B Logic Programming and Prolog

THIS APPENDIX contains a brief discussion of logic programming and Prolog. We present an Athena implementation of a Prolog interpreter for definite clauses, and we also discuss Athena's integration with external Prolog systems such as SWI Prolog. We provide a rudimentary overview of the central concepts of logic programming, at least as they manifest themselves in Athena, but due to space limitations the treatment here cannot be exhaustive. For in-depth introductions to these subjects, consult a textbook [19], [79], [61], [66], [96].

B.1 Basics of logic programming

The key notion in logic programming is that of a *definite Horn clause*.

In the present context a definite Horn clause can be understood as an Athena sentence of the following kinds:

1. A *fact* of the form

   $$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p\texttt{)},$$

   where $n \geq 0$ and p is an atomic sentence whose free variables are exactly $x_1, \ldots, x_n$; or
2. a *rule*, that is, a conditional of the form

   $$\texttt{(forall } x_1 \cdots x_n \texttt{ . } p_1 \texttt{ \& } \cdots \texttt{ \& } p_k \texttt{ ==> } p_{k+1}\texttt{)},$$

   where $k > 0$ and $x_1, \ldots, x_n$ are all and only the free variables that occur in the various p_i. We refer to p_{k+1} as the conclusion or *head* of the rule and to $p_1, \ldots, p_k$ as the *antecedents*. (We may also collectively refer to the entire conjunction of $p_1, \ldots, p_k$ as *the* antecedent of the rule. This is also known as the *body* of the rule in logic programming terminology.)

Facts can also be viewed as rules with the trivial body `true`, and we will occasionally treat them as such, so the distinction between facts and rules is not theoretically essential.

A (definite) *logic program* is simply a finite set of definite Horn clauses, that is, a finite collection of facts and rules. A (definite) Prolog program, by contrast, is a *list* of definite Horn clauses—a logic program where the order in which the clauses are listed is significant. We will see shortly why clause order is significant in Prolog. In addition to clause order, the order in which one lists the atoms of a rule's body is also significant in Prolog (whereas in the world of pure logic that order is immaterial, given that conjunction is commutative and associative).

The rules and facts of a logic program collectively express a declarative body of information: They state *what is* (or what might be) the case in a given domain. The programs are called definite because, viewed as theories (as sets of first-order logic sentences),

they always have unique minimal models, namely, the so-called *least Herbrand models*. Roughly speaking, the least Herbrand model of a logic program $\mathcal{P}$ is the set of sentences obtained by starting with the facts in $\mathcal{P}$ and then adding all possible consequences thereof, possibly infinitely many, as dictated by the rules in $\mathcal{P}$, where a sentence counts as "a possible consequence" only if it involves ground terms that can be formed with symbols that occur in the program.[1] For example, suppose we have a domain `D`, a constant `a` of sort `D`, a unary function symbol `f` from `D` to `D`, and a unary relation symbol `P` on `D` (a function symbol from `D` to `Boolean`). And suppose that $\mathcal{P}$ consists of the sole rule

```
(forall x . P x ==> P f x)
```
(1)

and the sole fact `(P a)`. Then the least Herbrand model of this program would be the following infinite set of sentences:

```
{(P a), (P (f a)), (P (f (f a))), (P (f (f (f a)))), ...}.
```

The model is said to be the "least" or "smallest" Herbrand model in that it can be shown that it is contained in (it is a subset of) *every* Herbrand model of the program, that is, it is contained in every set of facts over the Herbrand universe that satisfies the program. If we allowed disjunction in rule heads this property (the existence of a unique smallest model) would no longer hold.

As a stylistic point, a rule such as (1) is typically written in the reverse direction, as `(forall x . P f x <== P x)`, and the quantification is also implicit, with the tacit understanding that all variables in the rule are universally quantified. The backward direction is more in keeping with the *operational meaning* of such a rule, which can be loosely put as follows:[2] *To solve a goal of the same form as the head of the rule, solve the subgoals comprising the body of the rule*. To figure out whether a goal is "of the same form" as the head of a rule, we use unification. If successful, this produces a substitution θ, and we then recursively proceed to solve the subgoals obtained by applying θ to the atoms in the body of the rule. This process, which will be made fully precise in Section B.3, is called *backchaining* on the rule, and can be viewed as the inverse of modus ponens: Instead of proceeding from the antecedent(s) to the conclusion, we go from the conclusion to the antecedents. This algorithm must eventually be grounded (it must terminate) in the facts of the program, or otherwise we might keep backchaining indefinitely. As mentioned earlier, a fact can be regarded as a rule with the trivial body `true`, so backchaining on a fact immediately terminates successfully, assuming that the goal is of the same form as—unifies with—the (head of the) fact.

1 The set of all such ground terms is called the *Herbrand universe* of $\mathcal{P}$. To ensure that this set is nonempty, it is typically assumed that the program contains at least one constant symbol.

2 As opposed to their declarative meaning, which is the usual meaning as dictated by the standard Tarskian semantics of first-order logic (as discussed in Section 5.6).

Definite programs can be viewed as *positive*, insofar as they only express affirmative information. They cannot express negative rules such as "a man is a person who is not a woman." It is possible to extend logic programming to allow for negative clauses of that form, called *general clauses*. Neither the semantics nor the computational interpretation of such extensions are entirely satisfactory, and we do not discuss them here. Nevertheless, negation is often useful in clause bodies. For instance, many conditional equations have negative conditions, and it is useful to encode these directly when we translate such equations into clausal form, as we do in Section B.4. Full-blown Prolog implementations do allow for negated subgoals via a mechanism known as (finite) *negation as failure*, and since Athena is integrated with such systems, one can work with general clauses via the existing translation, as described in Section B.4.

So how do we compute with a logic program $\mathcal{P}$? By issuing *queries*, which are then automatically answered by a logic program engine. A simple query is just an atomic sentence, such as `(father-of adam peter)` or `(parent-of ?X peter)`. If the query is ground, like `(father-of adam peter)`, then it is basically asking whether the ground atom in question holds, so the answer will be a simple yes or no, respectively indicating that the ground atom does or does not follow from $\mathcal{P}$. If the query contains variables, like

```
(parent-of ?X peter),                                    (2)
```

then it can be understood as asking whether there are any values for these variables that make the atom true according to the program. For instance, is there a value of `?X` that makes (2) true according to `person-database`, the small program introduced in the first example of the next section? A positive answer in this case will be a *substitution* from the query variables to appropriate terms, such as

$$\{\texttt{?X:Person} \rightarrow \texttt{seth}\},$$

that renders the query a logical consequence of $\mathcal{P}$.[3] This is largely where the power of logic programming stems from: the ability to not merely confirm or deny, but to also *discover* objects that satisfy a given query.[4] As the queries become more complex, this ability to discover objects that may be related in arbitrarily complicated ways becomes increasingly useful. By contrast, a negative answer for a nonground query would indicate that *no* substitution instance of the query follows logically from $\mathcal{P}$.

A query need not be a simple atom. It may be conjunctive, typically represented by a *list* of atoms. An example might be

```
[(parent-of adam ?X) (parent-of ?X ?Y)].
```

3 More precisely, the sentence obtained by applying the substitution to the query is a logical consequence of $\mathcal{P}$.

4 Such queries might look familiar to readers who have been exposed to query languages like SQL or SPARQL. There is indeed a very close connection between logic programming and database (or knowledge base) queries, which is studied in the field known as *deductive databases* [106].

A positive answer here will again be a substitution which, when applied to the query atoms, will yield instances that are logical consequences of $\mathcal{P}$. In this case a substitution will give us a value for `?X`, representing a child of `adam`, *and* a value for `?Y`, representing a child of `?X`. Thus, in one fell swoop we will retrieve both a child and a grandchild of `adam`. As before, a negative answer will indicate that there is no (joint) ground instance of the query atoms that follows logically from $\mathcal{P}$.

There may well be multiple substitutions (indeed, potentially infinitely many) resulting in ground logical consequences of $\mathcal{P}$. Prolog engines are able to incrementally produce as many of these as desired, as is the interpreter we implement in Section B.3.

In addition to the possibilities already mentioned, there is another alternative for what might happen when a user poses a query: nontermination. The query engine might get into an infinite loop. This is where the aforementioned issue of clause and body orderings comes into the picture. Naive ways of expressing clauses and programs can easily send a Prolog system into an infinite loop. It takes some experience to write logic programs in a way that avoids such pitfalls.

B.2 Examples

Let us look at some examples to get a feeling for what we can do with logic programming. Queries will be answered here by the interpreter that we present in the next subsection. We will look at two domains, some genealogical relationships and a few arithmetic operations (addition and multiplication) on the natural numbers. We start with the former:

```
domain Person

declare father-of, mother-of, parent-of, grandparent-of, ancestor-of:
        [Person Person] -> Boolean

declare adam, eve, seth, peter, mary, paul, joe: Person

define facts :=
   [(adam father-of seth)
    (eve mother-of seth)
    (seth father-of peter)
    (peter father-of paul)
    (mary mother-of paul)
    (paul father-of joe)]

define rules :=
 (close [(x parent-of y <== x father-of y)
         (x parent-of y <== x mother-of y)
         (x grandparent-of z <== x parent-of y & y parent-of z)
         (x ancestor-of y <== x parent-of y)
         (x ancestor-of z <== x ancestor-of y & y parent-of z)])
```

```
define person-database := (facts joined-with rules)
```

We now answer a few queries against this program/database. The main tool we use to answer queries is the binary procedure `Prolog_Interpreter.solve`, which takes a query (either an atom or a list thereof) and a list of definite clauses and returns either a substitution θ such that the θ-instance of the query[5] is a logical consequence of the given clauses; or `false` if no such substitution can be found. If the query is a ground fact that holds,[6] or a conjunction or list of such facts, then the empty substitution is returned.

We also use the ternary procedure `Prolog_Interpreter.solve-N`, whose first two arguments are the same as those of `Prolog_Interpreter.solve`, and whose third argument is a nonnegative integer N indicating the desired number of solutions (substitutions). If at least N satisfying substitutions can be found, they are returned in a list. If fewer than N substitutions can be found, they are also returned in a list. If no satisfying substitutions can be found, then the empty list is returned. Some examples:

```
define (solve q) := (Prolog_Interpreter.solve q person-database)

define (solve-N q N) := (Prolog_Interpreter.solve-N q person-database N)

set-precedence solve 80

> (solve paul father-of joe)

Substitution: {}  # An empty substitution indicates a fact

> (solve paul mother-of joe)

Term: false       # false means the query cannot be proved/solved

> (solve ?X father-of peter))

Substitution: {?X:Person --> seth}

> (solve seth father-of ?Y)

Substitution: {?Y:Person --> peter}

> (solve ?X father-of ?Y)

Substitution:
{?Y:Person --> seth
?X:Person --> adam}
```

5 Meaning the sentence we obtain by applying θ to the conjunction of all the query atoms.

6 A sentence "holds" in this context iff it is a member of the input program/database, *not* if it is in the assumption base.

```
> (solve-N (?X father-of ?Y) 10)

List: [
{?X:Person --> adam
?Y:Person --> seth}


{?X:Person --> seth
?Y:Person --> peter}

{?X:Person --> peter
?Y:Person --> paul}

{?X:Person --> paul
?Y:Person --> joe}]
```

Note that the last result gave us four substitutions listing all pairs of fathers and sons in the database.

We continue with some more interesting queries that essentially require reasoning (using the rules):

```
> (solve eve grandparent-of ?X)

Substitution: {?X:Person --> peter}

> (solve eve ancestor-of ?X)

Substitution: {?X:Person --> seth}

> (solve-N (eve ancestor-of ?X) 10)

    ...
```

The last query sends the system into an infinite loop. The reason is the last of the given rules, namely

```
(x ancestor-of z <== x ancestor-of y & y parent-of z).
```

The problem here is that we have listed `(x ancestor-of y)`, a recursive subgoal (recursive in the sense that its predicate is the same as the predicate in the head of the rule), as the *first* subgoal in the body of the rule, which means that backchaining on this rule for certain goals will lead to an infinite regress. This will become clearer in the next subsection. Once we reformulate the rule as

```
(x ancestor-of z <== x parent-of y & y ancestor-of z)
```

and reload, we get the following complete list of all of Eve's ancestors:

```
> (solve-N (eve ancestor-of ?X) 10)

List: [{?X:Person --> seth}
       {?X:Person --> peter}
       {?X:Person --> paul}
       {?X:Person --> joe}]
```

This was an example of a case where the order of the subgoals in the body of a rule is important. In the next section we will see an example illustrating the importance of rule ordering.

Let us now give some definite clauses for addition and multiplication on the natural numbers. First, we introduce ternary predicates `plus-p` and `times-p` such that

(plus-p *x y z*)

holds iff *z* is the sum of *x* and *y*, and likewise for `times-p`:

```
declare plus-p, times-p: [N N N] -> Boolean [[int->nat int->nat int->nat]]
```

We now define `plus-p` by the following definite clauses:

```
define plus-clauses :=
   (close [(plus-p x zero x)
           ((plus-p x (S y) (S z)) <== (plus-p x y z))])
```

Compare this relational definition with a functional definition of natural-number addition in the style given in the text:

```
assert* Plus-def := [(x + zero = x)
                     (x + S y = S (x + y))]
```

The similarities here are deeper than the differences. If we read `plus-clauses` backward, then the first clause says that every instance of `(plus-p x zero x)` holds: For any `x`, the sum of `x` and `zero` is `x`, which is precisely the content of the first equation in the functional definition. The second clause says that to prove a goal of the form `(plus-p x (S y) (S z))`, we first need to prove the subgoal `(plus-p x y z)`. Thus, if the first operand to the addition is any number `x` and the second operand is nonzero (of the form `(S y)`), then we know that the result will be nonzero, of the form `(S z)`, and the precise value of `z` can be computed by backchaining and solving the subgoal `(plus-p x y z)`.

```
> (Prolog_Interpreter.solve (plus-p 1 2 ?RESULT) plus-clauses)

Substitution: {?RESULT:N --> (S (S (S zero)))}
```

To increase readability, let us print out substitutions so that natural numbers are written as integers:

```
define (transform-sub sub) :=
  (map lambda (v) [v --> (nat->int (sub v))]
       (supp sub))

define (solve q program) :=
  match (Prolog_Interpreter.solve q program) {
    (some-sub sub) => (transform-sub sub)
  | r => r}

define (solve-N q program N) :=
  (map transform-sub (Prolog_Interpreter.solve-N q program N))
```

The following queries begin to illustrate the flexibility of logic programming:

```
> (solve (plus-p 3 ?Y 10) plus-clauses)

List: [[?Y --> 7]]
```

This shows that logic programming is not unidirectional, meaning that it need not always go from fixed inputs to an output. We can just as well fix the output and some inputs and solve for other inputs, or fix only the output and solve for any inputs (as does the next query). Queries are viewed essentially as constraints, and the logic programming engine attempts to find any or all values satisfying the constraints. In this particular example we used the addition clauses to essentially perform subtraction: We asked what number `?Y` we must add to 3 in order to obtain 10, and we got the answer 7, which is of course the result of subtracting 3 from 10.

The following query produces all ways in which `?X` and `?Y` can add up to 3:

```
> (solve-N (plus-p ?X ?Y 3) plus-clauses 10)

List: [[[?Y --> 0] [?X --> 3]]
       [[?Y --> 1] [?X --> 2]]
       [[?Y --> 2] [?X --> 1]]
       [[?Y --> 3] [?X --> 0]]]
```

We can likewise define multiplication as follows:

```
define times-clauses :=
   (close [(times-p x zero zero)
           ((times-p x (S y) z) <== (times-p x y w) & (plus-p x w z))])

define num-clauses := (plus-clauses joined-with times-clauses)

> (solve (times-p 2 3 ?RES) num-clauses)

List: [[?RES --> 6]]
```

Reading the clauses backward, the first says that every goal of the form

```
(times-p x zero zero)
```

holds, which is another way of saying that the product of any number with zero is zero. The second clause says that to solve a goal of the form `(times-p x (S y) z)`, we first need to solve `(times-p x y w)`, and then solve `(plus-p x w z)`. In other words, letting `w` be the product of `x` and `y`, the product `z` of `x` and `(S y)` is `(x + w)`, that is, `(x + (x * y))`, which is precisely how an equational formulation would express this constraint. As before, however, we can not only compute products with these clauses, but quotients too! For example, the following computes 12 divided by 4:

```
> (solve (times-p 4 ?X 12) num-clauses)

List: [[?X --> 3]]
```

B.3 Implementing a Prolog interpreter

A nondeterministic algorithm $\mathcal{A}$ for answering a list of queries $L = [q_1 \cdots q_n]$ (or "solving the goals $q_1, \ldots, q_n$") with respect to a definite logic program $\mathcal{P}$ can be formulated as follows. The algorithm takes as input not just the list of goals L and the program $\mathcal{P}$, but also a substitution θ. The initial value of θ will be the empty substitution {}. As $\mathcal{A}$ progresses, this substitution will become increasingly specialized. The algorithm, which will either return a substitution or else fail, is simple:

1. If L is empty ($n = 0$), return θ.
2. Otherwise let θ' be the substitution obtained by executing q_1 against $\mathcal{P}$ and with respect to θ, and apply the algorithm recursively to $[(\theta'\ q_2)\ \cdots\ (\theta'\ q_n)]$, $\mathcal{P}$, and θ'.

The only undefined operation here is that of "executing a query q against $\mathcal{P}$ and with respect to θ," an operation that is supposed to return a substitution θ'. This is actually where the nondeterminism comes in. We perform this operation by first choosing a (freshly renamed) rule R in $\mathcal{P}$ whose head unifies with $(\theta\ q)$ under some substitution σ (if there is no such rule we raise an exception and the whole algorithm halts in failure); and then recursively applying $\mathcal{A}$ to (a) the list of atoms comprising the body of R; (b) $\mathcal{P}$, and (c) the composition of σ and θ. If R is a fact, then this second step can be omitted and we can simply return the composition of σ and θ. Note that freshly renaming the rules (alpha-renaming them, to be more precise) is important. Without such renaming we could get incorrect results.

There are two sources of nondeterminism here: the choice of R and the choice of how to list the subgoals (atoms) in the body of R. By somehow making all the "right choices" along the way, this algorithm is guaranteed to arrive at a solution if one exists. In other words this algorithm is both *sound*, meaning that if it produces a substitution θ then all

θ-instances of the input queries are logical consequences of $\mathcal{P}$; and *complete*, meaning that if there exists such a substitution at all, the algorithm will return one. Unfortunately, deterministic versions such as the one we are about to formulate (the same algorithm used by Prolog systems) are incomplete, as satisfying substitutions might only exist in parts of the search space that will never be explored by a fixed search strategy.

To arrive at a deterministic algorithm, we arrange all possible computations of $\mathcal{A}$ in one big search tree, called an *SLD tree*, and fix a strategy for exploring that tree. The nodes of the tree will be pairs (L,θ) consisting of a list of goals (atoms) $L = [q_1 \cdots q_n]$ and a substitution θ.

Given a pair (L,θ), the SLD tree for (L,θ) is built recursively: The root of the tree is (L,θ), and its children are defined as follows. If L is empty then there are no children—the node is a leaf. Otherwise L is of the form $[q_1 \cdots q_n]$ for $n > 0$. Let $R_1,\ldots,R_k$ be all and only those (freshly renamed) rules in $\mathcal{P}$ whose head unifies with q_1 under some substitution θ_i, where $R_1,\ldots,R_k$ are listed according to the order in which they appear in $\mathcal{P}$.[7] (If $k = 0$, i.e., if there are no such matching rules, then the node is again a leaf—there are no children.) For $i = 1,\ldots,k$, let σ_i be the composition of θ_i with θ. Then the node (L,θ) has exactly k children, namely

$$(body(R_1,\sigma_1,L),\sigma_1),\ldots,(body(R_k,\sigma_k,L),\sigma_k),$$

in that order, with $body(R_i,\sigma_i,L)$ defined as

$$[(\sigma_i\ A_1^i),\ldots,(\sigma_i\ A_{m_i}^i),(\sigma_i\ q_2),\ldots,(\sigma_i\ q_n)],$$

where $A_1^i,\ldots,A_{m_i}^i$ are the atoms that appear in the body (antecedent) of R_i, in the same left-to-right order in which they appear in R_i. We then proceed in the same fashion to construct the children of each of these children, and so on. This process need not terminate if the tree is infinite.

The `solve-all` algorithm given below works by building/exploring the SLD tree for the pair consisting of the input list of queries and the empty substitution in a depth-first manner, maintaining at all times what is essentially a stack of nodes to explore,

$$[[L_1\ \theta_1]\ \cdots\ [L_m\ \theta_m]],$$

and starting off with the one-element stack

$$[[L\ \{\}]],$$

where L is the input list of queries. The main loop, implemented by the recursive procedure `search`, keeps popping the leftmost pair off the stack, building its children, and pushing these back on the stack. Note that the stack is encoded as the (only) argument of `search`.

By using higher-order procedures—essentially continuations—this algorithm not only returns a substitution if there is one, but also returns a *thunk*, that is, a nullary procedure

7 Recall that $\mathcal{P}$ is a list, not a set.

that can be invoked later to obtain yet more satisfying substitutions, if any exist. That thunk will continue exploring the SLD tree at the point where the procedure left off last time, namely, at the point where the last satisfying substitution was found. Calling that thunk will again result in a pair consisting of a new substitution and yet another thunk, and so on. When there are no more substitutions to be found, a thunk will return the empty list `[]` rather than a substitution-thunk pair. In this way it is possible to eventually obtain *every* satisfying substitution that is reachable at all by a depth-first strategy, even if there are infinitely many such substitutions.

```
module Prolog_Interpreter {

define (restrict sub L) :=
  let {V := (vars* L)}
    (make-sub (list-zip V (sub V)))

define (conclusion-of P) :=
   match P {
    (forall (some-list _) (_ ==> p)) => p
  | (forall (some-list _) p) => p
  }

define (match-conclusion-of p goal) :=
  let {q := (rename p)}
    match (goal unified-with conclusion-of q) {
      (some-sub sub) => [q sub]
    | _ => false
    }

define (get-matches goal db) :=
  (filter (map lambda (rule) (match-conclusion-of rule goal) db)
          (unequal-to false))

define (subgoals-of clause) :=
  match clause {
    (forall (some-list _) (p ==> _)) => (get-conjuncts p)
  | _ => []
  }

define (make-new-goals previous-sub previous-goals) :=
  lambda (pair)
    match pair {
      [clause sub] =>
        let {new-goals := (sub (join (subgoals-of clause) previous-goals))}
          [new-goals (compose-subs sub previous-sub)]
    }

define (solve-all queries db) :=
  letrec {search := lambda (L)
                      match L {
```

```
                        [] => []
                      | (list-of [[] sub] rest) =>
                          [(restrict sub queries) lambda () (search rest)]
                      | (list-of [(list-of goal more-goals) sub] rest) =>
                          match (get-matches goal db) {
                            [] => (search rest)
                          | rule-matches =>
                              let {L' := (map (make-new-goals sub more-goals)
                                              rule-matches)}
                               (search (join L' rest))
                          }}}
    (search [[queries empty-sub]])

} # module Prolog_interpreter
```

Given `solve-all`, it is easy to implement a `solve-N` procedure that will attempt to retrieve N solutions, if N solutions exist, otherwise producing the maximum possible number $K < N$ of solutions:

```
1  define (solve-N queries program N) :=
2    let {queries := match queries {
3                      (some-sent p) => [p]
4                    | _ => queries}}
5      letrec {loop :=
6                lambda (i res thunk)
7                  check {(greater? i N) => (rev res)
8                       | else => match (thunk) {
9                                   [] => (rev res)
10                                | [sub thunk'] => (loop (i plus 1)
11                                                        (add sub res)
12                                                        thunk')
13                                }
14                 }
15             }
16       match (solve-with-thunks queries program) {
17         [] => []
18       | [sub rest] => (loop 2 [sub] rest)
19       }
```

Note that the code on lines 2–4 ensures that the procedure works when the first input argument is either a list of atoms or a single atom by itself. We can likewise implement a `solve` procedure that either returns a single substitution, if one can be found, or else returns `false`:

```
define (solve queries program) :=
  match (solve-N queries program 1) {
    [] => false
  | (list-of sub _) => sub
  }
```

Both `solve` and `solve-N` are parts of the `Prolog_Interpreter` module. Let us test the system on our earlier example:

```
domain D
declare a: D
declare f: [D] -> D
declare P: [D] -> Boolean
define clauses := (close [(P a)
                          (P f x <== P x)])

open Prolog_Interpreter

> (solve (P a) clauses)

Substitution: {}

> (solve (P f ?X) clauses)

Substitution: {?X:D --> a}

> (solve-N (P f ?X) clauses 2)

List: [{?X:D --> a}
       {?X:D --> (f a)}]

> (solve-N (P f ?X) clauses 10)

List: [{?X:D --> a}
       {?X:D --> (f a)}
       {?X:D --> (f (f a))}
       {?X:D --> (f (f (f a)))}
       {?X:D --> (f (f (f (f a))))}
       {?X:D --> (f (f (f (f (f a)))))}
       {?X:D --> (f (f (f (f (f (f a))))))}
       {?X:D --> (f (f (f (f (f (f (f a)))))))}
       {?X:D --> (f (f (f (f (f (f (f (f a))))))))}
       {?X:D --> (f (f (f (f (f (f (f (f (f a)))))))))}]
```

It should be clear by now why rule ordering (as well as the ordering of rule bodies) is important. Rules are tried in the textual order in which they are listed in a program (list of clauses), so a recursive rule might send our depth-first strategy into an infinite loop if it is listed before other rules for the same predicate. For example:

```
define clauses' := (close [(P f x <== P x)
                           (P a)])

> (solve (P f ?X) clauses')

     ... # infinite loop...
```

B.4 Integration with external Prolog systems

The Prolog interpreter we have presented is adequate for small problems and useful for instructive purposes, but it has two serious drawbacks. First, it doesn't handle negation. Second, and more important, the interpretation layer incurs a severe efficiency penalty. The performance of this interpreter on realistic problems would be unacceptable, as would indeed be the performance of any Prolog interpreter.

The first issue, negation, is not too serious; it would not be difficult to modify our interpreter so as to implement finite negation as failure. But efficiency is a more pressing concern. To get Prolog search to run fast, Prolog must be compiled into machine code.

For these reasons, Athena is integrated with two state-of-the-art Prolog systems, SWI Prolog and XSB, that can execute Prolog queries much more efficiently. The integration is seamless in that both queries and programs are expressed in Athena notation, as is the output. The integration with XSB is tighter, in that XSB can be started when Athena starts and thereafter the two programs will communicate at the process level. This has some efficiency advantages, as new knowledge (facts or rules) can be added incrementally. XSB is also a *tabled* Prolog system, which enables a whole class of interesting applications that are not otherwise possible. However, integration with XSB is only available on Linux, and not by default, so in what follows we only describe the connection to SWI Prolog. That connection works on a per-query basis: To solve a given query (or list of queries) with respect to a given list of clauses, Athena prepares an equivalent Prolog program and query, invokes SWI Prolog from scratch on these inputs, and then translates the output back into Athena notation.[8] Because this approach involves some file IO and starting the `swipl` process from scratch, there is a fixed runtime cost associated with it.

The main API is captured by three procedures in the module `Prolog`:

1. `(solve` *goal program*`)`;
2. `(solve-all` *goal program*`)`;
3. `(solve-N` *goal program N*`)`.

We describe each in turn.

In a call of the form `(solve` *goal program*`)`, *goal* is either a single query (atom) or a list of queries, and *program* is a list of (general) Horn clauses. Disjunctions of literals may also appear in the bodies of these clauses. A clause may be explicitly universally quantified, but need not be. If it is not quantified, it is tacitly assumed that all variables in it are universally quantified. In fact, *program* may contain completely arbitrary sentences, for instance, one may give the entire assumption base as the value of *program*, with a call of the

8 SWI Prolog must be locally installed on your machine and must be runnable from the command line with the command `swipl`.

form (solve *goal* (ab)). The implementation of Prolog will automatically try to extract general Horn clauses from the given sentences, and will simply ignore those sentences that it cannot convert into Horn clauses.

The output is always a two-element list whose first element is true or false, respectively indicating success or failure. When the first element of the output list is true, the second element is a substitution, which, when applied to the query goals, results in sentences that follow logically from the clauses. Some examples:

```
domain D
declare a,b,c: D
declare P: [D] -> Boolean
declare R: [D D] -> Boolean

define program := [(P a) (P b) (a R b) (b R c)]

> (Prolog.solve (P a) program)

List: [true {}]

> (Prolog.solve (P ?X) program)

List: [true {?X:D --> a}]

> (Prolog.solve (R ?X ?Y) program)

List: [true
       {?Y:D --> c
        ?X:D --> b}]

> (Prolog.solve (R ?X ?X) program)

List: [false {}]
```

The arguments of solve-all are the same as those of solve, but the result is a *list of all* satisfying substitutions (assuming there is a finite number of them). Failure here is signified by an empty output list, which means that no substitutions could satisfy the goal(s).

```
> (Prolog.solve-all (P ?X) program)

List: [{?X:D --> a} {?X:D --> b}]

> (Prolog.solve-all (R ?X ?X) program)

List: []
```

Note that the goal(s) given to solve-all must contain at least one variable. If the goal(s) are all ground, use solve instead.

Finally, `solve-N` takes an extra argument at the end, a positive integer N indicating the number of desired solutions. However, this is implemented by obtaining one substitution, then calling Prolog again with an explicit added constraint that rules out the first substitution; then calling Prolog again with yet another constraint that rules out the first two substitutions; and so on. This is not particularly efficient. It may be more expedient to compute a list of all solutions (using `solve-all`) and then simply take N elements from that list. However, if there are infinitely many solutions discoverable by depth-first search, then `solve-all` will diverge, so in that case `solve-N` is a more viable alternative.

There are timed versions of the above three procedures that can take a nonnegative integer M as a last argument, which will then force Prolog to quit after S seconds have gone by, if it is not done by then:

1. `Prolog.solve-with-time-limit`;
2. `Prolog.solve-all-with-time-limit`; and
3. `Prolog.solve-N-with-time-limit`.

(For the last procedure, the time limit applies to each of the N separate calls, so if you want all solving to last no more than M seconds, give a time limit of M divided by N.)

The equality symbol, when it appears in clauses, has the usual Prolog semantics (unifiability). There are a number of predefined Prolog symbols introduced in the `Prolog` module that can also be used in queries:

- The binary polymorphic symbols `Prolog.==` and `Prolog.\==` represent strict term identity and inequality, respectively.
- The binary symbols `Prolog.arith-eq` and `Prolog.arith-uneq` represent numeric Prolog identity and inequality; they are polymorphic in both argument positions.
- The ternary polymorphic symbols `setof`, `findall`, and `bagof` represent the corresponding Prolog symbols. The results are collected into Athena lists. For example:

```
> (Prolog.solve (Prolog.findall ?X (P ?X) ?Result) program)

List: [true
       {?Result:(List D) --> (a :: b :: nil:(List D))
        ?X:D --> ?v53752:D}]
```

As in Prolog, the first argument can be an arbitrarily complex term containing variables to be solved for:

```
> (Prolog.solve (Prolog.findall (?X :: ?Y :: nil)
                                (?X R ?Y)
                                ?Result)
                program)

List: [true
```

```
{?Result:(List (List D)) --> (:: (:: a (:: b nil:(List D)))
                                 (:: (:: b (:: c nil:(List D)))
                                     nil:(List (List D))))
          ?Y:D --> ?v1663:D
          ?X:D --> ?v1662:D}]
```

Here the results were assembled into a list of two lists, essentially the pairs `[a b]` and `[b c]`, that is, all and only the pairs of ground terms for which `R` holds according to the given program.[9]

- The `Prolog` module also introduces the symbols `cut`, `fail`, `call`, and `once`, whose use should be clear to experienced Prolog programmers. They can be used to provide greater control over the Prolog search. The `call` predicate is particularly useful for metaprogramming.
- The `Prolog.write` symbol, with signature `[Ide] -> Boolean`, can be used to perform output from inside Prolog. This can be useful for debugging purposes.

We stress that the inputs and outputs of these procedures are given entirely at the Athena level. Thus, for instance, syntactic idiosyncrasies of Prolog are immaterial: Any legal Athena variables and function symbols may appear inside clauses and queries, no matter how complicated, even if they are not legal Prolog syntax. To further facilitate output, one can specify "term transformers" that are automatically applied to resulting substitutions in order to print their content in a more customized (and presumably more readable) format. This is done by invoking the unary procedure `Prolog.add-transformer` as follows:

`(Prolog.add-transformer` *f*`)`,

where *f* is a unary procedure that takes a term and produces a term. For instance, to print natural numbers as integer numerals we may say:

```
(Prolog.add-transformer nat->int).
```

Multiple transformers can be specified in this way, and all of them will be applied to any substitution returned by any of the search procedures we have discussed so far.[10] Because a transformer may change the sort of a term, if there are any transformers specified then the returned results are no longer substitutions but rather lists of triples of the form

`[`*v* `-->` *t*`]`,

where *v* is a query variable and *t* is a transformed term.

9 More precisely, according to the completion of the given program.

10 However, the order in which these are applied is not guaranteed. If that is important then it might be better to specify one single transformer that performs all the necessary conversions in the desired order.

Moreover, as mentioned earlier, the "program" that gets passed to these procedures may contain arbitrary Athena sentences. Athena will try to extract appropriate Horn clauses from any sentences given as clauses.

B.5 Automating the handling of equations

The procedures described in the previous section do not properly address equations, simple or conditional. Consider, for instance, the defining equations for addition on natural numbers:

```
assert* Plus-def := [(x + zero = x)
                     (x + S y = S (x + y))]
```

These are universally quantified identities, so the naive approach of these procedures will translate them as facts, with the identity symbol = as the main predicate. But to properly capture the semantics of such equations in the setting of logic programming, we need new predicate symbols and new clauses for those symbols, and some of these clauses (the ones that correspond to recursive equations) will be proper rules rather than facts. In this particular case, for example, we would need to introduce a new ternary relation symbol `plus-p`, as we did earlier, and introduce new clauses that capture the computational content of the above equations:

```
define plus-clauses :=
   (close [(plus-p x zero x)
           ((plus-p x (S y) (S z)) <== (plus-p x y z))])
```

As the equations get more complex, this can get cumbersome. Accordingly, the `Prolog` module provides a unary procedure, `auto-solve`, that attempts to automate this process. This procedure is unary so it only takes a goal, not a program; it will attempt to automatically construct the right logic program for the given goal, by examining the assumption base and the form of the goal. Specifically, the procedure takes a goal like

```
(2 N.+ 3 = ?Result)
```

and automatically tries to:

1. Determine the (transitive closure of all the) defining equations of all function symbols that appear in the goal.
2. Introduce predicate symbols for all these symbols. Typically, the predicate symbol introduced for a function symbol f is f_p.[11]
3. Translate all defining equations and conditional equations for all of these symbols into general Prolog clauses.

11 That can differ if there is an already existing symbol by that name.

4. Include any other (nonequational) clauses that may be necessary for some of these symbols.
5. Transform the query itself into relational form.
6. Solve the transformed query with respect to the Prolog program produced the previous steps.
7. Output the results.

For example, assuming we have defined `N.+` and `N.*` with the usual recursive equations:

```
define goal := (2 N.+ 3 = ?sum-of-2-and-3)

> (Prolog.auto-solve goal)

List: [true {?sum-of-2-and-3:N --> (S (S (S (S (S zero)))))}]
```

If we wish to display natural numbers as integer numerals:

```
(Prolog.add-transformer nat->int)

> (Prolog.auto-solve goal)

List: [true [[?sum-of-2-and-3 --> 5]]]
```

There is also a unary `auto-solve-all` and a binary `auto-solve-N` procedure, whose second argument is the desired number of solutions:

```
define goal := (2 N.* ?X = ?Y)

> (Prolog.auto-solve-N goal 3)

List: [[[?Y --> 0] [?X --> 0]]
       [[?Y --> 2] [?X --> 1]]
       [[?Y --> 4] [?X --> 2]]]
```

We can inspect the functional-to-relational transformation by asking to see the constructed clausal program for a given goal by using the procedure `Prolog.defining-clauses`:

```
define goal := (2 N.* ?X = ?Y)

> (Prolog.defining-clauses goal)

List: [
(n_+_p ?n zero ?n)

(if (n_+_p ?n ?m ?v784)
    (n_+_p ?n
           (S ?m)
           (S ?v784)))
```

```
(n_*_p ?x zero zero)

(if (and (n_*_p ?x ?y ?v785)
         (n_+_p ?v785 ?x ?v786))
    (n_*_p ?x
           (S ?y)
           ?v786))
]
```

Here, `n_+_p` and `n_*_p` are the ternary predicate symbols that were automatically introduced for `N.+` and `N.*`, respectively.

As we have seen, of course, order is important, both for clauses and for body literals. The implementation of `auto-solve` and its two sibling procedures applies a few simple heuristics in order to determine the best possible ordering for clauses and goal literals, but the heuristics are not perfect. If these procedures fail to produce the expected results, it might be useful to look at the clause list produced for a given goal, using `Prolog.defining-clauses`, possibly make some changes to it, and then apply `Prolog.solve` (or `solve-all`, etc.) to the given goal and the modified clause list.

We can do the same thing for a goal; that is, we can inspect the clauses that are automatically produced for it (via `auto-solve`, etc.), with the unary procedure `Prolog.query-clauses`:

```
define goal := (2 N.* ?X = ?Y)

> (Prolog.query-clauses goal)

List: [
(n_*_p (S (S zero))
       ?X
       ?v785)

(= ?v785 ?Y)
]
```

Again, by applying these two procedures to a given goal, we can obtain and possibly rearrange clause lists for both the extracted program and the goal itself, and then use `Prolog.solve`, etc., to solve with respect to both.