Constructive Logic (15-317), Fall 2016
Recitation 12: Forward Logic Programming

Evan Cavallo (ecavallo@cs), Oliver Daids (ojd@andrew), Giselle Reis (greis@andrew)

Functional Evaluation with Forward Chaining

Consider the language of the untyped lambda calculus.

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

We can write a set of rules using three predicates

$\mathsf{eval}(e)$	evaluate e
$e \mapsto^* e'$	e reduces to e'
$e \hookrightarrow v$	e evaluates to v

so that we can evaluate e with forward chaining, by seeding the system with $\mathsf{eval}(e)$ and waiting for a fact of the form $e \hookrightarrow v$ to appear.

Task 1. Define such a set of rules.

Solution 1: See `http://www.cs.cmu.edu/~fp/courses/lp/lectures/20-bottomup.pdf`.

Implementing Forward Chaining in Prolog

Task 2. Define a predicate `forward/2` so that `forward(I,O)` takes an input list of facts `I` and returns an output list `O` of facts obtained by exhaustively applying inference rules to `I` until quiescence. Assume the existence of a predicate `fclause/2` which enumerates the set of rules by axioms `fclause(G,S)` where `G` is the conclusion and `S` is the list of predicates.

Solution 2:

```
forward(I,O) :- fclause(G,S), sublist(S,I), \+(member(G,I)), !, forward([G|I], O).
forward(I,I).
```

Task 3. Define `fclause/2` so that `forward` computes the symmetric transitive closure of an input graph.

Solution 3:

```
fclause(edge(N,M), [edge(M,N)]).
fclause(edge(M,P), [edge(M,N),edge(N,P)]).
```

Task 4. How can we use this (or something like it) to compute Fibonacci numbers like in yesterday's lecture? There are several possible answers.

Solution 4: Some design choices:

- How to ensure that the input eventually quiesces? One way is to include a timeout parameter in the fibonacci predicate; another is to add a timeout parameter to `forward`.
- How to compute addition? This could be implemented by another forward chaining predicate. It could also be implemented by allowing clauses to have side conditions (adding an extra argument to `fclause`), and checking those side conditions in `forward`. For example, using hardware integers:

```
forward(I,O) :- fclause(G,S,C), sublist(S,I), \+(member(G,I)), check(C), !,
                forward([G|I], O).
forward(I,I).

check(sum(M,N,P)) :- P is M + N.
```