COMP304 Tutorial 09
23/09/2016

1 Finite State Machines

We'll cover some examples from the lecture in finer details. Recall that a finite state machine (FSM) is a set of states and transitions. The system is in one state at any given time, and connected transitions tell you to move into another state when you see an event happen. This can be modelled as a graph. For example:

```
start(1).
finish(3).

transition(1, a, 2).
transition(1, b, 2).
transition(1, c, 3).
transition(2, c, 3).
```

The start are nodes and the transitions are labelled edges. `transition`$(1, \texttt{a}, 2)$ means: "if you are in state 1 and you see event `a` happen, move to state 2". If you have a language $\mathcal{L}$ of events, the string σ may be seen as a sequence of events. A finite-state machine may compute σ if there is a path of events matching σ from the start state to the final state.

A string (or a sequence of events) is accepted by an FSM if there is an accepting path from a start-state, along which those events happen.

```
accept(String) :-
    start(State),
    acceptingPath(State, String).
```

If there are no more events to be accepted, and you're in a finishing state, then this is an accepting path.

```
acceptingPath(State, []) :- finish(State).
```

Otherwise if you're in `State1` and you can take a transition from the next event in sequence to `State2`, and there's an accepting path from `State2` covering the rest of the events, this is an accepting path.

```
acceptingPath(State1, [NextEvent | RestOfEvents]) :-
    transition(State1, NextEvent, State2),
    acceptingPath(State2, RestOfEvents).
```

We may then ask Prolog for paths which are accepting by posing a query like this one:

```
?- accept(X).
X = [a, c] ;
```

```
3 X = [b, c] ;
4 X = [c] ;
5 false.
```

The order in which we defined transitions is significant. If transition t_1 is specified before transition t_2, Prolog will try transition t_1 first.

We might like to gather all solutions to a predicate, and SWI Prolog's `findall/3` lets us do that. Here is a predicate which makes use of `findall/3` to collect the set of strings accepted by our FSM (its language).

```
1 language(L) :-
2     findall(String, accept(String), L).
```

If it helps, you could think of the second argument as being "higher-order", in that it takes a predicate (a bit like how in Haskell you could pass functions around as arguments).

2 Musings On Computability

There's something kind of interesting in what we've done. If you think of the rules of an FSM as a model of computation, then an instance of an FSM is like a computer program. The predicate we wrote to figure out the language is essentially telling us every output this computer program can do. Can we do the same thing for a program written in Java or Haskell?

The answer is: no (Rice's theorem). Finite state machines are a weaker model of computation than λ functions or Turing/register machines, which are the foundations for functional and imperative languages. They are not Turing-complete, meaning there are programs you can write in Java and Haskell, which you cannot write as a finite-state machine.[1]

Although FSMs are weaker than Turing machines, that doesn't mean "programs" in FSMs are necessarily less useful. Sometimes you don't need the full computational power of a Turing machine for a certain program, so you can get away with an FSM. For example, a controller for some big system (e.g. traffic lights or elevator). Regular expressions are another one: a regular expression has a corresponding FSM which accepts exactly the same set of strings (language).

Because FSMs are weaker you can reason about them better. We just showed how you can look at the outputs of a program. If you have a more interesting FSM, e.g. an elevator, you can ask more interesting questions like: "is there a sequence of events where the elevator ends up stranded and not able to move". You may have used LTSA in SWEN224 last year, which basically uses FSMs to reason about concurrent systems.

[1] Some argue this isn't true, because computers have a finite amount of memory (while Turing machines don't) and can thus be considered as enormously FSMs; but that's kind of an implementation detail.

3 Non-Determinism

The FSM we wrote is deterministic, meaning in any given state if an event happens there is only one transition that may be taken. If we relax this requirement we get non-deterministic state machines. Here's an example:

```
start(1).
finish(3).

transition(1, a, 2).
transition(1, a, 3).
transition(1, b, 2).
transition(1, c, 3).
transition(2, c, 3).
```

A non-deterministic state machine accepts a string if any path consumes every token/event in the string. We may again ask what the language of this machine is.

```
?- language(L).
L = [[a, c], [a], [b, c], [c]].
```

The way Prolog works is that it may take either path when searching for a solution. It will try one path (which is based on which transition comes first in your Prolog file) and then try the other.

If we have a cycle in our graph, the language it accepts would be an infinite set.

```
start(1).
finish(3).

transition(1, a, 2).
transition(1, a, 3).
transition(1, b, 2).
transition(1, c, 3).
transition(2, c, 3).
transition(2, d, 2).
```

If we then ask Prolog for the language it will loop forever until it crashes (because it will be trying to generate the set, which it can't display to you). This means the order of predicates does matter when there are infinitely many distinct (i.e. can't be generally described) solutions involved. This program will crash:

```
?- language(L), print('there's a solution!')
```

Because it attempts to solve `language(L)` by explicitly constructing an answer (compare that with a proof-by-hand, where you could describe the infinite set in finite terms, rather than by listing every element). You can see how Prolog's search-tree is working:

```
?- accept(X).
X = [a, c] ;
X = [a, d, c] ;
X = [a, d, d, c] ;
X = [a, d, d, d, c] ;
X = [a, d, d, d, d, c] ;
X = [a, d, d, d, d, d, c] ;
```

You might wonder if non-deterministic FSMs are more powerful, in that they may compute things which deterministic FSMs cannot, but they are able to compute the same languages [2] (however, non-deterministic FSMs can compute things in a polynomial number of states for which deterministic FSMs require an exponential number of states).

State machines can't compute everything and Prolog can sometimes loop forever. The main takeaway is that, at the end of the day, we can't rely 100% on computers to tell us we've done something right.

4 Prolog Semantics

We've seen some weird things about Prolog, such as the way in which it interprets the "meaning" of a term. For example, $2 == 1 + 1$ is not true (because those two terms aren't syntactically equivalent), but 2 `is` $1 + 1$ is true (however, $1 + 1$ `is` 2 is not true).

Something is "true" in Prolog if it can prove that it exists from its knowledge-base. Otherwise, it is false. Negation works a bit weird on this definition. Take the following:

```
1 bird(kiwi).
2 bird(albatross).
3 flightless(kiwi).
```

If we want to negate something we may use the `not/1` predicate. To ask if an albatross is not flightless, we might try this:

```
1 ?- not(flightless(albatross)).
2 true.
```

But if we left the argument to flightless as an unbound variable, this is how Prolog responds:

```
1 ?- not(flightless(X)).
2 false.
```

Does this seem weird? It did to me at first. Recall truth in Prolog: something is false if you cannot prove it true. But Prolog can prove `flightless(X)` true, by binding $X =$ `Kiwi`. You might try to read `not(flightless(X)` as "give me an X for which `not(flightless(X))` is true", but what it really means in Prolog is "show that no X is flightless".

There's a reason for this. If you think about the constructive nature of Prolog, when you ask it to find a predicate $P(X)$ you may think of it as solving the logical equation $\exists X \mid P(X)$. The negation of this is $\neg(\exists X \mid P(X)) \equiv \forall X \mid \neg P(X)$. That means `not(flightless(X))` is true if `flightless(X)` is false, for all X. The mistake comes from thinking that Prolog will interpret it as $\exists X \mid (\neg P(X))$.

[2] Rabin & Scott (1958).