitive rules for $\beta\eta$-reduction. The following Twelf clauses, corresponding to λProlog clauses above, illustrate the use of dependent types for this kind of encoding.

```
tp : type.
tm : tp → type.
form : tp.
pf : tm form → type.
arrow : tp → tp → tp.
eq : {T:tp}tm T → tm T → tm form.
congr : {T:tp}{X:tm T}{Z:tm T}{H:tm T → tm form}
          pf (eq T X Z) → pf (H Z) → pf (H X).
```

Elf [20] and Twelf [23] are both implementations of LF [7], the Edinburgh logical framework. Elf 1.5 has full (nonprenex) statically checked polymorphism with explicit type quantification and explicit type binding, which we have used to implement polymorphic lemmas approximately as shown in Figure 11. But polymorphism in Elf 1.5 is undocumented and discouraged [21], so we recommend the above encoding instead. Twelf is the successor to Elf. Like Elf, it has higher-order data structures with a static type system, but Twelf is monomorphic. Thus, the above encoding is the only possibility.

Both of the above λProlog and Twelf encodings look promising as a basis for a proof system with polymorphic lemmas [2].

7 Type abbreviations

In the domain of proof-carrying code, we encode types as predicates which themselves take predicates as arguments. For example, our program has declarations like this one:

```
type hastype (exp → form) → (exp → exp) → exp →
              ((exp → form) → (exp → exp) → exp → form) → form.
```

Neither Terzo nor Teyjus allow such abbreviations and this is rather an inconvenience. ML-style (nongenerative) type abbreviations would be very

```
kind exp  type.
type const        int → exp.
type plus,minus   exp → exp → exp.
type greater      exp → exp → form.

eval_plus proves (eq (plus (const I) (const J)) (const K)) :-
  K is (I + J).

gt_c proves (greater (const I) (const J)) :- I > J.

plus_zero proves (eq (plus A (const 0)) A).
plus_comm proves (eq (plus A B) (plus B A)).

(trans_gt P1 P2 C) proves (greater A B) :-
  P1 proves (greater A C), P2 proves (greater C B).
```

Program 12: Inference rules for arithmetic (excerpt).

helpful. In the object-types-as-meta-terms encoding (Section 6) in Twelf, definitions provided by Twelf can act as type abbreviations, which is a great convenience.

8 Arithmetic

For proof-carrying code, we wish to prove theorems about machine instructions that add, subtract, and multiply; and about load/store instructions that add offsets to registers. Therefore we require some rudimentary integer arithmetic in our logic; Program 12 shows some of the inference rules we use.

The `is` used in the `eval_plus` rule is the Prolog arithmetic evaluation operator, and the `>` used in the `gt_c` rule is the Prolog arithmetic comparison operator. Both of these operators require their arguments to be fully instantiated; this restriction poses no problem.

Now consider the following lemma:

```
lemma
  (Proof\ pi A\ pi I\ pi J\ pi K\
    (Proof proves
      (eq (plus (plus A (const I)) (const J)) (plus A (const K)) :-
        K is (I + J)))
  ( ... proof ... )
  plus_redex\ ...
```

The premise of this lemma is that `K is I + J`, and the conclusion, roughly speaking, is that $(A + I) + J = A + K$. When we apply this lemma in a proof, we will be attempting to prove some subformula such as

```
eq (plus (plus a (const 2)) (const 3)) (plus a (const 5))
```

and one of the λProlog subgoals will be to check that `5 is 2 + 3`. However, when proving this lemma, we make up new Prolog atoms `I,J,K` (by `pi`-binding) and then add the clause `K is I + J` to the clause database while checking that *`proof`* proves the lemma. Presumably, somewhere inside *`proof`* is a use of `eval_plus` which will check that `K is I + J`.

Teyjus λProlog does not permit the addition of new clauses for the `is` operator. This forces us to make a new operator `is'` with a default clause that applies the `is` operator:

```
type  is'   int → int → o.
A is' B :- A is B.
```

Then our inference rules and lemmas can use `is'` everywhere; adding new clauses for `is'` works, and checking an `is'` subgoal first tries the new clauses, and then tries `is`.

Some logical frameworks have powerful arithmetic primitives, such as the ability to solve linear programs [19] or to handle general arithmetic constraints [8]. For example, Twelf will soon provide a complete theory of the rationals, implemented using linear programming [24]. Some such as Elf 1.5 have no arithmetic at all, forcing us to define integers ourselves. On the one hand, linear programming is a powerful and general proof technique, but we fear that it might increase the complexity of the trusted computing base. On the other hand, synthesizing arithmetic from scratch is no picnic. The standard Prolog `is` operator seems a good compromise and has been adequate for our needs.

Floating-point or real arithmetic is not as important for our application. We are trying to prove merely the safety of programs, and not correctness. Many safety policies speak only of memory addresses and system-call arguments, which rely only on integer arithmetic. We can treat floating-point arithmetic instructions completely abstractly and still prove the safety of these programs. Only if we needed to prove properties such as the avoidance of arithmetic overflow or division by zero would we need "real" arithmetic in the logic.

9 Representing proof terms

Parameterizable data structures with higher-order unification modulo β-equivalence provide an expressive way of representing formulas, predicates, and proofs. We make heavy use of higher-order data structures with both direct sharing and sharing modulo β-reduction. The implementation of the metalanguage must preserve this sharing; otherwise our proof terms will blow up in size.

Any logic programming system is likely to implement sharing of terms obtained by copying multiple pointers to the same subterm. In Terzo, this can be seen as the implementation of a reduction algorithm described by

Wadsworth [25]. But we require even more sharing. The similar terms obtained by applying a λ-term to different arguments should retain as much sharing as possible. Therefore some intelligent implementation of higher-order terms within the metalanguage—such as Teyjus's use of explicit substitutions [16, 17]—seems essential. Perhaps even a more sophisticated representation such as *optimal reductions* [9, 3] would be useful.

10 Programming the prover

In this paper, we have concentrated on an encoding of the logic used for proof checking. But of course, we will also need to construct proofs. For example, a typical safety property is "the program never stores to a memory location outside the address range *low*...*high*." One way to construct such a proof is to write programs in a soundly typed source language, such as ML or Java, and use a type-preserving compiler to produce a soundly typed low-level intermediate language [13]. The compiler's typing judgements on this language can be used as loop invariants (and function preconditions) for a safety proof.

A different way to produce proof-carrying code is to take an unsafe machine-language program of unknown provenance and transform it using *sandboxing*, which inserts extra instructions before each load and store instruction that will bound the range of addresses accessed by the load or store (e.g., by bitwise *and*-ing the address with a constant mask) [26]. Dataflow analysis can be used to eliminate many of these extra instructions. A sandboxer could, in principle, generate a safety proof for the program it outputs; this proof would be very different from a type-based proof.

The point of these two examples is that a program prover will take advantage of specific structural properties of the class of programs produced using a certain method, whether it is type-safe compilation or flow-based sandboxing. To do so, we write a *program* with special-purpose algorithms to prove our special class of theorems. For implementing this prover, we have found that the Prolog-style control primitives (such as the cut (`!`) operator and the `is` predicate), which are also available in λProlog, are quite important.

λProlog also provides an environment for implementing tactic-style interactive provers [6]. This kind of prover is useful for proving the lemmas that are used by the automatic prover.

Neither Elf nor Twelf have any control primitives. However, there are plans to add an operator to Twelf similar to Prolog cut [21], which would allow us to implement the automatic prover in the same way as in λProlog. It is not possible to build interactive provers in Elf or Twelf, so proofs of lemmas used by the automatic prover must be constructed by hand.

11 Conclusion

The logical frameworks discussed in this paper are promising vehicles for proof-carrying code, or in general where it is desired to keep the proof checker as small and simple as possible. We have proposed a representation for lemmas and definitions that should help keep proofs small and well-structured, and it appears that each of these frameworks has features that are useful in implementing, or implementing efficiently, our machinery.

Although the lemma system shown in this paper is particularly lightweight and simple to use, its lack of polymorphic definitions and lemmas has led us to further investigate the encodings (sketched in Section 6) that use object-level polymorphic types [2].

Acknowledgements

We thank Robert Harper, Frank Pfenning, Carsten Schürmann for advice about encoding polymorphic logics in a monomorphic dependent-type metalanguage; Robert Harper and Daniel Wang for discussions about untyped systems; Ed Felten, Neophytos Michael, Kedar Swadi, and Daniel Wang for providing user feedback; Gopalan Nadathur and Dale Miller for discussions about λProlog.

References

[1] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conf. on Computer and Communications Security*, Nov. 1999.

[2] Andrew W. Appel and Amy P. Felty. Polymorphic lemmas in LF and λProlog. In preparation, 1999.

[3] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[4] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–12, New York, 1982. ACM Press.

[6] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *J. Automated Reasoning*, 11(1):43–81, August 1993.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, January 1993.

[8] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119. ACM, January 1987.

[9] John Lamping. An algorithm for optimal lambda calculus reduction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 16–30. ACM Press, Jan 1990.

[10] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, to appear, 1999.

[11] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic Computat.*, 1(4):497–536, 1991.

[12] 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.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, January 1998.

[14] Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*. MIT Press, 1988.

[15] Gopalan Nadathur and Frank Pfenning. *The Type System of a Higher-Order Logic Programming Language*, pages 243–283. MIT Press, 1992.

[16] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.

[17] Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.

[18] George Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM Press.

[19] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[20] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[21] Frank Pfenning. personal communication, June 1999.

[22] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[23] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[24] Roberto Virga. Twelf(X): Extending Twelf to rationals and beyond. In preparation, 1999.

[25] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[26] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 203–216, New York, 1993. ACM Press.

[27] Philip Wickline. The Terzo implementation of λProlog. http://www.cse.psu.edu/∼dale/lProlog/terzo/index.html, 1999.