Homework 16: λProlog Programming and Operational Semantics

Due:

If you haven't done so already, you might want to read Chapter 14 in Watt's *Programming Languages: Concepts and Paradigms.*

Unless we specifically request, your λProlog code does not have to run "backwards". (But you may want to see which ones can do that.) In the examples you will see "no more solutions" or "yes" printed; your program does not have to have exactly this output.

Files containing the lambda prolog modules in this homework are available in the directory `~leavens/src/lambda-prolog/cs541`.

1 Describing Abstract Values

In this section we'll look at how to use λProlog to specify abstract data types. The idea is much like that in Chapter 6 of Watt's *Programming Languages: Syntax and Semantics*, which you might want to read. Another thing to read for background is the paper we passed out by Rollins and Wing titled "Specifications as Search Keys for Software Libraries" (especially Figure 2).

1. Specify in λProlog a module for homogeneous finite sets. Your module should start as follows.

```
module set.  % homogeneous finite sets.

kind set                type -> type.

% set terms
type emptyset           set T.
type set-insert         set T -> T -> set T.
type singleton          T -> set T.

% set observers
type has                set T -> T -> o.
type not-has            set T -> T -> o.
type set-delete         set T -> T -> set T -> o.
type set-union          set T -> set T -> set T -> o.
type set-intersect      set T -> set T -> set T -> o.
type set-diff           set T -> set T -> set T -> o.
type set-size           set T -> int -> o.
type subseteq           set T -> set T -> o.
type supseteq           set T -> set T -> o.
type proper-subset      set T -> set T -> o.
type proper-supset      set T -> set T -> o.
type set-equal          set T -> set T -> o.
```

You are to specify all the "observer" relations. The meaning of these is intended to be the usual thing for sets. The relation `(has S E)` holds iff `E` is an element of `S`. The relation `(set-delete X E Z)` holds iff `Z` has all the elements of `X` except `E`. The relation `(set-diff X Y Z)` holds iff `Z` has all the elements of X, except those in Y.

2. Specify in λProlog a module for homogeneous finite doubly-ended queues. Your module should start as follows.

```
module deque. % doubly ended queues

kind deq                type -> type.

% deque terms
type emptydeq           deq T.
type deq-precat         deq T -> T -> deq T.
type deq-postcat        deq T -> T -> deq T.

% deq observers
type deq-has            deq T -> T -> o.
type deq-size           deq T -> int -> o.
type deq-head           deq T -> T -> o.
type deq-last           deq T -> T -> o.
type deq-init           deq T -> deq T -> o.
type deq-tail           deq T -> deq T -> o.
type deq-count          deq T -> T -> int -> o.
type deq-isEmpty        deq T -> o.
type deq-equal          deq T -> deq T -> o.
```

You are to specify all the "observer" relations. The meaning of these is intended to be the usual thing for doubly-ended queues. The relation `(deq-has Q E)` holds iff `E` is in the deque `Q`. The relation `(deq-init Q1 Q2)` holds iff `Q2` has all the elements of `X` except the last. The relation `(deq-tail Q1 Q2)` holds iff `Q2` has all the elements of X, except the first.

2 Operational Semantics of Programming Languages

This section explores using λProlog to do a kind of Structural Operational Semantics. Structural Operational Semantics is essentially rewriting, and was first defined by G. Plotkin. The style presented here is called "big step" (C. Gunter), "evaluation" (M. Hennessy), or "natural" semantics. In this style of operational semantics, one describes binary relation (evalsto) between abstract syntax trees and a domain of program results called (generically) "Answers" or "Values". As an extremely simple example, consider the following.

```
module arith.

% Evaluation semantics for the language of arithmetic expressions.
% Adapted from Hennessy: The Semantics of Programming Languages.

kind exp        type.
kind operator   type.

% abstract syntax
type    add     operator.
type    sub     operator.
type    mult    operator.
type    divide  operator.
type    num     int -> exp.
type    op      operator -> exp -> exp -> exp.

% auxilliary semantic relations
```

```
type    ap      operator -> int -> int -> int -> o.

(ap add N1 N2 Ans) :- Ans is N1 + N2.
(ap sub N1 N2 Ans) :- Ans is N1 - N2.
(ap mult N1 N2 Ans) :- Ans is N1 * N2.
(ap divide N1 N2 Ans) :- Ans is N1 div N2.

% the evalsto relation
type    evalsto exp -> int -> o.

(evalsto (num N) N).

(evalsto (op Oper E1 E2) Ans) :-
        (evalsto E1 V1),
        (evalsto E2 V2),
        (ap Oper V1 V2 Ans).
```

Then one can write queries such as the following.

```
?- (evalsto (op divide (num 5) (num 2)) V).
Solving ...
V == 2
yes
?- (evalsto (op add (op mult (num 3) (num 4)) (num 8)) V).
Solving ...
V == 20
yes
```

Division by zero is not really specified by this semantics. It can be treated in two ways.

3. The simplest way to deal with errors is to be sure the evalsto relation fails in erroneous cases. Change the semantics so that it has the following behavior.

   ```
   ?- (evalsto (op divide (num 5) (num 0)) V).
   Solving ...
   no
   ```

4. Another way to treat such errors is by having the answer be either a number or an error. Define a type `answer`, a constant `error` of type `answer`, and a (so-called constructor or injection) function that converts integers into answers. Redo the semantics to work with answers. For example,

   ```
   ?- (evalsto (op divide (num 5) (num 2)) V).
   Solving ...
   V == (inInt 2)
   yes
   ?- (evalsto (op divide (num 5) (num 0)) V).
   Solving ...
   V == error
   yes
   ```

The following problems deal with extending this language and its semantics.

5. Add variables to the abstract syntax. One way to do this is as follows.

   ```
   type variable   string -> exp.
   ```

The type `string` is built-in to λProlog. (One writes a string in double quotes `"a string"`.) Add an abstract data type of (value) environments that represent maps from variables to values, along with operations on these. The type of evalsto should change to be the following

```
type evalsto    env -> exp -> int -> o.
```

Redo the semantics to take these changes into account.

6. Add a `let` expression to the abstract syntax, and specify its semantics. The `let` expression should act like the ML block expression (see Watt's section 4.5.2), except that you need only handle one declaration. That is, the equivalent ML form is `let val` *x* `=` *exp* `in` *exp*.

3 Extra Credit

7. Specify in λProlog a module for homogeneous finite sequences. Is it possible to use the work you did for doubly-ended queues by importing the module `deque`? Why or why not?

The other style of structural operational semantics, is called a "terminal transition system", or a "little step" (C. Gunter), or "computation" (M. Hennessy) semantics. In this style one describes the configurations of an abstract machine (Γ), a binary relation on configurations (reducesto). In the set of configurations there should be a subset of terminal configurations (T), such that if g is in T, then there is no $g' \in \Gamma$ such that (reducesto $g\ g'$) holds. To give the semantics to a programming language, one uses two auxilliary functions:

$$\begin{aligned} \text{input} &: \text{Programs} \rightarrow \Gamma \\ \text{output} &: T \rightarrow \text{Answers} \end{aligned}$$

Often one makes the set Γ equal to the set of programs; in this case one can use the identity function for input. For example, for the simple arithmetic example, one would write the computation semantics as follows.

The following must be specified outside of λProlog.

$$\begin{aligned} \text{Programs} &= \texttt{exp} = \Gamma \\ \text{Answers} &= \texttt{int} \\ T &= \{\texttt{(num } N\texttt{)} \mid N : \texttt{int}\} \end{aligned}$$

The rest is given by the following λProlog module.

```
module comput-arith.

% Computation semantics for the language of arithmetic expressions.
% Adapted from Hennessy: The Semantics of Programming Languages.

kind exp        type.
kind operator   type.

% abstract syntax
type    add     operator.
type    sub     operator.
type    mult    operator.
type    divide  operator.
type    num     int -> exp.
type    op      operator -> exp -> exp -> exp.
```

```
% auxilliary semantic relations
type    output   exp -> int -> o.
type    ap       operator -> int -> int -> int -> o.

(output (num N) N).

(ap add N1 N2 Ans) :- Ans is N1 + N2.
(ap sub N1 N2 Ans) :- Ans is N1 - N2.
(ap mult N1 N2 Ans) :- Ans is N1 * N2.
(ap divide N1 N2 Ans) :- Ans is N1 div N2.

% the reducesto relation
type    reducesto       exp -> exp -> o.

(reducesto (op Oper (num V1) (num V2)) (num N)) :- (ap Oper V1 V2 N).
(reducesto (op Oper E1 E2) (op Oper V1 E2)) :- (reducesto E1 V1).
(reducesto (op Oper E1 E2) (op Oper E1 V2)) :- (reducesto E2 V2).
```

Then one can write queries such as the following.

```
?- (reducesto (op divide (num 5) (num 2)) V), (output V N).
Solving ...
V == num 2
N == 2
yes
```

8. Can you use both of the techniques for handling errors in this semantics?

9. Add environments and a `let` expression to the semantics.

10. When using the computation semantics, you may be surprised by the following.

    ```
    ?- (reducesto (op add (op mult (num 3) (num 4)) (num 8)) V), (output V N).
    Solving ...
    no
    ?- (reducesto (num 3) (num 3)).
    Solving ...
    no
    ```

    What happened here? Is it possible to fix this by using the idea of the reflexive-transitive closure of a relation? Can that be done in a general way using λProlog's higher-order features?

11. Give a computation semantics for the simply-typed λ calculus. The main reduction rules are the beta and eta rules discussed on our handout.

12. Give a computation or evaluation semantics for part of your distributed programming language design. For example, you could model exception handling, or the commit of atomic actions.

The following are more investigations into λProlog's higher-order features.

13. In the module for finite sets, is it possible to specify an inductive principle? That is, suppose we want to know if some property `P` holds for all sets. By a property, we mean a λProlog predicate of type `set T -> o`. To show that `P` holds for all sets, we would show `(P emptyset)` and then assuming that `(P S)` holds, show that for all `E (P (set-insert S E))` holds. Can you write this idea down in λProlog?

14. Specify in λProlog a module for homogeneous infinite sets. Your module should start with as follows.

```
module infinite-set. % homogeneous infinite sets.

kind set                type -> type.

% set terms
type comprehend         (T -> o) -> set T.

% set observers
type has                set T -> T -> o.
type set-union          set T -> set T -> set T -> o.
type set-intersect      set T -> set T -> set T -> o.
type set-diff           set T -> set T -> set T -> o.
```

The only unusual thing here is the functor `comprehend`. The term `(comprehend P)` represents the set of all things that satisfy `P`. For example, assuming that the fact `false` is not in the database, then `(comprehend (x : int \ false))` is the empty set of integers. Another example, assuming that `prime` is a predicate on integers, then `(comprehend (x : int \ prime x))` is the set of all prime integers. Note that since sets like this can be infinite, you won't be able to put all the elements of such a set into a list (or whatever); so don't try! Instead work with terms of the form `(comprehend P)` as your representation.

Can you also program the following relations? (If not, describe the difficulty.)

```
type subseteq           set T -> set T -> o.
type supseteq           set T -> set T -> o.
type proper-subset      set T -> set T -> o.
type proper-supset      set T -> set T -> o.
type set-equal          set T -> set T -> o.
```