Evan Cavallo (ecavallo@cs), Oliver Daida (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

\[
\begin{align*}
& \text{eval}(e) \quad \text{evaluate } e \\
& e \rightarrow^* e' \quad e \text{ reduces to } e' \\
& e \rightarrow v \quad e \text{ evaluates to } v
\end{align*}
\]

so that we can evaluate \( e \) with forward chaining, by seeding the system with \( \text{eval}(e) \) and waiting for a fact of the form \( e \rightarrow v \) 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](http://www.cs.cmu.edu/~fp/courses/lp/lectures/20-bottomup.pdf)

**Implementing Forward Chaining in Prolog**

**Task 2.** Define a predicate \( \text{forward}/2 \) so that \( \text{forward}(I, O) \) 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 \( \text{fclause}/2 \) which enumerates the set of rules by axioms \( \text{fclause}(G, S) \) where \( G \) is the conclusion and \( S \) is the list of predicates.

**Solution 2:**

\[
\begin{align*}
\text{forward}(I, O) & :\:- \text{fclause}(G, S), \text{sublist}(S, I), \\neg\text{(member}(G, I)), !, \text{forward}([G|I], O). \\
\text{forward}(I, I).
\end{align*}
\]

**Task 3.** Define \( \text{fclause}/2 \) so that \( \text{forward} \) computes the symmetric transitive closure of an input graph.

**Solution 3:**

\[
\begin{align*}
\text{fclause}(\text{edge}(N, M), [\text{edge}(M, N)]). \\
\text{fclause}(\text{edge}(M, P), [\text{edge}(M, N), \text{edge}(N, P)]).
\end{align*}
\]
**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:

```prolog
forward(I,0) :- fclause(G,S,C), sublist(S,I), \+(member(G,I)), check(C), !, forward([G|I], 0).
forward(I,I).
check(sum(M,N,P)) :- P is M + N.
```