Equality for Prolog

William A. Kornfeld
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139
U.S.A.
telephone: (617) 492 6172
arpanet: BAK®MIT-MC

Abstract

The language Prolog has been extended by allowing the inclusion of assertions about equality. When a unification of two terms that do not unify syntactically is attempted, an equality theorem may be used to prove the two terms equal. If it is possible to prove that the two terms are equal the unification succeeds with the variable bindings introduced by the equality proof. It is shown that this mechanism significantly improves the power of Prolog. Sophisticated data abstraction with all the advantages of object-oriented programming is available. Techniques for passing partially instantiated data are described that extends the "multi-use" capabilities of the language, improve the efficiency of some programs, and allow the implementation of arithmetic relations that are both general and efficient. The modifications to standard Prolog are simple and straightforward and in addition the computational overhead for the extra linguistic power is not significant. Equality theorems will probably play an important role in future logic programming systems.

1. Introduction

Prolog is a computer language based on Horn clause resolution. The basic resolution procedure, similarly with Prolog, does not allow statements about equality. We cannot, for example, express that 6 = *successor(S)* and then be able to unify *P(6)* with P(succesor(5)). We have adapted a Lisp-based Prolog system <Kahn 82> so that it is possible to specify theorems about equality. The unification algorithm has been modified so that if the unification of two terms fails, an assertion is looked for that will prove the two terms are equal. As with any application of the unification procedure, the proof of equality of the two terms may cause the binding of variables in either or both of the terms.

Our Prolog-with-Equality is a natural extension to standard Prolog. We have found several applications for it that extend the power, expressibility, and generality of Prolog. These principally fall into two categories—extensible datatypes and greater opportunities to pass partially specified data objects. The former turns out, surprisingly, to augment Prolog with all the flexibility of "object-oriented" languages typified by Smalltalk <Ingalls78>, and potentially much more. The second major application area is a greatly improved facility for passing partially instantiated data objects as an alternative to a backtracking-based enumeration of possible bindings. These turn out to be quite easily implemented in Prolog-with-Equality. Moreover, the effect on efficiency in interpreted Prolog is minimal and there is every reason to suppose that most computational overhead that is introduced can be eliminated in compilation.

The notion of equality theorems is very similar to the paramodulation rule found in the theorem proving literature <Chang,Lee73>.

2. Example—Rational Number Datatype

We will create a rational number datatype as an example of the use of equality theorems. Rational numbers will be represented by a term made up of the functor rat with two arguments, the numerator and denominator respectively. There is a distinguished predicate symbol equals which is used to specify the equality axioms.

We add the following assertion to our database:[1]

```
(equals (rat <--numl <-denoml) (rat <-num2 «-denom2)) :-
            (times «-numl <-denon2 <-intermediate)
            (times <-num2 «-denoml <-intermediate)
```

This expresses the usual cross-multiplication rule for deciding if two rationals are equal to one another.

Our augmented unification rule works as follows: When the interpreter attempts to unify two objects Φ and Φ using standard unification and fails, it will establish the goal (EQUALS Φ). If this demonstration succeeds, the unification succeeds with the new variable bindings. If this fails, it will attempt to prove (EQUALS Φ). An attempt to unify (rat 2 3) with (rat «-X 3) will succeed because standard unification succeeds. An attempt to unify

'Our Prolog syntax is not standard and should be briefly explained. A term is represented as a Lisp list whose first element is the functor and the remaining elements are the arguments. A literal prefaced by the symbol "«-" is a variable.

(rat 2 3) with (rat «X 6) will cause the goal

```
(EQUALS (RAT 2 3) (RAT «X 6))
```

to be posed by the system. This will succeed with the variable «-- bound to 4. The original computation continues with this binding and it is placed in the trail for undoing on backtracking.

We have not yet told the system how to match rationals with any other kind of object. We wish to express a method to decide if a rat is the same as an integer. The following assertion is added:

```
(equals (rat ←num ←denom) ←n)   :-
      (type ←n integer)
      (if (and (variable ←num)
               (variable ←denom))
          (and (= ←num ←n) (= ←denom 1))
          (times ←denom ←n ←num))
```

If both the numerator and the denominator are uninstantiated it succeeds by unifying the numerator with the integer, otherwise it proves the multiplicative relationship that must hold for the equality to hold.

The following little program will now succeed 3 times:

```
(member (rat 4 «-x)
        [2 3 (cons <-y <-z) (rat <-r <-w) (rat 2 7)])

<-x = 2
+r = 4, «-x = «-w
<-x = 14
```

We will also wish, to specify the behavior of this new datatype for several additional relations. The > relation for rationals is implemented with the following assertion:

```
(> (rat ←n1 ←d1) (rat ←n2 ←d2)) :-
     (times ←n1 ←d2 ←x)
     (times ←d1 ←n2 ←y)
     (> ←x ←y)
```

This rule could of course work in a Prolog without equality theorems when comparing two terms beginning with the functor rat. Now, however, if we try to prove the goal (> (rat 3 2) 1), the attempt to unify this goal with the head of this assertion will cause an attempt to prove that 1 equals (rat «-n2 «-d2) which will succeed with the appropriate bindings (n2=l, d2=l), allowing the > goal to succeed. In other languages this might be called "automatic type conversion."

3. Extensible Datatypes and Generic Operations

Equality theorems allow us to gain the modularity advantages of generic operations and class structuring as are found in object-oriented languages like Smalltalk. The important contributions of this class of languages are two:

1. One can specify methods for computing facts about whole classes of objects and then have those methods automatically inherited by subclasses.

2. A given operation will have different effects on different data, in a manner determined by the datum's class. This is decided at run time rather than compile time.

The paradigm of logic programming is very different from the paradigm of message passing systems. We do not have "objects" which receive "messages." The role of objects are played by terms; the role of messages by relations. The concept of "class" has no formal analog in Prolog-with-Equality. The effect of class structuring is accomplished by the use of equals assertions. A subclass relationship is indicated by a single equals assertion containing terms for the sub- and super-class. The pattern of variables between the two terms and the body of the assertion express the relationship between the two classes.

As an example, we might have the class regular-polygon. All regular polygons have four attributes, an X and Y location, a side length, and the number of sides. In Prolog we might want to specify a method for computing the areas of regular polygons. This would be done as follows:

```
(area (regular-polygon (←x ←y ←side-number ←length))
    ←area)
  (quotient 180 ←side-number ←theta)
  (cotangent ←theta ←cotangent)
  (times ←length ←length ←length-squared)
  (times ←length-squared ←side-number ←prod1)
  (times ←prod1 ←cotangent ←prod2)
  (quotient ←prod2 4 ←area)
```

We could then have a functor equilateral-triangle that is defined to be equal to a regular-polygon with three sides:

```
(equals (equilateral-triangle ←x ←y ←length)
        (regular-polygon ←x ←y 3 ←length)) :-.
```

Then if we prove the goal

```
(area (equilateral-triangle «- «- 100) «-ans)
```

the attempt to unify it with the head of the area assertion will result in an attempt to show the term with functor equilateral-triangle is equal to a term with functor regular-polygon. This will succeed with <-length bound to 3 and the goal will succeed with «-ans bound to 150. Of course, since this is Prolog, we could ask the question: "How long does the side of an equilateral triangle have to be to have an area of 150?" This would be:

```
(area (equilateral-triangle «- «- <--length) 150)
```

This goal will succeed with «-length bound to 100.

We could also have area assertions for other kinds of objects such as ellipses:

```
(area (ellipse ←x ←y ←minor-axis ←major-axis) ←area) :
```

This will not interfere in any way with goals asking for the areas of triangles because there is no theorem that allows us to show Triangles and Ellipses are equal.

We could define a circle by saying:

```
(equals (circle ←x ←y ←radius)
        (ellipse ←x ←y ←diameter ←diameter)) :-
    (times 2 ←radius ←diameter)
```

that will then allow the use of theorems about ellipses for answering questions about circles.

There is one sense in which we have more flexibility in Prolog-with-Equality than can be achieved with object-oriented languages. As with other aspects of Prolog, we are able to leave many aspects of our computation undetermined and still carry out the computation. Within object-oriented languages an object's class is known at object instantiation time. This is not true for terms in Prolog-with-Equality. Suppose we have the goal:

```
(and (= ←obj (regular-polygon (← ← ←side-number ←length)))
     (= ←side-number 3)
     (predicate ←obj))
```

The variable obj is unified with a regular-polygon of an indeterminate number of sides. Side-number is later unified with 3 and predicate is called. If predicate has assertions that will allow proofs about triangles then we will succeed because obj will unify with triangles.

A fanciful example involves the term closed-figure with two argument, a perimeter and an area. We can state the following equality assertion:

```
(equals (closed-figure ←perimeter ←area)
        (circle ← ← ←radius))            :-
    (times 6.2831853 ←radius ←perimeter)
    (times ←radius ←radius ←squared-radius)
    (times 3.14159265 ←radius-squared ←area)
```

expressing the fact that if the perimeter of a closed figure is 27r and the area is nr^2 for some r, then it is a circle. A closed-figure described in this way, if instantiated to be a circle, can be used in any proof that was defined using the circle functor.

The code above is most unlikely to appear as part of a practical program. When things are left so undetermined, simple programs can take very long because there may be very esoteric ways of doing proofs. It is important, however, to see that such generality is available. Moreover, the uses of equality theorems that model the class-structuring idea are very efficiently implemented. We discuss these issues briefly in section 7.

4. Partially Specified Objects

One of the desirable features of Prolog is its ability to deal with partially instantiated data objects. One way to look at this is that the partially instantiated data object stands for a non-singleton subset of the Herbrand Universe corresponding to our Prolog program. If we were not able to pass such objects freely in our programs the control structure of Prolog (or any logic programming language remotely like it) would require us to successively generate bindings of the variable to ground objects for consideration by the remainder of the proof. This is the power that unification gets us and also the principle difference between Prolog and the language Micro-Planner <Sussman, et. al. 70> which preceded it but was otherwise similar. We find that our inclusion of equality proofs in Prolog extends the range of possibility for passing partially instantiated data. In some cases this can significantly improve the efficiency of programs by replacing an otherwise expensive enumeration of ground terms with backtracking. In other cases it makes possible a program that would not be possible otherwise because there is no convenient way to enumerate the space.

We define a binary functor called Ω which will represent an arbitrary partially specified object. The first argument to an Ω term is a variable that is uninstantiated when the term is introduced. The second argument is a relation that must hold for all terms in the subset of the Herbrand Universe that this Ω term represents; it is expressed in terms of the uninstantiated variable. For example, if we execute the goal (> <x 3) where the variable x is at this point uninstantiated, we will succeed with the binding for x: (Ω «-num (> «-num 3)). This represents (and will unify with) all numbers that are greater than three. When this is unified with a number (e.g. 5) the variable num is unified with 5 and the unification succeeds of the relation succeeds (which it does because three is greater than five).

The equals relation for Ω terms is defined as follows:

```
(equals (Ω ←obj ←relation) ←thing) :-
   (instantiate ←thing ←value)
   (= ←obj ←value)
   ←relation

(equals (Ω ←obj1 ←relation1) (Ω ←obj2 ←relation2)) :-
      (and  (not-instantiated ←obj1)
            (not-instantiated ←obj2)
            (conjoin-relations ←obj1 ←relation1
                               ←obj2 ←relation2
                               ←obj ←relation)
            (= ←obj1 ←obj2)
            (= ←obj2 (Ω ←obj ←relation)))
```

The predicate instantiate succeeds if its argument is instantiated, i.e. that it is neither a variable nor an uninstantiated Ω term; when it succeeds the second argument is bound to the instantiated value. The first assertion declares that an Ω term is equal to something if the item the Ω functor represents (obj) is equal to the thing's instantiated value and the constraining relation is true. The second assertion applies if two ft terms are unified and neither has yet been instantiated. Both ft terms are made equal to one another and to another ft term whose relation is the conjunct of the two previous relations. This new term represents the intersection of the space of possible ground terms of the two ft terms that were unified.

The code for > is as follows:[2]

```
(> ←m ←n)  :-
   (cases ((and (instantiate ←m ←m1)
                (instantiate ←n ←n1))
           (lisp-> ←m1 ←n1))
          ((instantiate ←m ←m1)
           (= ←n (Ω ←num (> ←m1 ←num))))
          ((instantiate ←n ←n1)
           (= ←m (Ω ←num1 (> ←num1 ←n1))))
          ((= ←n (Ω ←num (> ←m ←num)))
           (= ←m (Ω ←num1 (> ←num1 ←n))))))
```

If we had the Prolog goal:

```
(and (> ←x 3) (member ←x [2 4 6 ←y]))
```

we would succeed 3 times with the bindings 4, 6, and (Ω ←num1 (> ←num1 3)), the last which is also bound to y.

The goal

```
(and (> ←x ←y) (= ←x 3) (= ←y 2))
```

succeeds, while the goal:

```
(and (> ←x ←y) (= ←x 3) (= ←y 4))
```

does not.

The basic arithmetic functions are defined in a similar way, and have the property of not failing if any or all of their arguments are not instantiated. The definition of the times predicate is:

```
(times ←arg1 ←arg2 ←result) :-
   (cases
     ((instantiate ←arg1 ←ans1)
      (cases ((instantiate ←arg2 ←ans2)
              (lisp-value ←result (* ←ans1 ←ans2)))
             ((instantiate ←result ←ans)
              (lisp-value ←arg2 (// ←ans ←ans1)))
             ((= ←result
                 (Ω ←ans (times ←ans1 ←arg2 ←ans)))
              (= ←arg2
                 (Ω ←ans2 (times ←ans1 ←ans2 ←result))))))

     ((instantiate ←arg2 ←ans2)
      (cases ((instantiate ←result ←ans)
              (lisp-value ←arg1 (// ←ans ←ans2)))
             ((= ←result
                 (Ω ←ans (times ←arg1 ←ans2 ←ans)))
              (= ←arg1
                 (Ω ←ans1 (times ←ans1 ←ans2 ←result))))))

     ((instantiate ←result ←ans)
      (= ←arg1 (Ω ←ans1 (times ←ans1 ←arg2 ←ans)))
      (= ←arg2 (Ω ←ans2 (times ←arg1 ←ans2 ←ans))))
     ((= ←arg1 (Ω ←ans1 (times ←ans1 ←arg2 ←result)))
      (= ←arg2 (Ω ←ans2 (times ←arg1 ←ans2 ←result)))
      (= ←result (Ω ←ans (times ←arg1 ←arg2 ←ans)))))
```

The definition of times determines which arguments are instantiated and takes an appropriate action. If two or three are instantiated it executes an evaluable predicate corresponding to a Lisp function call. Otherwise it creates ft terms that express the arithmetic constraints on the value of the variable. Other arithmetic functions are defined analogously. Prolog programs written using these relations have the property that the relations can be in any order and the program will find the answer without any backtracking. For example, the goal

```
(and (times ←x ←y ←z)
     (plus ←a 3 ←x)
     (plus 1 ←a 2)
     (= ←z 12))
```

will succeed with correct bindings for all the variables. This is accomplished with sequential evaluation of the predicates and no backtracking. Arithmetic expressed in this way can solve systems of equations that are not simultaneous.[3] It is a very efficient way of doing arithmetic, and will work for floating point as well as integer computations.[4]

Arithmetic expressions are an example where it is impossible in a practical sense to generate each of the possible bindings as ground terms. However there are other situations where we may wish to use ft terms as an alternative to enumerating the bindings.

[2]The preponderance of the impure instantiate relation is only necessary when data if to be passed to evaluablc predicates (such as lisp->). As <Warren 77> points out, most of the work in building a Prolog is in writing the "evaluable predicates". However, our > relation is a true Prolog predicate in that it is not required for its arguments to be instantiated when called. Our goal is to be able to bury such impure evaluable predicates inside logically sound relations.

[3]This is equivalent in power to the constraint network formalisms <Borning 79, Sussman, Steele 78>.

[4]IC-Prolog <Clark 82> provides arithmetic on integers by building the integers using successor. This is very general, but not efficient for practical problems.

Consider the following goal:

```
(and (member «x «-very-long-list)
     (pred «-x))
```

where on entry very-long-list is instantiated to a very long list and x is uninstantiated. Suppose that pred is true of only a few of those elements. In this case, using the standard Prolog definition of member, the relation would become a generator of succesive elements from the list. An alternative way of writing member would be that, if x is not instantiated upon entry into the member relation, the relation would instantiate x to:

```
(Ω ←element (member ←element ←very-long-list))
```

This would effectively save the constraint that x be a member of the list until there was some item under consideration. Then, and only then, would the constraint be checked.

It is far from clear how one would characterize the conditions under which ft terms will be preferable to using a predicate as a generator. Yet this does seem an important tool. It is distinct from, yet compatible with, extralogical control annotation as in IC-Prolog < Clark 82 >.

5. Functional Notation

Many people are uncomfortable with the flat relational style of Prolog as opposed to the functional notation Lisp and most other languages. Lisp syntax allows the composition of functions without the need to introduce temporary variables to glue successive relations together as in the above example for computing the area of a polygon. We can straightforwardly augment Prolog with functional notation by using equality theorems:

```
(equals (%times ←x ←y) ←z) :=
   (times ←x ←y ←z)

(equals (%plus ←x ←y) ←z) :-
   (plus ←x ←y ←z)

(equals (%difference ←x ←y) ←z) :-
   (difference ←x ←y ←z)

(equals (%quotient ←x ←y) ←z) :-
   (quotient ←x ←y ←z)
```

Notice that we distinguish the function %plus from the relation plus. It makes no sense in (logic or) Prolog to unify a relation and a function. Using these new functors we can define a temperature converter relation that expresses the relationship between the Fahrenheit and centigrade scales. The relation is a one-liner:

```
(temp-converter of (5 times (% difference <--f 32) (rat 5 0)))
```

This is like the "executable pattern" of <Nakashima 82>.

6. Equality Assertion Invocation

Prolog constructs can be understood in either a declarative or procedural sense. Thus far we have not considered the precise procedural semantics of Prolog-with-Equality. Certain care must be exercised in defining the procedural semantics to avoid infinite loops or otherwise unacceptably wasteful searches. Our solution to this yields a system which is far from complete in the logical sense, yet is sufficient for the classes of examples in this paper. It has a further desirable property that allows the compiler to precompute certain information that will allow most failing unifications to fail in *constant time.* Such a property is critical if this mechanism is to find its was into practical programs.

The first restriction we put on unification via equality assertions is that the assertions can only be used *one way.* Suppose we attempt to match (foo 1) with (bar 2). This will cause the unifier to generate a goal of the form:

```
(equals (foo 1) (bar 1))
```

Now suppose we had an equality theorem whose head was: (equals (bar «-n) «-s). An attempt to unify the goal with this assertion head would cause a recursive goal of (equals (foo 1) (bar 1)). This will lead to an infinite computation. To avoid this problem we require that in matching the *first* term of the equals goal with its counterpart in the head of the assertion the functors must be the same. The unification of the second argument may recursively create new equality assertions. This allows for chains of equality matches (but not cycles).

Unification in Prolog-with-Equality, as in standard Prolog, is deterministic—it can only succeed once. For normal Prolog there is no problem because there can be only one most general unifier. However, in consideration of of models other than the standard Herbrand model (see, for example, a survey of equational theories <Siekmann, Saabo 82>) it is possible for there to be no unique most general unifier. To properly handle such cases would require introducing a backtracking point in the unification itself. For the class of applications I have looked at so far there does not seem to be a compelling reason for doing so, yet this is possible route to be taken in the future. One way this could be understood within the class hierarchy paradigm is that there may be more than one most general unifier because there is more than one way to view the first as an instance of the second.

7. Efficiency Considerations

It is critically important from the efficiency point of view that one of the functors must be explicitly present as the first functor of an equals assertion. Thus equals assertions can be grouped in much the same way that assertions with the same first predicate are grouped in the implementation. If one tries to match two terms, neither of which has equals assertions for it, the match can fail quickly. In the event that one or both have assertions associated with them a series of equals derivations may ensue that is potentially costly. A compiler can do much to eliminate this whenever the terms are not relatable by figuring the chains of functors statically from the code. If neither functor can be reached by a chain of equals assertions from the other the two can definitely not be unified. By using hash tables to store this information most failures can be discovered very quickly. In particular, the use of equality to model object-oriented languages requires no backtracking and a small, constant, overhead to determine failure.

8. Conclusions

My experience with Prolog-with-Equality to date has been very encouraging. The modifications necessary to the interpreter to make equality theorems possible was done in one evening of programming. The basic paradigms explored (extended datatyping, partially instantiated Ω terms) appear quite natural and general. Moreover the added features do not impinge on the efficiency of basic Prolog. I am hopeful that new paradigms for using equality will emerge as more experience is gained.

I am currently developing a system in Prolog-with-Equality for composition of musical examples of species counterpoint. This area of music theory has been studied extensively over several centuries and large, precise rules systems exists for it. The two paradigms are quite important for developing this system. The partially instantiated term notion is used to represent classes of pitches (when the fully instantiated note must be one of those pitches). The unification process for two partially instantiated pitches becomes a set intersect. This way expensive backtracking is avoided.

9. Bibliography

Attardi G., M. Simi, Consistency and Completeness of OMEGA, a Logic for Knowledge Representation, Seventh International Joint Conference on Artificial Intellgence, Vancouver, August 1981.

Borning, A., Thinglab—A Constraint-Oriented Simulation Laboratory, Xerox PARC report SSL-79-3, July 1979.

Chang, C, R. Lee, Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

Clark, K., IC-Prolog Language Features, in Clark & Tarnlund, Logic Programming, Academic Press, 1982.

Ingalls The Smalltalk-76 Programming System: Design and Implementation, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson Arizona, January 1978.

Kahn, K., Unique Features of LM-Prolog, unpublished manuscript.

Nakashima, H. Prolog K/R language feature, First International Logic Programming Conference, Marseille, September, 1982.

Siekmann, J., P. Szab6, Universal Unification and a Classification of Equational Theories, 6th Conference on Automated Deduction, New York, June 1982.

Sussman, G., T. Winograd, E. Charniak, Micro-Planner Reference Manual, MIT Artificial Intelligence Laboratory memo 203, 1970.

Sussman, G., G. Steele, Constraints MIT Artificial Intelligence Laboratory memo 502, November, 1978.

Warren, D., Implementing Prolog—Compiling Predicate Logic Programs, University of Edinburgh Department of Artificial Intelligence Report No. 39.