Week 14: Logic Programming (2)

We have seen that logic programming searches for solutions to queries, given a program consisting of facts and rules.

An interpreter for logic programming tries to apply rules to solve a query, using unification as the basic inference step.

We now take a look how this search is done.

Search by Backtracking

The Prolog interpreter will recursively match a part of the query (the *goal*) against a clause of the program.

- Given an empty query, the interpreter succeeds.
- Given a non-empty query, the interpreter tries to match the first predicate.
  - Matching means: Unify with a left-hand side of a clause.
  - Clauses are tried in the sequence they are written.
  - If no clause matches, fail.
  - If a clause matches, recursively invoke the interpreter on the clause's right hand side.
- If the recursive invocation fails, continue with next clause, otherwise succeed.
- If interpreter succeeds, continue with the next predicate in the clause.

Replacing Failure by a List of Successes

How can the previously described strategy be formalized?

Generally, how can a search with backtracking be expressed in a functional language?

Idea: Instead of representing *failure*, we model a list of all *successes*, i.e. possible solutions.

Then,

- failure is represented by an empty list of solutions.
- and-search becomes list intersection.
- or-search becomes list concatenation.

To ensure termination and for efficiency it is mandatory that the lists of solutions are constructed lazily, as new solutions are needed.

Hence, these lists will be modelled as streams.

Search Example

Let's say we are given a graph of nodes, where each node has a *successors* field which contains the list of its successors in the graph.

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

We assume that the graph is acyclic.

The task is to search a path between two given nodes, or to fail if none exists.

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

This is achieved by the function:

```
def paths(x: Node, y: Node): Stream[List[Node]] =
  if (x == y) [x]
  else
    for ( val z ← list2stream(x.successors);
          val p ← paths(z, y)) yield x :: p;
```

or, without the ***for*** notation:

```
def paths(x: Node, y: Node): Stream[List[Node]] =
  if (x == y) [x]
  else
    list2stream(x.successors) flatMap (z ⇒ paths(z, y) map (p ⇒ x :: p));
```

Note the concatenation of the different solutions, which is expressed by the *flatMap* application.

Note also that we need to convert the list of successors to a stream, in order to ensure that solutions are generated lazily.

The *list2stream* function is defined as follows.

$$
\begin{array}{l}
\textbf{\textit{def}}\ \mathit{list2stream}\,[\mathrm{a}]\,(\mathit{xs}\colon \mathit{List}\,[\mathrm{a}]\,)\colon \mathit{Stream}\,[\mathrm{a}\,] = \mathit{xs}\ \mathit{match}\ \{ \\
\quad \textbf{\textit{case}}\ [\,] \Rightarrow \mathit{EmptyStream} \\
\quad \textbf{\textit{case}}\ x :: \mathit{xs1} \Rightarrow \mathit{ConsStream}\,(x,\ \mathit{list2stream}\,(\mathit{xs1}\,)\,) \\
\}
\end{array}
$$

The Interpreter Function

We now return to the main interpreter function.

It's task is to solve a given query *query*, given a program consisting of a list *clauses* of clauses.

The solution(s) take the form of a stream of substitutions.

Interpretation: For every substitution s in the stream, the instance of the query given by *query map s* can be derived from *clauses*.

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

The Core of the Interpreter

The implementation of function *solve* is based on a nested function *solve1*, which takes two parameters:

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

Here is its implementation.

```
def solve1(query: List[Term], s: Subst): Stream[Subst] = query.match {
  case [] =>
    consStream(s, EmptyStream)
  case q :: query1 =>
    for (val clause <- list2stream(clauses);
             val s1 <- tryClause(clause.newInstance, q, s);
             val s2 <- solve1(query1, s1)) yield s2
}
```

The ***for***-expression expresses that, to solve a query $q :: query1$:

- We try to match all clauses (in the order they are written).
- For each clause, try to solve the first predicate q of the query using a new instance of the clause.
- For each success of the previous step, continue to solve the rest of the query $query1$.

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 ⇒ EmptyStream
  }
}
```

That is,

- if the goal predicate unifies with the head of the clause, continue solving the right-hand side of the clause,
- otherwise fail.

Creating New Instances

A clause can be used many times in a derivation.

For instance, here is a Prolog program to search for paths in a graph.

$successor(a, b).$
$successor(b, c).$
$path(x, x).$
$path(x, y) :- successor(x, z), path(z, y).$

To construct the path between a and c, we need to instantiate the $successor$ axiom to $successor(a, b)$ and $successor(b, c)$.

That's why *solve1* needs to create a *fresh instance* of a clause before using it in a derivation.

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

Class *Clause*

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

Handling Not

The *not* constructor needs to be handled specially.

not (P) succeeds iff P fails.

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

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

Is Logic Programming Logical?

Prolog can be seen as an approximation to programming with mathematical logic.

That is, given a query q, we would like the interpreter to return with 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 reverse implication $\Leftarrow$.

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

Furthermore, every engineer knows logic (or should know it), so the programming paradigm is easy to learn.

Unsoundness and Incompleteness of Logic Programming

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

The connection breaks down in several respects:

- Sometimes the interpreter gives an answer which is not a solution.
- Sometimes the interpreter fails to find a solution even though one exists.

The first shortcoming is called *unsoundness*, the second *incompleteness*.

An Unsoundness Problem

Most Prolog interpreters are unsound, because they omit the occurs check in unification, hence non-solutions can be found.

Example: To test whether two terms are the same, we use the axiom:

$$\mathit{same}(X, X).$$

Now, to characterize all numbers that are equal to their successors:

$$\mathit{strangeNum}(X) \;:-\; \mathit{same}(X, \mathit{succ}(X)).$$

Running the *strangeNum* test with a concrete number always fails.

However, if we want to find out whether there are strange numbers in a standard Prolog interpreter, we usually get a solution:

$$\mathit{prolog}{>}\ ?\ \mathit{strangeNum}(X).$$
$$[X = \mathit{succ}(X)].$$

This problem can be overcome easily by including an occurs check in the unification function (which is what we did in the Prolog interpreter written in Scala).

Incompleteness

It can happen that solutions to a query exist, yet the interpreter fails to find them.

Example: Taking our "find path" program, let's introduce a cycle in the graph.

$$
\begin{array}{l}
\mathit{successor}\,(a,\ b\,). \\
\mathit{successor}\,(b,\ a\,). \\
\mathit{successor}\,(b,\ c\,). \\
\mathit{path}\,(X,\ X\,). \\
\mathit{path}\,(X,\ Y\,) \ {:}{-}\ \mathit{successor}\,(X,\ Z\,),\ \mathit{path}\,(Z,\ Y\,).
\end{array}
$$

Now, the query

$$\mathit{prolog}{>}\ \mathit{path}\,(a,\ c\,)$$

leads to a stack overflow instead of a solution.

Considering the interpreter function *solve1*, can you tell what went wrong?

The Problem with Depth First Search

The interpreter applies depth-first search to find a solution.

That is, when a clause head matches, the interpreter immediately tries to prove the clause's precondition.

In our case, we get the following chain of goals to prove:

$$\begin{array}{l} path\,(a,\ c\,) \\ successor\,(a,\ b\,),\ path\,(b,\ c\,) \\ path\,(b,\ c\,) \\ successor\,(b,\ a\,),\ path\,(a,\ c\,) \\ path\,(a,\ c\,) \\ \ldots \end{array}$$

...and so on until stack overflow.

Problems with *not*

The handling of *not* also causes problems, both in relation to soundness and to completeness.

A predicate $not(P)$ is treated as true in Prolog if its negation cannot be proven.

This is also called the *closed world assumption*: Everything which is not derivable is assumed to be false.

For queries without variables, this can be a reasonable assumption.

But for queries with variables, the outcome is sometimes strange.

Example: Consider the following little Prolog program about food-stuffs:

$$\begin{array}{l} vegetable(bean). \\ vegetable(tomato). \\ food(hamburger). \\ junkFood(X) :- not(vegetable(X)). \\ healthy(X) :- not(junkFood(X)). \end{array}$$

Clearly, this implies that hamburgers are junk food:

$$\begin{array}{l} prolog> ?junkFood(hamburger). \\ yes \end{array}$$

On the other hand, if we ask for any X which is junk food, we get:

$$\begin{array}{l} prolog> ?junkFood(X). \\ no \end{array}$$

This holds even if we narrow the search:

$$\begin{array}{l} prolog> ?junkFood(X), same(X, hamburger). \\ no \end{array}$$

The example shows a problem with completeness: The interpreter failed to find a solution which was a logical consequence of the program.

By using an additional *not* we can turn the loss of completeness into a loss of soundness.

On the one hand, we get:

> *prolog> ?healthy(hamburger).*
> *no*

But by reformulating the query, we get a "solution":

> *prolog> ?healthy(X), same(X, hamburger).*
> $[X = \mathit{hamburger}]$

Since *hamburger*'s being healthy is not a logical consequence of the program, we have a case of unsoundness.

Summary

In the last 14 weeks, we have encountered wide range of approaches to programming.

- Functional – programming by composing functions.
- Imperative – programming by affecting mutable state.
- Logic – programming by searching for consequences of facts and rules.

Orthogonal to these three paradigms are:

- Type systems
  - Static typing (e.g. Scala) vs. dynamic typing (e.g. Lisp, Prolog)
  - Forms of polymorphism: generics, subtyping.
- Object-Orientation
  - Classes and objects
  - Inheritance

- Evaluation orders
    - Call-by-value vs. call-by-name.
    - Lists, streams, and iterators as abstractions of sequences.

This is quite a rich tool-box of concepts and techniques.

It opens up the possibility to program on a higher-level, and in particular to design higher-level libraries.

Throughout everything, two principal concepts emerged:

- Abstraction: The ability to isolate a concept, name it, and use only the name afterwards while hiding the implementation details.
- Composition: Focus on combinators to build new abstractions rather than on just the primitives themselves. This gives increased power (and responsibility) to users of your abstractions.

So, as a competent programmer, look for abstractions that compose!

I hope you found the course interesting.