Chapter 32

Programming in Prolog

32.1 Example: Academic Family Trees

A Prolog program consists of a set of facts and a collection of rules. Given a program, a user can ask the Prolog evaluator whether a particular fact is true or not. The query may be true atomically (because it’s in the set of facts); if it’s not, the evaluator needs to apply the rules to determine truth (and, if no collection of rules does the trick, then the query is judged false). With this little, Prolog accomplishes a lot!

Let’s plunge into a Prolog program. It’s inevitable, in the course of studying Prolog, to encounter a genealogy example. We’ll look at the genealogy of a particular, remarkable mathematician, where parenthood is determined by PhD advisors. We’ll first list a few facts:

```
advisor(barwise,feferman).
advisor(feferman,tarski).
advisor(tarski,lesniewski).
advisor(lesniewski,twardowski).
advisor(twardowski,brentano).
advisor(brentano,clemens).
```

All facts are described by name of a relation (here, `advisor`) followed by a tuple of values in the relation. In this case, we will assume that the person in the first position was advised by the person in the second position. Prolog does not ask us to declare relations formally before providing their contents; nor does it provide a means (beyond comments) of formally describing the relationship. Therefore, Prolog programmers must be careful to keep track of how to interpret each relation.

Facts relate constants. In the example above, `barwise` and `feferman` are both constants. A constant is, in fact, just a relation of arity zero, but understanding this isn’t necessary for the rest of this material.

In this example, all relations (including constants) are named by lower-case initial letters. This is not a coincidence; Prolog requires this. Accidentally writing `Barwise` instead of `barwise` would change the meaning of code quite a bit, because an initial capital denotes a variable, which we will study soon. Just keep in mind that the case of the initial letter matters.

Given this input, we can ask Prolog to confirm a few basic facts:
This asks Prolog at its prompt whether the constants `barwise` and `feferman` are in the `advisor` relationship. Prolog responds affirmatively. In contrast,

```
:- advisor(f reverman,barwise).
no
```

Prolog responds negatively because it has no such fact in the database specified by the program.

So far, Prolog has done nothing interesting at all. But with rules, we can begin to explore this universe.

A standard genealogical question is whether one person is another’s ancestor. Let’s build up this rule one step at a time:

```
ancestor(x,y):-advisor(x,y).
```

This says that `y` is `x`’s (academic) ancestor if `y` is `x`’s advisor. But this isn’t very interesting either: it just sets up `ancestor` to be an alias for `advisor`. Just to be sure, however, let’s make sure Prolog recognizes it as such:

```
:- ancestor(barwise,feferman).
no
```

What?!? Oh, that’s right: `x` and `y` are constants, not variables. This means Prolog currently knows how to relate only the constants `x` and `y`, not the constants `barwise` and `feferman`, or indeed any other constants. This isn’t what we mean at all! What we should have written is

```
ancestor(X,Y):-advisor(X,Y).
```

Now, sure enough,

```
:- ancestor(barwise,feferman).
yes
```

So far, so good. There is another way for one person to be another’s academic ancestor: by transitivity. We can describe this verbally, but it’s at least as concise, and just as readable, to do so in Prolog:

```
ancestor(X,Y):-
  advisor(X,Z),
  ancestor(Z,Y).
```

Read the ‘,” as “and”, while the multiple definitions for `ancestor` combine with “or” (i.e., each represents a valid way to be an ancestor, so to be in the `ancestor` relation it’s enough to satisfy one rule or the other). All Prolog rules are written in this “and of or’s” form (Conjunctive Normal Form). Notice the use of `Z` twice on the right-hand side. This is intentional: this is what captures the fact that the `same` person must be both the immediate advisor and herself a descendant.

Armed with this extended rule, we can ask more interesting queries of Prolog:

---

1(Careful with those capital letters!)
32.1. EXAMPLE: ACADEMIC FAMILY TREES

 outgoing advisor(barwise,tarski).
 yes

 so we can confirm that Barwise was advised by a legend. But it’s always a good idea to write tests that
 ensure we didn’t write a faulty rule that made the relation too large:

 outgoing advisor(tarski,barwise).
 no

 By the way, here’s an easy kind of mistake to make in Prolog: suppose you write

 advisor(tawrdowski,brentano).

 instead of

 advisor(twardowski,brentano).

 Then you get

 outgoing advisor(barwise,clemens).
 no

 Prolog doesn’t have any way of knowing about slight misspellings of Polish names. It accepts your facts
 as truth; garbage in, garbage out. This is another important pitfall (along with capitalization and making a
 relation too large) to keep in mind.

 Now let’s expand the relation with a few more facts. Franz Brentano actually had two advisors, of whom
 we’ve given credit to only one above. So we should add the fact

 advisor(brentano,trendelenburg).

 We can now ask Prolog

 outgoing advisor(barwise,clemens).
 yes
 outgoing ancestor(barwise,trendelenburg).
 yes

 and it shows that the relationships Prolog tracks really are relations, not functions: the mapping truly can
 be one-to-many. (We could already have guessed this from the rule for ancestor, where we provided
 multiple ways of determining whether or not a pair of constants was in that relation.)

 Now let’s add some more rules. The simplest is to ask questions in the other direction:

 descendant(X,Y):-ancestor(Y,X).

 As of now, each person has only one immediate descendant. But most of these people produced many
 students. Tarski, one of the great logicians and mathematicians, not only generated a rich corpus of material,
 but also trained a string of remarkable students. We already know of one, Feferman. Let’s add a few more:
advisee(tarski,montague).
advice(tarski,mostowski).
advice(tarski,robinson).

Now, clearly, there are more ways of being one’s descendant. There are two ways of fixing the descendant relation. One, which involves a lot of work, is to add more facts and rules. A much easier fix is to note that every advisee relationship subscribes a corresponding advisor relationship:

advisor(X,Y):-advisee(Y,X).

And sure enough,

:- descendant(clemens,montague).
yes
:- descendant(trendelenburg,montague).
yes
:- descendant(feferman,montague).
no
:- descendant(barwise,montague).
no

We haven’t at all explained how Prolog evaluates, and that’s largely because it seems so very intuitive (though once we have multiple clauses, as in descendant, it may be a little less than obvious). But then, we also haven’t seen Prolog do anything truly superlative. Let’s explore some more.

Let’s first assume we’ve removed Trendelenburg from the database, so Brentano has only one advisor. (We can do this in a source file by using C-style comments, delimiting text in /* and */.) Then let’s ask Prolog the following query:

:- ancestor(barwise,X).

What does this mean? We know all the parts: advisor is a relation we’ve defined (by both facts and rules); barwise is a constant; and X is a variable. We should interpret this as a query, asking Prolog whether there is a value for X that would satisfy (make true) this query. In fact, we know there is (clemens). But Prolog’s response is worth studying. This particular Prolog system\footnote{Trinc-Prolog R3. In many textual Prolog systems, it’s conventional to print a caret to indicate that another solution is available. The user types a semi-colon to ask Prolog to present it.} prints

SOLUTION:
X=feferman

So not only did Prolog establish that the query was valid, it also found a solution for X! Now this isn’t the solution we expected above, but if you think about it for a moment, it’s clear that the query has multiple solutions, and Prolog has picked one of them. In fact, at the bottom of the window (in this interface), Prolog says Press cancel to stop, or continue for more solutions. Clicking on the Continue button provides one more solution, then another, then another, and so on until there are no more, so the final output is
32.1. EXAMPLE: ACADEMIC FAMILY TREES

SOLUTION:
   X=feferman
SOLUTION:
   X=tarski
SOLUTION:
   X=lesniewski
SOLUTION:
   X=twardowski
SOLUTION:
   X=brentano
SOLUTION:
   X=clemens
no

Wow! Prolog actually “filled in the blank”. In fact, if we put Trendelenburg back into the picture, Prolog prints one more solution:

SOLUTION:
   X=trendelenburg

We can ask a similar query with the variable in the first position instead:

:- ancestor(X,clemens).
SOLUTION:
   X=brentano
SOLUTION:
   X=barwise
SOLUTION:
   X=feferman
SOLUTION:
   X=tarski
SOLUTION:
   X=lesniewski
SOLUTION:
   X=twardowski

This shows that Prolog isn’t just working as a functional program might, where the last position in the relation is like a “return” location. Prolog really doesn’t discriminate between different positions where you might put variables.

Maybe this isn’t so surprising. After all, Prolog is merely listing the same chain of relationship that we entered as facts at the top of the program. Actually, this isn’t quite true: it had to apply the transitive rule of ancestor to find all the solutions (and these are indeed all of them). But perhaps a more impressive test would be to ask a query that runs counter to the facts we entered. For this, we should employ the advisee relation.
:- descendant(tarski,X).
SOLUTION:
    X=feferman
SOLUTION:
    X=montague
SOLUTION:
    X=mostowski
SOLUTION:
    X=robinson
SOLUTION:
    X=barwise

Sure enough, Prolog produces the entire part of Alfred Tarski’s family tree that we’ve taught it. Notice that
to get to Barwise it had to recur through Feferman using advisor, and to get to Robinson it had to employ advisee.

This is pretty impressive already. But we can take this a step further. Why stop with only one variable? We
could in fact ask

:- ancestor(X,Y).

In response, Prolog will actually compute all pairs in the ancestor relation and present them sequentially:

SOLUTION:
    X=barwise
    Y=feferman
SOLUTION:
    X=feferman
    Y=tarski
SOLUTION:
    X=tarski
    Y=lesniewski
SOLUTION:
    X=lesniewski
    Y=twardowski

and so on and on.

Now let’s explore another relation: academic siblings. We can define a sibling pretty easily: they must have a common advisor.

sibling(X,Y):-
    advisor(X,Z),
    advisor(Y,Z).

We can either ask Prolog to confirm relationships (and just to be sure, try them both ways):
32.2. INTERMISSION

`: sibling(robinson, montague).
yes
`: sibling(montague, robinson).
yes

or to generate their instances:

`: sibling(robinson, X).
SOLUTION:
    X=feferman
SOLUTION:
    X=montague
SOLUTION:
    X=mostowski
SOLUTION:
    X=robinson

How’s that? When Robinson comes along to find out who her academic siblings are, she finds... herself!

It’s not very surprising that we got this output. What we meant to say was that different people, X and Y, are academic siblings if... and so on. While we may have mentally made a note that we expect X and Y to be different people, we didn’t tell Prolog that. And indeed, because Prolog programs can be “run backwards”, it’s dangerous to not encode such assumptions. Making this assumption explicit is quite easy.

```
sibling(X, Y):-
    advisor(X, Z),
    advisor(Y, Z),
    X \== Y.
```

Now, sure enough, we get the right number of siblings.

32.2 Intermission

At this point, we have seen most of the elements of (the core of) Prolog. We’ve seen fact declarations and the expression of rules over them to create extended relations. We’ve also seen that Prolog evaluates programs as simple rule-lookups or as queries (where the former are a special case of the latter). We’ve seen Prolog’s variables, known as logic variables, which can take on multiple values over time as Prolog boldly and tirelessly seeks out new solutions until it has exhausted the space of possibilities. And finally, related to this last step, Prolog backtracks as necessary to find solutions, in accordance with the non-determinism of the rules.

\footnote{For a very subtle reason, we cannot move the last line earlier. We will understand why better once we’ve implemented Prolog.}
32.3 Example: Encoding Type Judgments

Let’s look at another use for Prolog: to encode type judgments. Recall that we had rules of the form

\[ \Gamma \vdash e : \tau \]

where some were axioms and the others were conditionally-defined judgments. The former we will turn into facts, the latter into rules.

First, we must determine a representation for abstract syntax in Prolog. We don’t want to deal with parsing, and we don’t actually distinguish between individual values of a type, so we’ll assume constants have been turned into an abstract node that hides the actual value. Thus, we use the constant \texttt{numConst} to represent all syntactically numeric expressions (i.e., those abstract syntax terms of type \texttt{numE}), \texttt{boolConst} to represent true and false, and so on.

Given this, we will define a three-place relation, \texttt{type}. The first place will be the type environment, represented as a list; the second will be the expression; and the third the type of the expression. (When writing this as a function in a traditional language, we might define it as a two-argument function that computes the expression’s type. But because Prolog can “run backward”, it doesn’t have a distinguished “return”. Instead, what we normally think of as the “result” is just another tuple in the relation.) Our axioms therefore become:

\begin{verbatim}
type(_,numConst,num).
type(_,boolConst,bool).
\end{verbatim}

The _ represents that we don’t care what goes in that position. (We could as well have used a fresh logic variable, but the underscore makes our intent clearer.) That is, no matter what the type environment, numeric constants will always have type num.

The easiest judgment to tackle is probably that for conditional. It translates very naturally into:

\begin{verbatim}
type(TEnv,if(Test,Then,Else),Tau) :-
  type(TEnv,Test,bool),
  type(TEnv,Then,Tau),
  type(TEnv,Else,Tau).
\end{verbatim}

Pay close attention to lower- and upper-case initials! Both \texttt{type} and \texttt{if} are in lower-case: the former represents the type relation, while the latter is the abstract syntax term’s constructor (the choice of name is arbitrary). Everything else is a type variable. (Notice, by the way, that Prolog performs pattern-matching on its input, just as we saw for Haskell.)

Given these two facts and one rule for \texttt{type}, we can ask Prolog to type-check some programs (where [ ] denotes the empty list):

\begin{verbatim}
:- type([],boolConst,bool).
yes
:- type([],if(boolConst,numConst,numConst),num).
yes
:- type([],if(boolConst,numConst,boolConst),num).
no
\end{verbatim}
The implementation of this rule in your type checkers reflected exactly the semantics we gave: *if* the three conditions in the antecedent were met, *then* the consequent holds. In contrast, because Prolog lets us query relations in any way we please, we can instead *use the same implementation* to ask what the type of an expression is (i.e., make Prolog perform type inference):

```
:- type([], boolConst, T).
T= bool
no

:- type([], if(boolConst, numConst, numConst), T).
T=num
no

:- type([], if(boolConst, numConst, boolConst), T).
no
```

It should be no surprise that Prolog “inferred” a type in the first case, since the use precisely matches the axiom/fact. In the second case, however, Prolog used the rule for conditionals to determine solutions to the type of the first expression and matched these against those for the second, finding the only result. In the third case, since the program does not have a type, Prolog fails to find any solutions.

We can now turn evaluation around by asking Prolog strange questions, such as “What expression have type num?”

```
:- type([], T, num).
```

Amazingly enough, Prolog responds with:

```
SOLUTION:
  T=numConst
SOLUTION:
  T=if(boolConst, numConst, numConst)
SOLUTION:
  T=if(boolConst, numConst, if(boolConst, numConst, numConst))
SOLUTION:
  T=if(boolConst, numConst,
       if(boolConst, numConst, if(boolConst, numConst, numConst)))
```

The output here actually gives us a glimpse into the search order being employed by this implementation (notice that it depth-first expands the else-clause of the conditional).

Next let’s deal with identifiers. We’ve said that the type environment is a list; we’ll use a two-place bind relation to track what type each identifier is bound to.

---

4 The output has been indented for readability.

5 We’ll make the simplifying assumption that all bound identifiers in the program are consistently renamed to be distinct.
CHAPTER 32. PROGRAMMING IN PROLOG

\[
\text{type}([\text{bind}(V,T)|\_], \text{var}(V), T).
\]

\[
\text{type}([\text{bind}(\_, \_)|TEnvRest], \text{var}(V), T):=
\hspace{1cm} \text{type}(TEnvRest, \text{var}(V), T).
\]

A quick test:

\[
:- \text{type}([\text{bind}(w, \text{bool}), \text{bind}(v, \text{num})], \text{var}(v), T).
\]

\[
T = \text{num}
\]

Next we’ll specify the rule for functions:

\[
\text{type}(TEnv, \text{fun}(\text{Var}, \text{Body}), \text{arrow}(T1, T2)) :-
\hspace{1cm} \text{type}([\text{bind}(\text{Var}, T1)|TEnv], \text{Body}, T2).
\]

Testing this:

\[
:- \text{type}([], \text{fun}(x, \text{if}(\text{var}(x), \text{numConst}, \text{boolConst})), T).
\]

\[
\text{no}
\]

\[
:- \text{type}([], \text{fun}(x, \text{if}(\text{var}(x), \text{numConst}, \text{numConst})), T).
\]

\[
T = \text{arrow} (\text{bool}, \text{num})
\]

Notice that in the second example, Prolog has determined that the bound identifier must be a boolean, since it’s used in the test expression of a conditional.

Finally, the rule for applications holds no surprises:

\[
\text{type}(TEnv, \text{app}(\text{Fun}, \text{Arg}), T2) :-
\hspace{1cm} \text{type}(TEnv, \text{Fun}, \text{arrow}(T1, T2)),
\hspace{1cm} \text{type}(TEnv, \text{Arg}, T1).
\]

Running it:

\[
:- \text{type}([],
\hspace{1cm} \text{app}(\text{fun}(x, \text{if}(\text{var}(x),
\hspace{1cm} \text{numConst},
\hspace{1cm} \text{numConst})),
\hspace{1cm} \text{boolConst}),
\hspace{1cm} T).
\]

\[
T = \text{num}
\]

Now let’s try some more interesting functions:

\[
:- \text{type}([], \text{fun}(x, \text{var}(x)), T).
\]

\[
T = \text{arrow}(_{2823020}, _{2823020})
\]

This is Prolog’s way of saying that parts of the answer are indeterminate, i.e., there are no constraints on it. In short, Prolog is inferring parameterized types!
Exercise 32.3.1 Are Prolog’s types truly polymorphic? Do they automatically exhibit let-based polymorphism (Section 31)? Write appropriate test expressions and present Prolog’s output to justify your case.

32.4 Final Credits

We’ve now seen even more of Prolog. We’ve encountered the “don’t care” notation. Prolog computes the most general response it can, so if there are no constraints on some part of the answer, it leaves them undefined (using the same symbol to show sharing constraints, as in the inferred type of the identity function). Prolog will match patterns as deep as they are nested, and programmers can use the same variable twice in a rule to indicate that they intend for the values of both to be the same. (Having already seen this with genealogical trees, we made much more extensive use of it to encode type judgments, mimicking the use of meta-variables when we wrote the judgments on paper.)

Putting together the pieces, we found that Prolog was a very convenient encoding of the rules of a type checker. Indeed, for free, we were able to turn our type checker into a type inference engine. Thinking about how we implemented type inference manually may give us some clues as to how to implement Prolog!