A Declarative Alternative to “assert” in Logic Programming

Scott Dietzen and Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890, USA
Internet: dietzen@cs.cmu.edu, fp@cs.cmu.edu

Abstract

The problem with the standard means by which Prolog programs are extended — **assert** — is that the construct is not semantically well-behaved. A more elegant alternative (adopted, for example, in λProlog) is implication with its intuitionistic meaning, but the assumptions so added to a logic program are of limited applicability. We propose a new construct **rule**, which combines the declarative semantics of implication with some of the power of **assert**. Operationally, **rule** provides for the extension of the logic program with results that deductively follow from that program. **rule**, used in conjunction with higher-order programming techniques such as continuation-passing style, allows the natural and declarative formulation of a whole class of logic programs which previously required **assert**. Example applications include memoization, partial evaluation combined with reflection, resolution, ML type inference, and explanation-based learning.

1 Introduction

There are many features of logic programming languages like Prolog which have very little to do with its logical foundation. The so-called ‘meta-logical’ predicates such as **call**, **assert**, **retract**, **univ**, or **var** form an interesting class of Prolog constructs. They are concerned with the manipulation of logic programs, including the currently running program itself. Even though these constructs often operate at an intuitively different level (the meta-level) than other predicates, this level distinction is not made explicit in Prolog. This generates many well-known problems in logic program analysis and compilation (see [10], for example).

There has been a recent surge of interest in giving a ‘logical’ or ‘formal’ account of these constructs in the context of logic programming. λProlog [16] uses a fragment of intuitionistic higher-order logic and gives a logical foundation for **call** (through higher-order predicates) and some uses of **assert** (through embedded implication). HiLog [3] uses an even narrower fragment of higher-order logic and can give a declarative account for many uses of **univ** and **call**. The language Gödel [2] follows a different approach by explicitly separating the meta-level from the object-level.

In this paper we are concerned with giving a logical foundation to uses of **assert** for which we have not found satisfactory account in the literature (although the reader may also see [11, 19]). We propose the **rule** construct, which can be viewed operationally as extending the logic program with consequences of that program. The construct thereby introduces an element of forward reasoning into logic programming, albeit in a very limited and tightly controlled fashion. **rule** allows us to program in a natural and declarative way many applications which previously had to rely on extra-logical features. Examples are memoization, partial evaluation combined with reflection, resolution and many other typical applications of meta-programming in logic programming.

Within [5, 4], we develop a closely related construct, **rule_ebg**, which similarly extends a program by one of its consequences. The difference is how the interpreter arrives at the consequence to be assumed: **rule** creates a general consequence by universally quantifying over logic variables; **rule_ebg**, on the other hand, additionally abstracts over constants and terms, so long as their structure is irrelevant for the proof of the proposed assumption (in the usual manner of explanation-based generalization).

Higher-order logic programming. The vehicle for this work is λProlog, a logic programming language supporting higher-order functions and predicates introduced by Nadathur & Miller [16]. λProlog extends Prolog in several principle directions: It provides the simply-typed λ-calculus as a data type and incorporates higher-order unification. It also generalizes beyond higher-order Horn clauses by allowing so-called embedded implication and universal quantification.

The primary aspects of λProlog relevant to the work reported herein are embedded implication and higher-order programming — that is, the ability to create goals and programs and pass them as arguments. Although the constructs **rule** and **rule_ebg** are proposed and applied in the framework of λProlog, the underlying ideas are general and, we believe, relevant to other logic programming languages.

Logical preliminaries and notation. Within our examples, constants are in boldface while variables are in italics. Juxtaposition denotes term application, which associates to the left: $\mathbf{abc}$ is read as $(\mathbf{ab})\mathbf{c}$, as in the λ-calculus. Thus the Prolog term $\mathbf{p}(\mathbf{a}, \mathbf{b})$ is written as $\mathbf{p\,a\,b}$ in λProlog. Both $\Rightarrow$ and $\Leftarrow$ represent (intuitionistic) implication; the latter is equivalent to Prolog's '**:-**'.

Higher-order Horn clauses may be defined inductively as

$$\begin{array}{lcl} G & ::= & \textbf{true} \mid A \mid G_1\,,G_2 \mid G_1\,;G_2 \mid \exists x\,[:\tau].\,G \\ D & ::= & \textbf{true} \mid A \mid D_1\,,D_2 \mid D \Leftarrow G \mid \forall x\,[:\tau].\,D \end{array}$$

where G ranges over arbitrary goals, D over program clauses, τ over simply types, and A over atoms. (Atoms are propositional terms, or predicates, that do not contain any logical operators at the top-level.) We omit types in the sequel since they can be inferred through type reconstruction. Finally, we use $\mathcal{P}$ to denote an arbitrary logic program (list of D's), and E to stand for an expression that must be both a legal program *and* goal.

The extension to allow embedded implication and embedded universal quantification leads to the higher-order hereditary Harrop formulas — the basis of λProlog:

$$\begin{array}{lcl} G & ::= & \textbf{true} \mid A \mid G_1\,,G_2 \mid G_1\,;G_2 \mid \exists x\,[:\tau].\,G \mid D \Rightarrow G \mid \forall x\,[:\tau].\,G \\ D & ::= & \textbf{true} \mid A \mid D_1\,,D_2 \mid D \Leftarrow G \mid \forall x\,[:\tau].\,D \end{array}$$

We use $\mathcal{P} \vdash G$ to mean that there exists an (intuitionistic) proof of G given $\mathcal{P}$; that is, G follows from $\mathcal{P}$.[1] In order to speak about the interpretation of λProlog, we use $\mathcal{P} \Vdash G$ to represent the problem of solving G given the program $\mathcal{P}$. If this succeeds with substitution θ, then we say that $\mathcal{P} \Vdash_\theta G$ holds. We denote the application of a substitution θ to an expression M (containing free variables) as θM, and we use $\psi\theta$ for the composition of substitutions $\psi \circ \theta$.

The $\Vdash_\theta$ relation can be formally defined as an inference system [4]. By induction over the construction of the derivation of $\mathcal{P} \Vdash_\theta G$ within that inference system, one can show that $\theta\mathcal{P} \vdash \theta G$ both for higher-order Horn logic and for the richer logic with embedded implication and universal quantification.[2] We will refer to this property as *soundness of the interpreter*. The proof is not difficult and essentially contained in [13]. The dual theorem (*completeness of the interpreter*) usually fails in logic programming: even if $\theta\mathcal{P} \vdash \theta G$, the interpreter may fail to terminate and thus $\mathcal{P} \not\Vdash_\theta G$.

2 Existing Approaches to Extending Logic Programs

2.1 Prolog's "assert"

Prolog permits the modification of the current logic program through the primitives **assert** and **retract**: **assert** D adds clause D to the program,

[1] For Horn logic and even higher-order Horn logic, intuitionistic and classical provability coincide [13], so the by-word "intuitionistic" is only important for logics extended with embedded implication and embedded universal quantification, such as λProlog.

[2] In our formulation of Horn logic, all assumptions are closed (explicitly quantifying over free variables), and thus $\theta\mathcal{P} = \mathcal{P}$. However, as we shall illustrate, this is not the case for λProlog with its embedded implication.

while **retract** D removes D.[3] We are particularly interested in the following applications of **assert**:

- *Memoization* — To avoid the re-computation of previously solved goals, derived results are memoized, or cached, by applying **assert** to goals deductively following from a program $\mathcal{P}$. We call this a *conservative extension* of the original program $\mathcal{P}$. For example, the following definition of the Fibonacci function will not recompute values (unless the user causes it to backtrack):

$$\begin{array}{llll} \textbf{fib} & 0 & 1. & \\ \textbf{fib} & 1 & 1. & \\ \textbf{fib} & m & n & \Leftarrow \; m > 1,\; m_1 \textbf{ is } m-1,\; m_2 \textbf{ is } m-2, \\ & & & \quad \textbf{fib}\, m_2\; n_2,\; \textbf{asserta}\, (\textbf{fib}\, m_2\; n_2), \\ & & & \quad \textbf{fib}\, m_1\; n_1,\; \textbf{asserta}\, (\textbf{fib}\, m_1\; n_1),\; n \textbf{ is } n_1 + n_2. \end{array}$$

  Memoization may be considered an example of *forward reasoning* — a paradigm in which a knowledge-base (in this case, a logic program) grows by computing and assimilating facts that follow deductively. Although individual goals are derived through the standard back-chaining of Prolog, their assimilation represents a forward reasoning step.

- *Program reflection* — Reflection is the mapping of the data structure representing a program into an executable version of that program. The need for reflection arises when we need to run a program constructed by another program: for example, executing the more specialized program resulting from a partial evaluation. Reflection allows the derived program to be executed directly, thereby avoiding the inefficiency and complexity of interpreting the program data structure.

  The results of partial evaluation (PE) represent one important application for reflection. In the context of logic programming, partial evaluation consists of deriving a sufficient condition G for a particular query E; that is, PE is a *specialization* of the logic program $\mathcal{P}$ that captures the computation leading from E to G. Through use of the resulting derived rule $E \Leftarrow G$, we avoid redoing the intervening computation.[4]

  Rules derived through **peval** could be assimilated with **assert**, as within the top-level predicate **peval_top**:

$$\textbf{peval\_top}\; E \quad \Leftarrow \quad \textbf{peval}\; E\; G,\; \textbf{asserta}\; (E \Leftarrow G).$$

[3] Prolog implementations typically offer both **asserta** and **assertz**: the former adds the clause to the beginning of the program, while the latter does so at the end. For purposes of general discussion, our use of **assert** encompasses both constructs.

[4] As a rudimentary example, from an appropriate logic program, the goal **grandparent** $x\ z$ might be partially evaluated to a subgoal (**son** $z\ y$, **daughter** $y\ x$). The resulting derived rule $E \Leftarrow G$ is then **grandparent** $x\ z \Leftarrow$ **son** $z\ y$, **daughter** $y\ x$.

In §3 we show how memoization and reflection can be achieved declaratively with our proposed **rule** construct.

assert (in combination with **retract**) also supports other programming mechanisms such as the mutation of global data, search control by hiding and revealing clauses, or self-modifying code in the general sense. Programs using **assert** in these ways are often stylistically questionable, and can frequently be reformulated without **assert** in a manner no more complex and no less efficient.

The principle drawback of **assert** is that it has no accessible declarative meaning. Consequently, work on the semantics of logic programs typically ignores it, and Prolog implementations behave inconsistently (see [10, p.22], for example).

For this and other reasons, λProlog does not include **assert**, although some of **assert**'s functionality is subsumed by another construct — embedded implication. However, as we shall illustrate, embedded implication is not powerful enough to support all of the above applications of **assert**. This led us to explore the possibility of making logically motivated extensions to λProlog that address some of these deficiencies, in particular the mechanisms of memoization and reflection.

2.2 Embedded Implication

It has been argued in the literature [12, 8, 1] that implication (with its intuitionistic meaning) can be used in place of **assert** in many applications, and can also be given a simple declarative semantics. The operational reading of embedded implication is that when solving the goal $D \Rightarrow G$, assume D while solving G. Thus, for example, without any program, the query

$$?-\ \ \mathbf{p}\,1 \Rightarrow \mathbf{p}\,x.$$

succeeds with the answer $x = 1$. The implication's assumption is in effect exactly while solving the consequent, and thus

$$?-\ \ (\mathbf{p}\,1 \Rightarrow \mathbf{p}\,x),\ \ \mathbf{p}\,y.$$

will fail, though

$$?-\ \ ((\mathbf{p}\,1,\ \ \mathbf{p}\,2) \Rightarrow \mathbf{p}\,x),\ \ x = 2.$$

succeeds after some backtracking.

Implication is of particular importance when we wish to make an assumption for a particular computation and then 'forget' it. Consider a reformulation of **peval_top**:

$$\mathbf{peval\_top}\ E\ K \quad \Leftarrow \quad \mathbf{peval}\ E\ G,\ \ (E \Leftarrow G) \Rightarrow K.$$

The revised **peval_top** takes two arguments: the goal E to be partially evaluated, and a second goal K representing the context for which the assumption $E \Leftarrow G$ will be valid. ('K' is for continuation.[5]) The rationale behind **peval_top** is that the client has some computation (captured in K) for which a particular specialization of the program ($E \Leftarrow G$) is applicable, yet he does not desire to make that optimization permanent (since, perhaps, it impairs performance in the general case).

At first, it might appear that the following definition would behave identically:

$$\mathbf{peval\_top}\; E\; K \;\;\Leftarrow\;\; \mathbf{peval}\; E\; G,\;\; \mathbf{asserta}\,(E \Leftarrow G),\;\; K,\;\; \mathbf{retract}\,(E \Leftarrow G).$$

However, the above is *not* equivalent to the preceding version: Suppose that the computation associated with K also makes extensions to the logic program. Should one of these assumptions unify with $E \Leftarrow G$, **retract** will leave $\mathcal{P}$ in an inconsistent state. Such potentially conflicting side-effects illustrate the difficulty in reasoning about programs that use **assert**.

In fact, **assert** and **retract** are not sufficient to encode implication in general: the problem with

$$(D \Rightarrow G) \quad \Leftarrow \quad \mathbf{asserta}\, D, G, \mathbf{retract}\, D.$$

(besides that of conflicting side-effects) is that should the interpreter initially successfully apply this clause but then later backtrack, the assumption D is unavailable for subsequent solutions of G (as backtracking over **retract** does not reinvoke **assert**).

2.3 Universal Quantification in Assumptions

In a Horn logic, all assumptions are closed: whatever apparently free variables occur in a clause D are in fact universally quantified. Moreover, in a Horn logic, the program cannot change during its execution. This is *not* the case for logics extended with embedded implication: assumptions added to $\mathcal{P}$ may therein contain logic variables that are *not* copied when that clause is used. Instead, since these logical variables may also occur in goals, the program may actually change (through variable instantiation) in the course of solving a goal. As a consequence, we must distinguish between the assumptions $\mathbf{p}\; x$ and $\forall x.\mathbf{p}\; x$. This is no great inconvenience: a clause occurring at

[5]The realization of *continuation-passing style* (CPS) [17] (as familiar from functional programming) within a logic programming language requires predicates be given an additional argument K (a goal). This goal is intended to represent the remainder of the computation, and rather than returning control upon success, clauses invoke this 'goal continuation.' In this way, accumulated assumptions are made available to extended computations. An example of CPS in λProlog may be found in [4].

the top-level in a program is still considered to be universally quantified over its free variables, but no such convention exists for embedded implications.

This points out a way in which implication is less powerful than **assert**: the assumption which is therein added to the program is not universally generalized. For example,

$$?- \ \mathbf{asserta}\,(\mathbf{p}\,x),\ \mathbf{p}\,1,\ \mathbf{p}\,2.$$

succeeds in Prolog, while

$$?- \ \mathbf{p}\,x \Rightarrow (\mathbf{p}\,1,\ \mathbf{p}\,2).$$

fails in λProlog: as one can see, there is no x such that $\mathbf{p}\,x$ implies both $\mathbf{p}\,1$ and $\mathbf{p}\,2$. Operationally, what happens is that resolving $\mathbf{p}\,1$ with the assumption $\mathbf{p}\,x$ instantiates x to 1, and the now instantiated assumption $\mathbf{p}\,1$ does not unify with the second subgoal $\mathbf{p}\,2$. On the other hand, the following clearly succeeds:

$$?- \ (\forall x.\mathbf{p}\,x) \Rightarrow (\mathbf{p}\,1,\ \mathbf{p}\,2).$$

It should be remarked here that this behavior of embedded implication is not a design mistake, but has its applications and, moreover, is entailed by the desire to make only logically sound extensions to basic Horn logic (for a further discussion see [12]).

This limitation of implication points out a problem in our implicational definition of **peval_top**: a clause derived by partial evaluation and then assumed can only be used with one substitution for its logical variables. Hence neither implication nor **assert** is the proper mechanism for this situation.

2.4 Embedded Implication and "assert"

We have seen that **assert** and **retract** are insufficient to program implication, in part due to the lack of proper scoping. Conversely, there are aspects of **assert** which are difficult to model with embedded implication [4]. Of these, the most problematic is the universal generalization of assumed clauses, because there is often no way to program around the problem short of completely reformulating the data representation.[6] It is universal generalization which is addressed by our proposed **rule** construct.

[6]This is essentially the solution advocated by Burt *et al.* [2].

3 Conservatively Extending Logic Programs

Let us now return to the **peval_top** example introduced in §2.2. Recall that the problem with

$$\textbf{peval\_top}\ E\ K \quad \Leftarrow \quad \textbf{peval}\ E\ G,\ (E \Leftarrow G) \Rightarrow K.$$

is that the free variables of $E \Leftarrow G$ (such as x, y, & z in **grandparent** $x\ z \Leftarrow$ **son** $z\ y$, **daughter** $y\ x$) are not universally generalized, thereby restricting the applicability of the assumption.

Operationally, what we would like to achieve is

1. Solve **peval** $E\ G$. If this succeeds with a substitution θ, θE and θG may contain logic variables. Let $\mathcal{Y}$ be those logic variables which do not occur free in any current assumption.
2. Assume $\forall\mathcal{Y}.(\theta E \Leftarrow \theta G)$ while solving θK.

Why is this a sound way of establishing θK? We need to make three crucial observations:

1. Since we quantify only over those variables which are not free in a current assumption, we know that $\forall\mathcal{Y}.$ **peval** $\theta E\ \ \theta G$ is a logical consequence of the program for **peval** (simply apply the principle of universal generalization).
2. We also know that, for any E and G, if **peval** $E\ G$, then $E \Leftarrow G$.
3. Hence, using a simple forward reasoning step, we conclude that the derived rule $\forall\mathcal{Y}.(\theta E \Leftarrow \theta G)$ is true and can thus be safely assumed before solving θK.

Trying to abstract from this particular example, we can see that we need two pieces of information in order to carry out the operations described above: the original goal to be solved — **peval** $E\ G$, and the general rule establishing the connection between this goal and the assumption we would like to make — $\forall E.\forall G.((E \Leftarrow G) \Leftarrow$ **peval** $E\ G)$. In order to properly scope assumptions, we also need to pass a goal continuation K as an argument. This line of reasoning is embodied within our new construct **rule**. For **peval**, the **rule** invocation is

$$\begin{aligned}\textbf{peval\_top}\ E\ K \ \Leftarrow\ \textbf{rule}\ & (\textbf{peval}\ E\ G)\\ & (\forall M.\forall G.\ \textbf{peval}\ E\ G \Rightarrow (E \Leftarrow G))\\ & K.\end{aligned}$$

3.1 The "rule" Construct

The general form of **rule** is

$$\textbf{rule}\ \ G\ \ (\forall\mathcal{X}.\,G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ \ K$$

for goals G, $G_{\mathcal{X}}$, K, and clause $D_{\mathcal{X}}$, where $\mathcal{X}$ is a (perhaps empty) subset of the variables free in $D_{\mathcal{X}}$ or $G_{\mathcal{X}}$. To simplify the discussion, we assume that the variables in $\mathcal{X}$ do not occur elsewhere.

The operational interpretation of

$$\mathcal{P} \Vdash \textbf{rule}\ \ G\ \ (\forall\mathcal{X}.\,G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ \ K$$

is as follows:

1. Find a minimal substitution $\sigma_{\mathcal{X}}$ such that $\mathsf{dom}(\sigma_{\mathcal{X}}) \subseteq \mathcal{X}$ and $\sigma_{\mathcal{X}} G_{\mathcal{X}} = G$. The existence of $\sigma_{\mathcal{X}}$ guarantees that the forward-chaining step is applicable. Should $\sigma_{\mathcal{X}}$ not exist, fail and issue a diagnostic message.
2. Solve $\mathcal{P} \Vdash G$. If this fails, fail. Otherwise, it succeeds with some substitution θ.
3. Let $\mathcal{Y} = \mathsf{free}(\theta G) - \mathsf{free}(\theta\mathcal{P})$.
4. Let $\mathcal{X}' = \mathcal{X} - \mathsf{dom}(\sigma_{\mathcal{X}})$.
5. Solve

$$\{\forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_{\mathcal{X}} D_{\mathcal{X}}\} \cup \theta\mathcal{P} \ \ \Vdash \ \ \theta K$$

The proper declarative reading for

$$\textbf{rule}\ \ G\ \ (\forall\mathcal{X}.\,G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ \ K$$

is simply

$$G,\ \ (\forall\mathcal{X}.\,G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}}) \Rightarrow K$$

which makes no mention of universal generalization whatsoever.

The above operational definition ensures that **rule** will succeed only if its corresponding declarative interpretation is (intuitionistically) true. This may be proved by induction on the definition of the $\Vdash$ relation; the proof may be found in [4].

The difference between the operational and declarative readings illustrates the savings provided by **rule**: Under the declarative interpretation, multiple instances of the same general goal G must be solved in order to establish instances of G's consequent D. Operationally, however, we need solve G only once, universally generalize, and then assume the universal closure of its consequent D.[7]

That it is the free variables of G, rather than those of D, which are universally generalized is essential to the correctness of this declarative reading: consider that

$$?-\ \ \mathbf{rule}\ \ \mathbf{true}\ \ (\mathbf{true}\ \Rightarrow\ \mathbf{p}\,x)\ \ (\mathbf{p}\,1, \mathbf{p}\,2).$$

fails. The following variation, on the other hand, should (and does) succeed:

$$?-\ \ \mathbf{rule}\ \ \mathbf{true}\ \ (\forall x.\, \mathbf{true} \Rightarrow \mathbf{p}\,x)\ \ (\mathbf{p}\,1, \mathbf{p}\,2).$$

Note that $\forall \mathcal{X}.\, G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}}$, though true, is never used in the back-chaining search of the interpreter; only the result of the forward step $\forall \mathcal{X}'\, \forall \mathcal{Y}.\, \sigma_{\mathcal{X}} \theta D_{\mathcal{X}}$ is assumed. This is essential as the clause one typically uses for this forward-chaining step is often hopelessly inefficient, or else quickly leads to non-termination if used in the reverse direction. We have already given as an example the step associated with **peval_top**: $(E \Leftarrow G) \Leftarrow \mathbf{peval}\ E\ G$.

3.2 Example: "lemma"

In applications such as memoization, the goal we would like to solve and the (generalized) assumption we would like to make coincide. Sterling & Shapiro [18, p.181] suggest **lemma** as a good way to achieve memoization in Prolog:

$$\mathbf{lemma}\, E \quad \Leftarrow \quad E,\ \mathbf{asserta}\,(E \Leftarrow !).$$

A version of **lemma** that restricts the scope of the assumption to a goal continuation K, but otherwise behaves identically, can easily be programmed with **rule**:

$$\mathbf{lemma}\, E\ K \quad \Leftarrow \quad \mathbf{rule}\ \ E\ \ (\forall E.\, (E \Leftarrow !) \Leftarrow E)\ \ K.$$

lemma *cannot* be effectively implemented without **rule** in λProlog, since there is no means to universally generalize over free variables.

[7] The declarative reading of **rule** is *not*, however, equivalent to its operational definition, as the declarative version may succeed where the operational fails. This is because **rule**'s assumption $\forall \mathcal{X}'\, \forall \mathcal{Y}.\, \sigma_{\mathcal{X}} \theta D_{\mathcal{X}}$ is typically less general than $\forall \mathcal{X}.\, \theta G_{\mathcal{X}} \Rightarrow \theta D_{\mathcal{X}}$, and thus K may follow from the latter, but not from the former. But this is, of course, the whole purpose of **rule**: to focus search by making use of a *selected consequence* $(\forall \mathcal{X}'\, \forall \mathcal{Y}.\, \sigma_{\mathcal{X}} \theta D_{\mathcal{X}})$ of the general assumption, which, by itself, may be too powerful to be computationally useful.

```
rtp    ⇐   rule   (infer R)
                  (∀R. clause R ⇐ infer R)
                  (R = false;  rtp).

infer R′   ⇐   clause P,  clause Q,  resolve P Q R,
               simpl R R′,  (R′ = false; keep? R′).

resolve   (P ; Q)    S          (P ; R)   ⇐   resolve Q S R.
resolve   (P ; Q)    S          (Q ; R)   ⇐   resolve P S R.
resolve   S          (P ; Q)    (P ; R)   ⇐   resolve Q S R.
resolve   S          (P ; Q)    (Q ; R)   ⇐   resolve P S R.
resolve   P          (not P)    false.
resolve   (not P)    P          false.

keep? R   ⇐   write R, write_string "Keep?  : ", read λG. G.
```

Figure 1: Rudimentary resolution theorem prover.

3.3 Example: Resolution

Consider **rtp**, a rudimentary resolution theorem prover, given in Figure 1. The predicate **clause** enumerates disjunctive expressions to be resolved, such as

> **clause** (**p** x y; **not** (**q** y x)).
> **clause** (**q** **a** z).

resolve blindly resolves its first two arguments, yielding a resolvent R, which is then simplified by **simpl** (whose clauses we omit). To illustrate,

> ?− **resolve** (**p** x y; **not** (**q** y x)) (**q** **a** z) R, **simpl** R R'.

instantiates $R' =$ **p** x **a**. To avoid infinitely rederiving the same clause, the user is queried by the predicate **keep**? to determine whether R' should be used or discarded.[8] **rtp** succeeds if it is able to derive a contradiction ($R' =$ **false**). **rtp** first invokes **infer**, which produces a resolvent of two clauses. If either $R' =$ **false** or **keep**? R' succeeds (*i.e.*, the user enters **true**), **infer** R succeeds. **rtp** then makes the forward step **infer** $R \Rightarrow$ **clause** R, and assumes the universally closure of **clause** R before the recursive call to **rtp**.

Due to the lack of assert, even this rather simple program could not have been expressed in λProlog.

[8]Within λProlog's input predicate **read** $\lambda x.Gx$, the variable x is bound to the entered term before execution of **read**'s body Gx.

$$\begin{array}{llll}
\textbf{typeof} & (\textbf{if}\ E\ F\ H) & A & \Leftarrow\ \textbf{typeof}\ E\ \textbf{bool},\ \textbf{typeof}\ F\ A,\ \textbf{typeof}\ H\ A. \\
\textbf{typeof} & (\textbf{lam}\ F) & (A \longrightarrow B) & \Leftarrow\ \forall x. \textbf{typeof}\ x\ A \Rightarrow \textbf{typeof}\ (Fx)\ B. \\
\textbf{typeof} & (\textbf{appl}\ F\ E) & B & \Leftarrow\ \textbf{typeof}\ F\ (A \longrightarrow B),\ \textbf{typeof}\ E\ A. \\
\textbf{typeof} & (\textbf{let}\ E\ F) & B & \Leftarrow\ \textbf{typeof}\ E\ A,\ \textbf{typeof}\ (FE)\ B. \\
\textbf{typeof} & (\textbf{fix}\ F) & A & \Leftarrow\ \forall x. \textbf{typeof}\ x\ A \Rightarrow \textbf{typeof}\ (Fx)\ A.
\end{array}$$

Figure 2: ML Type Inference

3.4 Example: ML-style Type Inference

As a further application of **rule**, consider the example of programming ML-style type inference [14], as implemented by Hannan & Miller [9].[9] Some of the more interesting rules for type inference are included within Figure 2. We are particularly interested in type inference over the ML **let** construct. We represent (**let** $x = F$ **in** Hx) within λProlog as **let** $F\ H$, where H is a λ-abstraction. (This reverses the order of arguments used within Hannan & Miller's representation.) Type inference for this construct can be captured by the following λProlog clause [9]:

$$\textbf{typeof}\ (\textbf{let}\ F\ H)\ B\ \Leftarrow\ \textbf{typeof}\ F\ A,\ \textbf{typeof}\ (HF)\ B.$$

The problem with the above formulation is that the type of F is computed once (to insure that it is indeed typable), and then thrown away. Instances of F are then re-typed at each occurrence of x within $\lambda x.Hx$.[10] This is necessary because the type of F, namely A, could be polymorphic — *i.e.*, contain variables such as the $C \longrightarrow C$ typing of the identity **lam** $(\lambda x.x)$. Without this re-computation, a polymorphic F can only be assigned one typing (*e.g.*, **int** $\longrightarrow$ **int**), since in the course of deriving that type, the logical variable C would be instantiated to **int** thus preventing it from matching, say, **bool** $\longrightarrow$ **bool** later.

Now consider another formulation

$$\begin{array}{ll}
\textbf{typeof}\ (\textbf{let}\ F\ H)\ B\ \Leftarrow & \forall x.\ \textbf{typeof}\ F\ A, \\
 & \quad (\forall A.\ \textbf{typeof}\ x\ A \Leftarrow \textbf{typeof}\ F\ A) \\
 & \quad\quad \Rightarrow \textbf{typeof}\ (H\ x)\ B.
\end{array}$$

As above, the initial **typeof** $F\ A$ insures that F has some typing (which is necessary in the case that the argument x does not occur in the body F). Now, however, rather than type HF, we type Hx using the additional rule

$$\forall A. \textbf{typeof}\ x\ A \Leftarrow \textbf{typeof}\ F\ A.$$

[9]We regret that for readers unfamiliar with λProlog, this example may be inscrutable. We include it, nevertheless, because of **rule**'s relevance to the problem. Space considerations preclude a fully treatment; see Hannan & Miller [9] instead.

[10](HF) is λProlog notation for the result of substituting F for occurrences of x in Hx.

This formulation simply separates the re-computation of F's type from that of typing H. Just as before, different occurrences of x may be given different types, and, just as before, the type of x (and hence the type of E) is re-computed from scratch at every occurrence.

Once the re-computation has been separated, it can be avoided entirely using the limited amount of forward reasoning and universal generalization afforded by **rule**:

$$\begin{aligned}\mathbf{typeof}\ (\mathbf{let}\ F\ H)\ B\ \Leftarrow\ \forall x.\mathbf{rule}\ &(\mathbf{typeof}\ F\ A)\\ &(\forall A.\,\mathbf{typeof}\ x\ A \Leftarrow \mathbf{typeof}\ F\ A)\\ &(\mathbf{typeof}\ (H\ x)\ B).\end{aligned}$$

This makes an assumption of the form $\forall\mathcal{Y}.\mathbf{typeof}\ x\ A_{\mathcal{Y}}$ while inferring the type of the body Hx. $\mathcal{Y}$ includes exactly those type variables in $A_{\mathcal{Y}}$ which are not free in any assumption, thus directly expressing the restriction on the ML inference rule for **let**. Here, we do not lose any solutions (and thus have completeness), since ML has the principal type property, and therefore all solutions to **typeof** $F\ C$ are instances of $\forall\mathcal{Y}.\mathbf{typeof}\ F\ A_{\mathcal{Y}}$.

4 Conclusion

The implementation of **rule** (and **rule_ebg**) as an extension of the eLP implementation of λProlog [6] is largely completed, and the examples in this paper have been run. The issue of efficient compilation of **rule** is tied to the very difficult general question of efficient compilation of λProlog (see [15]) and is beyond the scope of this paper.

However, one might ponder the more immediately tractable question whether **rule** could be added to Prolog as a declarative alternative to **assert** and **retract**. Assuming the necessary syntax to express variable binding (for the second argument of **rule**), **rule** could directly be incorporated into Prolog. Implementation would be substantially easier than in λProlog, since programs are always closed and thus we can simply quantify over all logic variables in the manner of **assert** without having to check any current assumptions. The implementation problems posed by **rule** in this context are thus very similar to those associated with **assert**, and variations on the techniques proposed by Lindholm & O'Keefe [10] are applicable.

There are a few examples closely related to the ones we have given here for which **rule** does not appear to be powerful enough. The problem is that it is not possible to translate the universal quantifiers of the logic programming language introduced during the universal generalization step into explicit quantifiers at the object level. Preliminary investigation indicates that it will be possible to add this through a generalization of **rule**, but we have not yet arrived at a declaratively and operationally satisfactory solution.

Interesting is also to consider the proof-theoretic view of the extension we propose here. Usually, the execution of a logic program produces what is known as a *cut-free* or *normal* proof of the query (actually, they belong to the even more restrictive class of *uniform proofs* [13]). Can we characterize the class of deductions which can be found by programs involving **rule**? Is there a way to extend **rule** so even more general deductions can be found without destroying the basic character of logic program execution as goal-directed search?

Finin, *et al.* consider a more complete integration of forward and backward chaining [7]. Their approach supports extended computations in both directions by allowing the programmer to write both forward and backward chaining Horn clauses. We do not see, however, a way to express the interplay between forward and backward reasoning required by **rule** within their language. It would be interesting to consider whether their approach to forward reasoning could be fruitfully combined with the higher-order constructs and scoping available in λProlog.

Acknowledgments. The existing implementation depends upon eLP, the implementation of λProlog developed by Conal Elliott and Frank Pfenning in the framework of the Ergo project at Carnegie Mellon University [6]. We thank Nevin Heintze, Spiro Michaylov, and Dale Miller for thoughtful discussions and comments. This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

References

[1] Anthony J. Bonner, L. Thorne McCarty, and Kumar Vadaparty. Expressing database queries with intuitionistic logic. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 831–850, Cambridge, Massachusetts, 1989. MIT Press.

[2] A.D. Burt, P.M. Hill, and J.W. Lloyd. Preliminary report on the logic programming language Gödel. Technical Report TR–90–02, Department of Computer Science, University of Bristol, March 1990.

[3] Widong Chen, Michael Kifer, and David S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989, Volume 2*, Cambridge, Massachusetts, 1989. MIT Press.

[4] Scott Dietzen. *A Language for Higher-Order Explanation-Based Learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. To appear.

[5] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 1991. To appear.

[6] Conal Elliott and Frank Pfenning. eLP: A Common Lisp implementation of λProlog in the Ergo Support System. Available via ftp over the Internet, October 1989. Send mail to elp-request@cs.cmu.edu on the Internet for further information.

[7] Tim Finin, Rich Fritzson, and Dave Matuszek. Adding forward chaining and truth maintenance to Prolog. In *IEEE Conference on Artificial Intelligence Applications*, March 1989. Also available as Paoli Research Center technical report PRC–LBS–8802.

[8] D. M. Gabbay and U. Reyle. N-prolog: an extension of Prolog with hypothetical implications I. *Journal of Logic Programming*, 1(4):319–355, 1985.

[9] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In John Lloyd, editor, *Proceedings of the Workshop on Meta-Programming in Logic Programming*, Bristol, England, June 1988. University of Bristol.

[10] Timothy G. Lindhold and Richard A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In Jean-Louis Lassez, editor, *Proceedings of the International Conference on Logic Programming*, pages 21–39. MIT Press, 1987.

[11] Sanjay Manchanda and David Warren. A logic-based language for database updates. In *Foundations of Deductive Dattabases and Logic Programming*, 1987.

[12] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):57–77, January 1989.

[13] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1989.

[14] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[15] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for lambda Prolog. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 1180–1198. MIT Press, October 1989.

[16] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[17] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, New York, 1972. ACM.

[18] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.

[19] D. S. Warren. Database updates in pure prolog. In Jean-Louis Lassez, editor, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*. North-Holland, 1984.