Lightweight Lemmas in λProlog

Andrew W. Appel
Bell Laboratories and Princeton University

Amy P. Felty
Bell Laboratories

May 14, 1999

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. The terms of the meta-language (λProlog) can be used to express the statement of a lemma, and the type checking of the meta-language can directly implement the type checking of the lemma. The ML-style prenex polymorphism of λProlog allows easy expression of polymorphic inference rules, but a more general polymorphism would be necessary to express polymorphic lemmas directly. We discuss both the Terzo and Teyjus implementations of λProlog as well as related systems such as Elf.

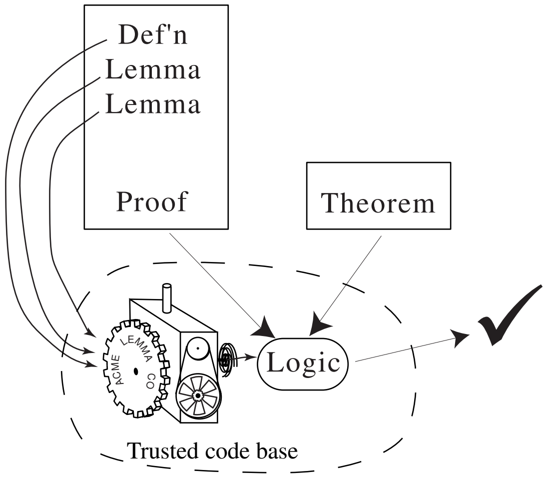

Figure 1: Lemma machinery is inside the TCB.

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* [Nec97] and distributed authentication frameworks [AF99] – 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, 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 etc.; 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.

```
kind  form   type.
kind  proof  type.

type  eq      A→A→form.
type  and     form→form→form.
type  imp     form→form→form.
type  forall  (A→form)→form.

type  proves  proof→form→o.

infixl  and     7.
infixr  imp     8.
infix   proves  5.

type assume      form→o.

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

Program 2: Type declarations for core logic.

The λProlog language [NM88] also has several features that allow concise and clean implementation of logics, proof checkers, and theorem provers [Fel93]. We use λProlog [NM88], but the ideas should also be applicable to logical frameworks such as Elf/Twelf [Pfe91, PS99]. 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*, *metalevel formulas as terms*, *prenex* and *non-prenex polymorphism*, *nonparametricity*, and *type abbreviations*.

2 A core logic

The clauses we present use the syntax of the Terzo implementation of λProlog [Wic99]. 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. Both Terzo and Teyjus [NM99] (which we discuss later) extend simple types to include ML-style prenex polymorphism [DM82], which we make use of in our implementation. Second, it permits implication and universal quantification over objects of any type.

We introduce new types and new 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 representing polymorphic types. In program goals and clauses, tokens with initial capital letters will denote either bound or free variables. All other tokens will denote constants. λ-abstraction is written using backslash \ as an infix operator. Universal quantification is written using the constant `pi` in conjunction with a λ-abstraction, *eg.*, `pi X\` represents universal quantification over variable `X`. The symbols `,` and `=>` represent conjunction and implication, respectively. 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 tiny core logic of a sequent calculus for a higher-order logic, illustrated in Program 2 and 3. We call this the *object logic* to distinguish it from the *metalogic* implemented by λProlog.

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 Prolog 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

$$\frac{}{A, \Gamma \vdash A}$$

```
initial proves A :- assume A.
```

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

```
(and_r Q1 Q2) proves (A and B) :-
    Q1 proves A,  Q2 proves B.
```

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \rightarrow B}$$

```
(imp_r Q) proves (A imp B) :-
    assume A  =>  Q proves B.
```

$$\frac{A, B, \Gamma \vdash C}{(A \wedge B), \Gamma \vdash C}$$

```
(and_l A B Q) proves C :-
    assume (A and B),
    assume A => assume B => Q proves C.
```

$$\frac{\Gamma \vdash A \qquad B, \Gamma \vdash C}{(A \rightarrow B), \Gamma \vdash C}$$

```
(imp_l A B Q1 Q2) proves C :-
    assume (A imp B),  Q1 proves A,
    assume B => Q2 proves C.
```

$$\frac{\Gamma \vdash A(Y) \qquad \text{for any } Y \text{ not in the conclusion}}{\Gamma \vdash \forall x.\ A(x)}$$

```
(forall_r Q) proves (forall A) :-
    pi Y\ ((Q Y) proves (A Y)).
```

$$\frac{A(T), \Gamma \vdash C}{(\forall x.\ A(x)),\ \Gamma \vdash C}$$

```
(forall_l A T Q) proves C :-
    assume (forall A),
    assume (A T)  =>  Q proves C.
```

$$\frac{\Gamma \vdash A \qquad A, \Gamma \vdash C}{C}$$

```
(cut Q1 Q2 A) proves C :-
    Q1 proves A,
    assume A  =>  Q2 proves C.
```

$$\frac{\Gamma \vdash X = Z \qquad \Gamma \vdash H(Z)}{\Gamma \vdash H(X)}$$

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

$$\frac{}{\Gamma \vdash X = X}$$

```
refl proves (eq X X).
```

Program 3: Inference rules of core logic.

`((eq x y) imp (eq x y))`; then λProlog will execute the (instantiated) body of the `imp_r` clause:

```
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.

In our `forall_r`, `forall_l`, and `congr` rules for universal quantification and congruence we take advantage of λProlog's higher-order data structures. That is, in the formula `forall A`, the term `A` is a lambda expression taking a bound variable and returning a formula; in this case the bound variable is intended to be the quantified variable. 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* [PE88]; it is very valuable as a mechanism for avoiding the need to describe the mechanics of substitution explicitly in the object logic [Fel93].

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 we have already saved one constructor and two inference rules! But the user of our logic may wish to construct a proof that uses universal quantification at an arbitrary higher-order type, so we cannot possibly anticipate all the types at which monomorphic `forall` constructors would be needed. Therefore, polymorphism is not just a syntactic convenience – it makes the logic more expressive.

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.

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

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

Adequacy. It is important to show that our encoding of higher-order logic in λProlog is *adequate*. To do so, we must show that there is a one-to-on mapping between proof trees in higher-order logic using the inference rules of Figure 3 and our encoded proof terms of type `proof`. Showing the existence of such a mapping should be straightforward. In particular, since we have encoded our logic using prenex polymorphism, it requires expanding out instantiated copies of all of the polymorphic expressions in terms of type `proof`; the expanded proof terms will then map directly to sequent proof trees.

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

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 logic rule shown in Program 4. The proof-constructor `lemma` takes three arguments:

1. an inference rule `Inference` (of type `A→o`, where `o` is the type of Prolog goals and `A` is any type) parameterized by a proof-constructor (of type `A`);

2. a higher-order `Proof` (of type `A`) built from core-logic proof constructors (or using other lemmas);

3. and a proof of the main theorem `C` that is parametrized by a proof-constructor (of type `A`).

For example, we can prove a lemma about the symmetry of equality:

Lemma `symmx`: $\dfrac{B = A}{A = B}$.

The proof uses congruence and reflexivity of equality:

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

This theorem can be checked as a successful λProlog query in the logic of Programs 2 and 3: for an arbitrary `P`, add `(P proves (eq B A))` to the logic; checking the proof of congruence will need to use this fact.

The syntax F => G means exactly the same as G :- F, so we could just as well write this query as

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

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

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

The solution of this query proceeds in four steps: the variable `Inference` is unified with a lambda-term; `Proof` is unified with a lambda-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 Prolog 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 P\ pi A\ pi B\
  (symmx P A B) 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.

```
(lemma
  (Proof\ pi P\ pi A\ pi B\
     (Proof P A B) proves (eq A B) :-
         P proves (eq B A))
  (P\A\B\(congr P refl (eq A) B A))
 symmx\

 forall_r I\ forall_r J\
    imp_r (symmx initial J I)
)
proves
 forall I\ forall J\ eq I J imp eq J I.
```

Theorem 2. $\forall I \forall J \forall K. I = J \rightarrow J = I$.

To "make a new Prolog 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` lemma is a bit unwieldy, since it requires `A` and `B` as arguments. We can imagine writing a primitive inference rule

```
pi P\ pi A\ pi B\
  (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` – to the logic, as shown in Program 5.

These can be used in the following stereotyped way to extract components of the formula to be proved. First bind variables with `elam`, then match the target formula with `extract`. For example, see Theorem 3.

The `extractGoal` asks the checker to run Prolog code to help construct the proof. Of course, if we want proof-checking to be finite we must restrict what kinds of Prolog code can be run, and this is accomplished by `valid_clause`. 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.

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

(elam Q) proves B :- (Q A) proves B.

(extract B P) proves B :- (P proves B).
extractGoal Goal P proves B :-
   valid_clause Goal, Goal, P proves B.
```

Program 5: Proof constructors for implicit arguments of lemmas.

```
(lemma
  (Proof\ pi P\ pi A\ pi B\
    (Proof P) proves (eq A B) :-
            P proves (eq B A))
  (P\ elam A\ elam B\ extract(eq A B)
   (congr P refl (eq A) B A))
 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. $\forall I \forall J \forall K. I = J \rightarrow J = I$.

3.1 Dynamic clauses and goals

Our technique allows lemmas (and, as we will show in the next section, definitions) to be contained *within* the proof. We do not need to install new "global" lemmas and definitions into the proof checker. If, for example, `symm` were a global atom instead of a locally bound variable, it would presumably need a global type declaration; how would this be installed and removed? 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.

Our lemma machinery uses several interesting features of λProlog:

Metalevel formulas as terms. The expression `Inference`, which equals

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

is just a data structure (parameterized by `Proof`); it does not "execute" anything, in spite of the fact that it contains the Prolog connectors `:-` and `pi` (and similar data structures containing comma and semicolon can also be built). The type of the term `Inference(Proof)` is just `o`, the type of Prolog goals; but even so no execution is implied. It is only when `Inference(Proof)` appears in "goal position" that it becomes the current subgoal on the execution stack. 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 `lemma` proof-constructor checks the validity of a lemma by executing the goal `Inference(Proof)`, we are executing a goal that is built from a run-time-constructed data structure. This is an important feature of λProlog for our lemma system.

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 Prolog database. This feature – distinct from dynamically constructed goals and perhaps especially hard to implement in a compiled λProlog system – is also important for our lemma machinery.

Section 5 will show how we can relax the requirements on dynamically constructed clauses and goals, respectively.

3.2 Valid clauses.

Since the type of `Inference(Proof)` is `o`, the lemma `Inference` might conceivably contain any Prolog clause at all, including those that do input/output. Such Prolog 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 Prolog code that accesses private local resources.

```
type valid_clause   o→o.
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 F).
valid_clause (assume _).
```

Program 6: Valid clauses.

To limit the kind and amount of execution possible in the executable part of a lemma, we introduce the notion of valid clauses (Program 6).

A clause is valid if contains `pi`, comma, `proves`, `assume`, `:-`, `=>`, and nothing else. Of course, a `proves` clause contains subexpressions of type `proof` and `form`, and an `assume` clause has a subexpression of type `form`, so all the connectives in proofs and formulas are also permitted. Absent from this list are Prolog input/output (such as `print`) and the Prolog semicolon (backtracking search).

3.3 Doing without lemmas

In principle, we do not need lemmas at all. The lemma

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

can be expressed as a single formula,

```
forall A\ forall B\ eq B A imp eq A B
```

This formula can be proved, then cut into the proof of a theorem using the ordinary `cut` of sequent calculus. To make use of the fact requires two `forall_l`'s and an `imp_l`. This approach adds a significant amount of complexity to proofs, which we wish to avoid.

Soundness and adequacy. One way to extend soundness and adequacy to the system with lemmas is to show that it is possible to replace any lemma with a cut-in formula in the way we have discussed

```
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_l\ pi Name\ pi B\ pi Q\
         pi D\ pi F\ pi A\
    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 (symm initial)
            initial B F Name)
     Q
     (B F)))
  def_l\

lemma
  (Def_r\ pi Name\ pi B\ pi P\ pi F\
     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 initial P B Name F)))
  def_r\
```

Program 7: Machinery for definitions.

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

$$\forall X \forall Y \forall Z.\ f\,X\,(f\,Y\,Z) = f\,(f\,X\,Y)\,Z$$

or in λProlog notation,

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

We can abstract this formula over f to make a 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 associative
   (F\ (forall X\ forall Y\ forall Z\
     eq (F X (F Y Z)) (F (F X Y) Z)))
```

To use definitions in proofs we need three new proof rules:

define to bind a (higher-order) formula to a name,

def_l expand a definition (or, in sequent logic terms, to replace a use of the defined name on the left of the sequent arrow with the formula it stands for), and

def_r turn a proof of a formula into a proof of the definition that stands for it.

All three of these proof-constructors are just lemmas provable in our system using congruence of equality, as Program 7 shows.

To check a proof

```
define Formula (Name\ RestProof(Name))
```

the system interprets the `pi D` within the `define` lemma to create a new Prolog atom `D` to stand for the name of the definition. It then adds `assume(eq D Formula)` to the Prolog 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 associative (A\ A f) P)
  proves  (associative f)
```

the prover first checks that `(A\ A f)(associative)` matches `(associative f)` and that

```
assume (eq associative 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 associative (A\ A f) P
```

proves some formula `D`, the checker first calculates `(A\ A f)(associative)`, that is, `associative f`, and checks that

```
assume(associative f)
```

is among the assumptions in the Prolog database. Then it verifies that

```
assume(eq associative 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`.

5 Dynamically constructed clauses and goals

Teyjus λProlog [NM99] restricts the form of clauses dynamically introduced using `=>` and `:-`, of goals with variable head, and of syntactic terms of type `o`. In this section we show how these restrictions, and other similar restrictions not necessarily imposed by Teyjus, can be evaded.

5.1 Dynamically constructed clauses

Teyjus doesn't allow programs to construct arbitrary clauses and add them dynamically—because it would require invocation of the compiler at runtime—but does allow at least the dynamic construction and addition of clauses with a constant at the head. It is still possible to implement lemmas fairly directly as in Section 3 by implementing a meta-interpreter. The code for the meta-interpreter and the modified clause for checking lemmas is in Program 8.

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

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

P proves F :-
   cl Cl, backchain (P proves F) Cl, !.
```

Program 8: A meta-interpreter for dynamic clauses.

As before, the clauses representing lemmas that occur inside proofs are used in two ways. Lemmas are checked by presenting them as goals and they are also used during the checking of the rest of the proof. The difference now is that in the latter operation they are not used directly as clauses by λProlog. Instead, we view them as terms of type `o` with no special meaning and write code to manipulate them ourselves. First, we add a new predicate `cl` of type `o` → `o` used for adding our new clauses as atomic clauses of λProlog. Note that this predicate is used in the last line of the modified `lemma` clause. The remaining code implements their use in checking proofs that use the lemmas. The last clause in the figure is a new clause for the `proves` predicate which is required for nodes representing lemma applications. 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 λProlog language itself.

5.2 Dynamically constructed goals

Note that in the last two clauses for `backchain` in Program 8, the goal `G'` which appears as an argument inside the head of the clause also appears as a goal in the body of the clause. Teyjus prohibits this kind of goal formation. To run our lemma system under Teyjus, we have implemented an interpreter that handles solving of goals as well as backchaining over clauses.

We implement a predicate `solvegoal` of type `o` → `o` and invoke `(solvegoal G')` instead of simply `G'` in the last clause of `backchain` in Program 8. We omit the details here.

5.3 Metalevel formulas as terms

Our system takes advantage of the ability to use metalevel formulas as terms. For example the `symm` lemma

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

uses the data constructors `:-` and `pi` which are normally used to construct λProlog programs. Teyjus prohibits the operators `:-` and `=>` as data constructors. This forces us to write lemmas using some other operator, which is easy enough for the `backchain` predicate to interpret, but is an inconvenience for the user, who must use different syntax in lemmas than in inference rules.

Uninterpretable cut. Although we have shown that it is possible to interpret dynamically constructed metasyntax even though it cannot be executed, the special Prolog cut `!` operator cannot be implemented this way. We could certainly have the following clause in the interpreter,

```
solvegoal ! :- !.
```

but the behavior would not be as desired. The backtracking behavior of the goal `(G1, !, G2)` is different from that of the goal

```
solvegoal G1, solvegoal !, solvegoal G2.
```

The lack of Prolog cut does not affect proofs themselves – which don't need it – but it will certainly affect the implementation of theorem provers, which may rely on it heavily.

6 Meta-level types

ML-style prenex polymorphism has been important in the encoding of our object logic, in particular, in the implementation of the `forall_r` and `congr` rules in Program 3 and in implementing lemmas as shown in Program 4. In this section we discuss the limitations of prenex polymorphism for implementing lemmas which are themselves polymorphic; and we discuss ways to overcome these limitations.

6.1 Non-prenex polymorphism and lemmas

We can generalize prenex polymorphism 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 lambda 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 ΠA. proof→proof→(A→form)→
A→A→proof.

type  forall_r  ΠA. (A→proof)→proof.

ΠA. pi Q: proof\ pi P: proof\
  pi H: A→form\ pi Z: A\ pi X: A\
 (congr A Q P H X Z) proves (H X) :-
    Q proves (eq A X Z),  P proves (H Z).

ΠT. pi Q: T→proof\ pi A: T→form\
(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 9 is pseudo-λProlog in which all type quantifiers and type bindings are shown explicitly. The crucial point is that the line marked *here* contains a lambda-expression, λProof.body, in which the type of `Proof` is `ΠT. proof→proof`. Requiring a function argument to be polymorphic is an example of non-prenex polymorphism, which is permitted in second-order lambda calculus but not in an ML-style type system.

```
type  lemma ΠA. (A→o)→A→(A→proof)
                 →proof.

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

(lemma
  T
  (Proof: ΠT. proof→proof \       ← here!
      ΠT. pi P:proof\ pi A:T\ pi B:T\
    (Proof T P) proves (eq T A B) :-
                P proves (eq T B A))
  (ΛT. P:proof\ elam A:T\ elam B:T\
   extract(eq T A B)
    (congr T P refl (eq T A) B A))
 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 9: Explicitly typed version of Theorem 3.

Since type inference for such non-prenex polymorphism is undecidable, any fully polymorphic language must have explicit type bindings and explicit type quantification – although it may be possible to do partial type inference to avoid the need for all type arguments to be written explicitly [Pfe88].

Why did Theorem 3 work at all, if it uses a polymorphic lemma `symm`? The answer is that `symm` isn't really behaving polymorphically, as we can see from Theorem 6 which attempts to use it at two different types.

This proof fails to check in our implementation, because the first use of `symm` unifies its polymorphic type variable with the type `T` of `x`, and then the use of `symm` at

```
(lemma
  (Proof\ pi P\ pi A\ pi B\
     (Proof P) proves (eq A B) :-
         P proves (eq B A))
  (P\ elam A\ elam B\ extract (eq A B)
     (congr P refl (eq A) B A))
 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. $\forall f, g, x. f = g \rightarrow f(x) = x \rightarrow (g = f \wedge x = f(x))$.

type T→T fails to match in the proof checker.

Polymorphic definitions (using `define`) run into the same problems and also require non-prenex polymorphism. 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 get around requiring such an extension to the meta-logic by always duplicating each lemma at several different types within the same proof.

Adequacy. 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 constructions [CH88], and prove the soundness and adequacy of our system with respect to that logic (although we have not done such a proof).

6.2 Should your metalanguage be typed?

The prenex-polymorphic λProlog language can represent only a restricted set of lambda-expressions, sufficient for polymorphic inference rules but not polymorphic lemmas and definitions. Perhaps the problem lies in using a statically typed metalanguage. Lamport and Paulson [LP99] 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.

If we had a completely dynamically typed (or "untyped") version of λProlog, we could represent fully polymorphic functions, and our proofs using polymorphic lemmas would work immediately. Unfortunately, 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 $\exists x. x = (x \rightarrow \text{false})$ from which anything can be proved.

Therefore, if we built 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.

6.3 Alternate encodings

There are also several ways to encode our polymorphic logic and allow for polymorphic lemmas without going to the extreme of eliminating types from the meta-language. One possiblity 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 exp,form  tp.
type eq     tp→tm→tm→tm.
type congr  tp→proof→proof
               →(A→tm)→A→A→proof.
```

```
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 proof. To illustrate, the new constructors and corresponding type checking clauses are given below.

```
type app  tp → tp → tm → tm → tm.
type lam  tp → tp → (tm → tm) → tm.
typecheck (app T1 T2 F X) T2.
typecheck (lam T1 T2 F) (arrow T1 T2).
```

This encoding loses some economy of expression because of the extra constructors needed for the encoding, and requires a limited amount of type-checking, 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.

Another alternative is to use a similar encoding in a metalanguage such as Elf/Twelf [Pfe91, PS99]. 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 explict constructors for app and abs as well as primitive rules for βη-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).
```

6.4 Type abbreviations

Regardless of whether prenex or full polymorphism is used, a logical framework should allow type abbreviations to be defined. The theorems we prove in our application are from the domain of proof-carrying code, where we are using the same kinds of higher-order types that show up in traditional denotational semantics. Thus, we may have the type exp of machine integers; machine memory is a function exp→exp. In our formulation, an object type ty is a three-argument predicate taking an allocated predicate (exp→form), a memory, and an exp.

Therefore, the predicate hastype, which takes an allocated predicate, a memory, and an exp, has the following type:

```
kind exp  type.

type store = exp→exp.
type allocated = exp→form.
type ty = allocated→store→exp→form.

type hastype
   allocated→store→exp→ty→form.
```

However, λProlog does not have type abbreviations of the form type regs = exp→exp, so our program is full of declarations like this one:

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

This is rather an inconvenience. ML-style (nongenerative) type abbreviations would be very helpful in a logical framework.

7 Other issues

Although we are focusing on the interaction of the meta-level type system with the object logic lemma system, there are other aspects of meta-language implementation that are relevant to our needs for proof generation and proof checking.

Arithmetic. For our application, 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.

Some logical frameworks have powerful arithmetic primitives, such as the ability to solve linear programs [Nec98] or to handle general arithmetic constraints [JL87]; some have no arithmetic at all, forcing us to define integers as sequences of booleans. 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.

Representing proof terms. Parametrizable 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 meta-language must preserve this sharing; otherwise our proof terms will blow up in size.

Any Prolog system implements sharing of terms obtained by copying multiple pointers to the same subterm. In λProlog, this can be seen as the implementation of a reduction algorithm described by Wadsworth [Wad71]. 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 meta-language—such as Nadathur's use of explicit substitutions [Nad97]—seems essential. Perhaps even a more sophisticated representation like *optimal reductions* [Lam90, AG98] will be useful.

Programming the prover. In this paper, we have concentrated on an encoding of the logic used for proof checking. But of course, we will need to construct proofs, too, which is sufficently difficult [Göd31] that the full power of an imperative programming language (such as λProlog with the cut (`!`) operator) seems necessary.

Our proof manipulation algorithms often need to walk over arbitrary proof terms, an operation which is complicated by our use of polymorphism. We illustrate by a very simple example: a function that tells the arity (number of function arguments) of an arbitrary value.

```
type  arity  A→int→o.
arity F N :- arity (F X) N1, N is N1 + 1.
arity X 0.
```

The first clause matches only when `F` is a function; the second clause matches any value. An ML program of type $\alpha \rightarrow \mathrm{int}$ would never be able to dispatch on the representation of its argument like this, because ML polymorphism is based on the principle of *parametricity*: the behavior of a polymorphic ML function is independent of the particular type at which it is used. This property is essential in providing data abstraction [Rey83]. Without parametricity, logic programming languages such as λProlog lose the ability to do data abstraction, but nonparametricity is very useful to us in manipulating polymorphic proof terms.

8 Comparison of existing logical framework systems

We have used the Terzo λProlog system to build and check proofs in a prototype system. There are other implementations of λProlog as well as implementations of other logical frameworks in which this work can be done.

Terzo [Wic99] is an intepreter for λProlog implemented in Standard ML. Terzo permits metalevel formulas as terms, dynamically constructed goals and clauses, nonparametricity, and prenex polymorphism. Terzo uses Wadsworth-style representation of lambda expressions, rather than the potentially more efficient explicit substitutions, so proof checking tends to use a lot of memory. Terzo's interpreter uses linear search to find a clause that matches the current subgoal; this is another source of inefficiency. The current implementation of Terzo is adequate for prototyping, but might not support large-scale work in proof-carrying code.

Teyjus [Nad99] is a new system under development at the University of Chicago that compiles to bytecodes for a higher-order variant of the Warren Abstract Machine using explicit substitutions for representation of lambda terms [Nad97]. It restricts the syntax of terms and the use of dynamic clauses and goals, so we must implement an interpreter as described in Section 5.

Both Terzo and Teyjus λProlog have prenex polymorphism but not full (nonprenex) polymorphism; neither permits type abbreviations. Both support the Prolog `is` for integer arithmetic. Both implementations support basic Prolog imperative constructs such as the cut `!` and input/output, which we have found useful in building a tactical theorem prover.

The fact that Terzo is an interpreter, not a compiler, is what allows it to so easily handle dynamically constructed goals and formulas. Teyjus is implemented with many techniques that should drastically improve its efficiency; these include better data structures for finding clauses that match (as manifested in the Warren Abstract Machine), better representations for higher-order data structures (i.e., explicit substitutions), and compilation to an interpreted byte code (instead of interpreting abstract syntax trees). For our application, since we will so frequently introduce new clauses, an interpretation of AST's might be preferable to byte-code compilation. But many of the techniques used in Teyjus (for efficient clause matching and for substitution) would be useful even in an interpreter.

Elf [Pfe91] is an implementation of LF [HHP93], the Edinburgh logical framework.

Elf 1.5 has full (nonprenex) statically checked polymorphism with explicit type quantification and explicit type binding. Explicit quantification and binding is necessary because type inference for full polymorphism is undecidable. However, Elf contains a partial type inference algorithm that often permits the programmer to avoid providing all type arguments of polymorphic functions.

Elf has the nonparametricity we need to implement meta-operations in the prover.

Elf has no built-in arithmetic at all, which is a severe handicap.

Twelf is the successor to Elf. Like Elf, it has higher-order data structures with a static type system, but Twelf is monomorphic. The lack of even prenex polymorphism forces us to use the object-types-as-terms representation described in Section 6.3. Twelf has no imperative constructs (such as the Prolog cut) which would allow the implementation of specialized theorem provers.

Twelf allows definitions, which may help with implementing lemmas and definitions for our logic. For example, we can use the definitions of Twelf to name the proof of the symmetry lemma and add it to the signature of our core logic and then use it directly in subsequent proofs.

```
symmx: {X,Z:T} provable (eq Z X)
        →provable (eq X Z)
      = [X,Z:T][P:provable (eq Z X)]
         (eq_congr Z X (eq X) P (refl X)).
```

Twelf will soon provide a complete theory of the rationals, implemented using linear programming [Pfe99].

9 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. But none of these systems has all the features we need; it would be interesting to design and construct a system with the best features of all of them, or to modify one of the existing frameworks.

Acknowledgements. We thank Robert Harper, Frank Pfenning, Carsten Schürmann for advice about encoding polymorphic logics in a monomorphic dependent-type metalanguage; Doug Howe, David MacQueen, and Jon Riecke for advice about recursive and nonrecursive definitions; Robert Harper and Daniel Wang for discussions about untyped systems; Ed Felten, Neophytos Michael, Kedar Swadi, and Daniel Wang for providing user feedback.

References

[AF99] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. Technical report, Princeton University Dept. of Computer Science, April 1999.

[AG98] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[DM82] 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.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica and verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, January 1993. To appear. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[JL87] 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.

[Lam90] 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.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Trans. on Programming Languages and Systems*, to appear, 1999.

[Nad97] Gopalan Nadathur. An explicit substitution notation in a lambdaProlog implementation. http://www.cs.uchicago.edu/˜gopalan, December 1997.

[Nad99] Gopalan Nadathur. The Chicago Lambda Prolog system. http://www.cs.uchicago.edu/˜gopalan, 1999.

[Nec97] 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.

[Nec98] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[NM88] 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.

[NM99] Gopalan Nadathur and Dustin. J. Mitchell. System description: Teyjus — a compiler and abstract machine bases implementation of λProlog. In *The 16th International Conference on Automated Deduction*. Springer-Verlag, July 1999.

[PE88] 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.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[Pfe91] 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.

[Pfe99] Frank Pfenning. personal communication, March 1999.

[PS99] 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.

[Rey83] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *IFIP Conference*, pages 513–24, 1983.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[Wic99] Philip Wickline. The terzo implementation of λProlog. http://www.cse.psu.edu/-∼dale/lProlog/terzo/index.html, 1999.