Logic programming

Most computations are directed, progressing from an input to an output.

In functional programming, this is made very explicit, as the input is the argument of a function and the output is its result.

We have already seen an exception: solving constraints.

We defined a set of relations that the computer could "solve" in various directions.

Logic programming adds two ideas to this paradigm of relational programming:

- The idea that a solution is found through a *query* that can test multiple alternatives.
- A type of symbolic pattern matching called *unification*.

Prolog

The most prevalent logic programming language is *Prolog*.

Prolog is an acronym for "*pro*grammation en *log*ique" ("programming in logic").

It was developed in the 70s by Alain Colmerauer, originally to be used for parsing natural languages.

Prolog is used in artificial intelligence applications, such as expert systems, knowledge bases and natural language processing.

Just like Lisp, Prolog is a small language with a simple syntax and without static typing.

The following two implementations are free and available on various platforms:

- GNU Prolog (`http://gprolog.org`)
- SWI-Prolog (`http://www.swi-prolog.org`)

Example : append

The function append is defined as follows in Scala:

```
def append[A](xs: List[A], ys: List[A]): List[A] = xs match {
  case Nil => ys
  case x :: xs1 => x :: append(xs1, ys)
}
```

This function can be seen as a translation of the following two rules:

1. For all lists ys, the concatenation of the empty list and ys is ys.
2. For all x, xs1, ys, zs, if the concatenation of xs1 and ys is zs, then the concatenation of x :: xs1 and ys is x :: zs.

In Prolog, these two rules can be written as follows:

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Remarks

- Variables and parameters in Prolog start with an uppercase letter, e.g. `X`, `Xs`, `Ys`.
- `[...|...]` is the "cons" of lists, e.g. `[X|Xs]` is written as `X :: Xs` in Scala.

Predicates

In Prolog, append is called a *predicate*.

A predicate is nothing else than a procedure that can succeed or fail.

Note that the result of a Scala function is now an additional parameter.

Clauses

Predicates are defined using *clauses*, which can be *facts* (also called axioms) or *rules*.

- `append([], Ys, Ys)` is a fact; it establishes that concatenating `[]` and `Ys` results in `Ys` (for all `Ys`).
- `append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs)` is a rule; it stipulates that concatenating `[X|Xs]` and `Ys` results in `[X|Zs]`, *provided that* concatenating `Xs` and `Ys` results in `Zs` (for all `X`, `Xs`, `Ys`, `Zs`).

Hence, `:-` can be interpreted as an implication from right to left $\Leftarrow$.

Every clause is terminated by a period (’.’).

Queries

A Prolog query is a predicate that may contain variables as parameters.

The Prolog interpreter will try to find an assignment of the variables that renders the predicate true.

For example, the call to `append(List(1), List(2, 3))` in Scala would be modeled by

```
append([1], [2, 3], X)
```

in Prolog. This would produce the answer `X = [1, 2, 3]`.

But it is also possible to put variables in other locations.

For example,

- `append(X, [2, 3], [1, 2, 3])` returns `X = [1]`.
- `append([1, 2], Y, [1, 2, 3])` returns `Y = [3]`.

Queries (cont)

- `append(X, Y, [1, 2, 3])` returns multiple answers:
  - `X = []`, `Y = [1, 2, 3]`, or
  - `X = [1]`, `Y = [2, 3]`, or
  - `X = [1, 2]`, `Y = [2]`, or
  - `X = [1, 2, 3]`, `Y = []`.
- `append([1], Y, Z)` returns a solution schema containing one variable: `Y = X`, `Z = [1|X]`.

This strategy, when it works, can be very flexible.

It is very similar to database query languages.

In fact, Prolog is often used as a language for retrieving information from a database, especially when deductive reasoning is required.

Deductive information retrieval

Here is a small database representing a family tree:

```
female(mary).
female(ann).
female(elaine).
female(jane).
female(sue).
female(jessica).
```

```
married(fred, mary).
married(peter, elaine).
married(tom, sue).
married(alfred, ann).
```

```
child(bob, fred).
child(bob, mary).
child(peter, fred).
child(peter, mary).
```

```
child(sue, fred).
child(sue, mary).
child(jane, sue).
child(jane, tom).
```

```
child(jessica, ann).
child(jessica, alfred).
child(paul, jerry).
child(paul, jane).
```

Deductive information retrieval (cont)

We can access the information contained in the database through a query.

A query is a predicate followed by a question mark.

Here is the transcript of a session with a small Prolog interpreter written in Scala:

```
prolog> child(bob, fred)?
yes
prolog> child(bob, bob)?
no
prolog> child(bob, X)?
[X = fred]
...
```

Deductive information retrieval (cont)

```
...
prolog> more
[X = mary]
prolog> more
no
prolog> child(X, bob)?
no
```

The special query `more` requests additional solutions to the previous query.

Deductive information retrieval (cont)

We can also define rules to derive facts that are not directly encoded in the database. For example:

```
prolog> sibling(X, Y) :- child(X, Z), child(Y, Z).
```

results in

```
prolog> sibling(peter, bob)?
yes
prolog> sibling(bob, jane)?
no
prolog> sibling(bob, X)?
[X = peter]
prolog> more
[X = sue]
...
```

Deductive information retrieval (cont)

```
...
prolog> more
[X = bob]
prolog> more
[X = peter]
prolog> more
[X = sue]
prolog> more
no
```

Deductive information retrieval (cont)

Question: Why does every sibling appear twice in the solutions?

The previous request does not return what expected: `bob` appears as his own brother.

We can correct this by defining

```
prolog> sibling(X, Y) :- child(X, Z), child(Y, Z), not(same(X, Y)).
```

Here, the predicate `same` is simply defined as

```
same(X, X).
```

The `not` operator is special (and somewhat problematic!) in Prolog.

`not(P)` succeeds if the original predicate `P` fails.

For example, to define that a person is male, we can use

```
male(X) :- not(female(X)).
```

Recursive rules

Rules can also be recursive.

For example, to define that `X` is an ancestor of `Y`:

```
parent(X, Y) :- child(Y, X).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

The capability to define recursive rules distinguishes logic programming from database query languages.

Exercise: Define the predicate "`X` is an uncle of `Y`".

Exercise: Define the predicate "`X` and `Y` are cousins".

Implementation of Prolog

The Prolog interpreter seems to have a sort of “intelligence”.

We are now going to discover what makes this possible.

There are two main ingredients:

- A pattern matching mechanism based on unification.
- A derivation finding mechanism.

Representing terms

We can represent Prolog terms using a `Term` class with two sub-classes, `Var` for variables and `Constr` for constructors.

```
trait Term {
  def freevars: List[String] = ...
  def map(s: Subst): Term = ...
}
case class Var(a: String) extends Term
case class Constr(a: String, ts: List[Term]) extends Term
```

For example, the variable `X` is represented by

```
Var("X")
```

And the term `cons(X, nil)` is represented by

```
Constr("cons", List(Var("X"), Constr("nil", List())))
```

Representing terms (cont)

Prolog also has syntactic sugar for terms representing lists that can be translated as follows:

```
[]              =  nil
[S|T]           =  cons(S, T)
[S]             =  cons(S, nil)
[T_1, ..., T_n] =  cons(T_1, ... cons(T_n, nil) ... )
```

The class `Term` defines two methods:

- `freevars` returns a list of all the names of the type variables of the term.
- `map` applies a substitution to the term (see below).

Simple pattern matching

When given a query such as `child(peter, X)?`, the interpreter tries to find a fact in the database that *matches* the query.

Matching means assigning terms to the variables of the query in such a way that the query and the fact become identical.

In our example, `[X = fred]` or `[X = mary]` would be possible assignments since `child(peter, fred)` and `child(peter, mary)` are facts in the database.

Variable assignments (or *substitutions*) are represented by lists of bindings.

Each binding associates a variable name with a term:

```
type Subst = List[Binding]
case class Binding(name: String, term: Term)
```

Simple pattern matching (cont)

We can define a `lookup` function that searches for a substitution for a binding involving a given name:

```
def lookup(s: Subst, name: String): Option[Term] = s match {
  case List()  => None
  case b :: s1 => if (name == b.name) Some(b.term)
                  else lookup(s1, name)
}
```

Substitutions as functions

The function `map` applies a substitution to a term.

It is defined as follows:

```
class Term {
  def map(s: Subst): Term = this match {
    case Var(a) => lookup(s, a) match {
      case Some(b) => b map s
      case None => this
    }
    case Constr(a, ts) => Constr(a, ts map (t => t map s))
  }
  ...
}
```

Substitutions as functions (cont)

In other words, we can interpret substitutions as idempotent functions ($\forall t,\ \sigma(\sigma(t)) = \sigma(t)$) on terms, which behave as the identity function for all but a finite number of variables.

Functions for pattern matching

We are now ready to implement the pattern matching algorithm using two `pmatch` functions.

Here is the first of them:

```
def pmatch(pattern: Term, term: Term, s: Subst): Option[Subst] =
  (pattern, term) match {
    case (Var(a), _) => lookup(s, a) match {
      case Some(term1) => pmatch(term1, term, s)
      case None => Some(Binding(a, term) :: s)
    }
    case (Constr(a, ps), Constr(b, ts)) if a == b =>
      pmatch(ps, ts, s)
    case _ => None
  }
```

Functions for pattern matching (cont)

Explanations: The function `pmatch` takes three arguments:

- a pattern (i.e. a term containing variables),
- a term (which must not contain variables itself), and
- a substitution representing an assignment of variables, the terms of which have already been fixed.

If a match is found, the function returns a result of the form `Some(s)` where `s` is a substitution.

If no match is found, it returns the constant `None` as a result.

The matching algorithm works by pattern matching pairs of patterns and terms:

- If the pattern is a variable, we first have to check that the variable has not been assigned already.

Functions for pattern matching (cont)

- If this is the case, we continue by matching the term that was assigned to the variable.
- Otherwise, we extend the substitution with a new binding that associates the name of the variable with the term.
- If the pattern and the term both have the same constructor as their head, we continue by recursively matching their elements, using the second matching function.

Functions for pattern matching (cont)

Exercise: Implement the second pattern matching function, which has the following signature:

```
def pmatch(patterns: List[Term], terms: List[Term], s: Subst)
    : Option[Subst]
```

This function must return `Some(s1)`, where `s1` is a substitution extending `s` that matches the patterns in `patterns` to the corresponding terms in `terms`.

It must return `None` if no such substitution exists or if the two lists are of different lengths.

Unification

The pattern matching algorithm works well for extracting facts, but fails to work on rules.

In fact, the left-hand side (or head) of a rule can itself contain variables.

In order to match a rule, we need to assign variables in the head of the rule and in the query at the same time.

For example, given the rule

```
sibling(X, Y) :- child(X, Z), child(Y, Z), not(same(X, Y)).
```

and the query `sibling(peter, Z)?`, we need to match `sibling(X, Y)` to `sibling(peter, Z)`, which leads to either one of the assignments

`[X = peter, Y = Z]` or `[X = peter, Z = Y]`.

Unification (cont)

The pattern matching algorithm must be generalized in order to make it symmetric.

The resulting algorithm is called *unification*.

To unify two terms `x` and `y` means to find a substitution `s` such that `x map s` and `y map s` are equal.

Example: Here are some examples of unifications.

```
unify(sibling(peter, Z), sibling(X, Y)) = [X = peter, Z = Y]
unify(same(X, X), same(mary, Y))        = [X = mary, Y = mary]
unify(cons(X, nil), cons(X, Y))         = [Y = nil]
unify(cons(X, nil), cons(X, a))         = <failure>
unify(X, cons(1, X))                    = <failure>
```

The last case is rather subtle: here unification fails because there is no finite term `T` such that `T = cons(1, T)`.

Unification (cont)

However, there is an *infinte* term that satisfies the equation, namely the term representing an infinite list of 1's.

Normal Prolog interpreters only compute finite terms composed of variables and constructors (called *Herbrand terms*), after the logician Jacques Herbrand (1908–1931)).

Implementation of unify

```
def unify(x: Term, y: Term, s: Subst): Option[Subst] = (x, y) match {
  case (Var(a), Var(b)) if a == b =>
    Some(s)
  case (Var(a), _) => lookup(s, a) match {
    case Some(x1) => unify(x1, y, s)
    case None => if ((y map s).freevars contains a) None
                 else Some(Binding(a, y) :: s)
  }
  case (_, Var(b)) =>
    unify(y, x, s)
  case (Constr(a, xs), Constr(b, ys)) if a == b =>
    unify(xs, ys, s)
  case _ => None
}
```

Implementation of `unify` (cont)

Like for pattern matching, we implement an *incremental* version of `unify`, where an intermediate substitution is passed as a third parameter.

The main changes with respect to pattern matching are the following:

- We now explicitly handle the case where the two sides are the same variable. In that case, the unification succeeds with the given substitution.
- The case where one side is a variable has been duplicated to make the procedure symmetric.
- We now verify that a variable does not appear in the term to which it is bound, in order to avoid infinite terms.
  (This is often called *occurrence test*).

Complexity of unification

Without the occurrence test, the complexity of unification is linear in the size of the two terms to be unified.

This may seem surprising as the size of the result of a unification can be exponential in the size of the terms!

Example: Unifying

```
seq(X1, b(X2, X2), X2, d(X3, X3))
seq(a(Y1, Y1), Y1, c(Y2, Y2), Y2)
```

results in

```
seq(a(b(c(d(X3, X3), d(X3, X3)), c(d(X3, X3), d(X3, X3))),
      b(c(d(X3, X3), d(X3, X3)), c(d(X3, X3), d(X3, X3)))),
    b(c(d(X3, X3), d(X3, X3)), c(d(X3, X3), d(X3, X3))),
    c(d(X3, X3), d(X3, X3)),
    d(X3, X3))
```

Complexity of unification (cont)

It is easy to see that, when we extend the sequence, the first term of the unifier grows exponentially with the size of the sequence.

However, the unification remains linear, because it shares trees rather than copying them.

Complexity of unification (cont)

Complexity of the occurrence test

With the occurrence test as it is implemented now, the unification becomes worst-case exponential in the size of its input, because sub-trees may be traversed multiple times.

By marking sub-trees that have already been visited, we can speed up the occurrence test to make it linear in the size of the term graph.

Thus, unification the becomes a quadratic algorithm.

It is possible to do even better, and make unification an $O(n \log n)$ algorithm.

Backtracking

The Prolog interpreter recursively compares a part of the query (the *goal*) with a clause of the program.

- If the query is empty, the interpreter has succeeded.
- If the query is not empty, the interpreter tries to match the first predicate.
  - To match means to unify with the left-hand side of the clause.
  - Clauses are tried in the order in which they are written.
  - If no clause succeeds, the interpreter fails.
  - If a clause matches, the interpreter applies itself recursively on the right-hand side of the clause.
- If the recursive call fails, the interpreter continues with the next clause, otherwise it succeeds.
- If the interpreter succeeds, it continues with the next predicate in the query.

Replacing a failure with a list of successes

How can we formalize the previously described strategy?

In general, how can we express backtracking in a functional language?

Idea: instead of representing *failures*, we construct the list of all *successes* (the possible solutions).

Thus,

- a failure is represented as an empty list of solutions.
- a conjunctive search becomes an intersection of lists.
- a disjunctive search becomes a concatenation of lists.

To guarantee termination and for efficiency reasons, the lists of solutions must be constructed lazily, upon request of a new solution.

Hence, we will model such lists using streams.

Search example

Assume that we are given a graph in which every node has a `successors` field that contains a list of the successors of the node in the graph.

```
class Node {
  val successors: List[Node]
  ...
}
```

We will assume that the graph is acyclic.

The goal is to find a path between two given nodes, or fail if there is no such path.

The idea is that, instead of returning a path or failing, we always return a list, that of all possible paths between the two given nodes.

Search example (cont)

This is achieved through the following function:

```
def paths(x: Node, y: Node): Stream[List[Node]] =
  if (x == y)
    Stream.cons(List(x), Stream.empty)
  else
    for { z <- x.successors.toStream
          p <- paths(z, y) } yield x :: p
```

or, without using `for` comprehensions:

```
def paths(x: Node, y: Node): Stream[List[Node]] =
  if (x == y)
    Stream.cons(List(x), Stream.empty)
  else
    x.successors.toStream flatMap (z => paths(z, y) map (p => x :: p))
```

Search example (cont)

Note that the concatenation of the different solutions is achieved through the use of the `flatMap` function.

Note also, that we need to convert the list of successors to a stream, in order to ensure the lazy generation of the solutions.

The interpreter function

We now come back to the main interpreter function.

Its task is to answer a given `query`, given a program consisting in a list of `clauses`.

The solutions are represented by a stream of substitutions.

Interpretation: For each substitution `s` in the stream, the instance of the query obtained from `query map s` can be derived from the `clauses`.

```
def solve(query: List[Term], clauses: List[Clause]): Stream[Subst] = {
  def solve1(query: List[Term], s: Subst): Stream[Subst] = { ... }
  solve1(query, List())
}
```

Recall that `type Subst = List[Binding]`.

The heart of the interpreter

The implementation of the function `solve` is based on the nested function `solve1`, that takes two parameters:

- The list of predicates `query` that remain to be solved.
- The current substitution `s`, which needs to be applied to all the terms in the query and the clauses.

Here is its implementation:

```
def solve1(query: List[Term], s: Subst): Stream[Subst] = query match {
  case List() =>
    Stream.cons(s, Stream.empty)
  case q :: query1 =>
    for { clause <- clauses.toStream
          s1 <- tryClause(clause.newInstance, q, s);
          s2 <- solve1(query1, s1) } yield s2
}
```

The heart of the interpreter (cont)

The `for` comprehension expresses that, in order to solve a query `q :: query1`,

- we try to match all clauses (in order);
- for each clause, we try to solve the first predicate `q` of the query, by using a new instance of the clause;
- for each success in the previous step, we continue by solving the remainder `query1` of the query.

The function `tryClause` is defined as follows:

```
def tryClause(c: Clause, q: Term, s: Subst): Stream[Subst] =
  unify(q, c.lhs, s) match {
    case Some(s1) => solve1(c.rhs, s1)
    case None => Stream.empty
  }
}
```

The heart of the interpreter (cont)

In other words,

- if the predicate `q` unifies with the head of the clause, we continue by solving the right-hand side of the clause,
- otherwise we fail.

Creating new instances

A clause can be used several times in a derivation

For example, here is a Prolog program that finds all the paths in a graph:

```
successor(a, b).
successor(b, c).
path(X, X).
path(X, Y) :- successor(X, Z), path(Z, Y).
```

In order to construct the path between `a` and `c`, we need to match the clause `successor(X, Z)` to the two axioms `successor(a, b)` and `successor(b, c)`.

This is why `solve1` needs to create a *fresh instance* of a clause before using it in a derivation.

Class `Clause`

This leads to the following implementation of the class `Clause`.

```
case class Clause(lhs: Term, rhs: List[Term]) {
  def freevars =
    (lhs.freevars ::: (rhs flatMap (t => t.freevars))).distinct
  def newInstance = {
    var s: Subst = List()
    for (a <- freevars) { s = Binding(a, newVar(a)) :: s }
    Clause(lhs map s, rhs map (t => t map s))
  }
}
```

Handling negation

The `not` constructor needs special handling:

`not(P)` succeeds if and only if `P` fails.

This is expressed by the following clause in `solve1`:

```
def solve1(query: List[Term], s: Subst): Stream[Subst] = query match {
  ...
  case Constr("not", qs) :: query1 =>
    if (solve1(qs, s).isEmpty) solve1(query1, s)
    else Stream.empty
  ...
}
```

Is logic programming logical?

We can consider Prolog as an approximation of programming with mathematical logic.

That is, given a query `q`, we would like the interpreter to return a substitution `s`, such that `q map s` is a logical consequence of the program.

The program itself can be seen as a set of axioms and rules, where `:-` denotes inverse implication $\Leftarrow$.

When it works, this gives us a well-founded and very flexible notation, in which we can express programs and queries.

Furthermore, every engineer knows (or should know) logic well enough to make the paradigm of logic programming easy to learn.

Incorrectness and incompleteness of logic programming

However, the execution of the interpreter only gives us an approximation of this ideal.

It has several flaws:

- Sometimes the interpreter returns an answer that is not a solution.
- Sometimes the interpreter fails to find an existing solution.

The first limitation is called *incorrectness*, the second *incompleteness*.

A problem related to incorrectness

Most Prolog interpreters are not correct because they omit the occurrence test during unification, and hence find false solutions.

Example: In order to test whether two terms are equal, we use the axiom

```
same(X, X).
```

Now, in order to find all numbers that are equal to their successors, we use the rule

```
strangeNum(X) :- same(X, succ(X)).
```

The execution of the test `strangeNum` with a concrete number always fails.

A problem related to incorrectness (cont)

However, if we want to check whether such strange numbers exists using a standard Prolog interpreter, we usually get a solution:

```
prolog> strangeNum(X)?
[X = succ(X)]
```

This problem can easily be overcome by including an occurrence test in the unification function (which we did in our Prolog interpreter written in Scala).

Incompleteness

It may happen that a solution to a query exists, but the interpreter fails to find it.

Example: Let's revisit our "path-finding" program. This time, we introduce a cycle into the graph:

```
successor(a, b).
successor(b, a).
successor(b, c).
path(X, X).
path(X, Y) :- successor(X, Z), path(Z, Y).
```

Now, the query

```
prolog> path(a, c)
```

results in a stack overflow instead of returning a solution.

Incompleteness (cont)

By examining the interpreter function `solve1`, can you tell what went wrong?

Problems with depth-first search

The interpreter uses a depth-first search strategy in order to find a solution.

In other words, when the head of a clause matches, the interpreter immediately tries to prove the precondition of the clause.

In our case, this leads to the following sequence of goals to prove:

```
path(a, c)
successor(a, b), path(b, c)
path(b, c)
successor(b, a), path(a, c)
path(a, c)
...
```

... and so on until the stack overflows.

Problems with negation

The handling of `not` may also causes problems, with respect to both correctness and completeness.

A `not(P)` predicate is considered true in Prolog if `P` can not be proven.

This is called the *closed world assumption*: everything that is not derivable is false.

For queries without variables, this seems a reasonable hypothesis.

But for queries with variables, it can lead to strange results.

Problems with negation (cont)

Example: Consider this simple dietary program:

```
vegetable(bean).
vegetable(tomato).
food(hamburger).
junkFood(X) :- not(vegetable(X)).
healthy(X) :- not(junkFood(X)).
```

Clearly, this implies that hamburgers constitute a poor diet.

```
prolog> junkFood(hamburger)?
yes
```

Problems with negation (cont)

On the other hand, if we ask for an arbitrary piece of junk food `X`, we obtain

```
prolog> junkFood(X)?
no
```

The problem remains even if we restrict the search space:

```
prolog> junkFood(X), same(X, hamburger)?
no
```

This example illustrates a problem with respect to completeness: the interpreter fails to find a particular solution, even though it is a logical consequence of the program.

Problems with negation (cont)

By using an additional `not`, we can transform the lack of completeness into a lack of correctness.

On one hand we obtain

```
prolog> healthy(hamburger)?
no
```

But by reformulating the query, we can obtain the "solution"

```
prolog> healthy(X), same(X, hamburger)?
[X = hamburger]
```

Since the fact that `hamburger` is good for your health is not a logical consequence of the program, we have a case of incorrectness.