CS 538

Final Exam

Wednesday, May 17, 2000
12:25 PM — 2:25 PM
168 Noland

Instructions

Answer any four questions. (If you answer more, only the first four will count.) Point values are as indicated. Please try to make your answers neat and coherent. Remember, if we can't read it, it's wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (a) (13 points)
Complete the following Java class definition. Class `Set` implements a set of objects (of type `Object`). The constructor of no arguments creates an empty set. The constructor with an `Object` array argument creates a set containing the values in the array (with duplicates removed). Function `union` computes a set that is the union of its two sets parameters. Function `intersect` computes a set that is the intersection of its two sets parameters. Function `member` tests whether a given object is a member of a set.

```
class Set {
    Set() { ... }
    Set(Object[] values) { ... }
    static Set union(Set s1, Set s2) { ... }
    static Set intersect(Set s1, Set s2) { ... }
    static boolean member(Object v, Set s) { ... }
}
```

(b) (6 points)
Change your Java solution to part (a) into a Pizza definition that uses parametric polymorphism. That is, rather than containing objects of type `Object`, a set, now referenced as `Set<T>`, contains values of a single fixed type, `T`.

(c) (6 points)
Extend your solution to part (b) to include a member function `map(f,s)`. Parameter `f` is a function that takes a value of type `T` and returns a result of type `T`. Parameter `s` is a set of type `Set<T>`. Function `map` applies `f` to each member of `s`, returning a set of result values (with duplicates removed).

2. (a) (9 points)
Write facts and rules that define a Prolog relation `notIn(V,L)`. `V` is an atomic value and `L` is a list. The relation is satisfied (answers yes) if `V` is not a member of `L`. Thus

```
notIn(1,[]) ⇒ yes
notIn(1,[3,2,1]) ⇒ no
notIn(1,[3,2]) ⇒ yes
```

(b) (8 points)
Explain how Prolog, using your definition of `notIn`, will solve the query

```
notIn(1,[2,1]).
```

(c) (8 points)
What will happen if, using your definition of `notIn`, we pose the following query (where `X` is a free variable)

```
notIn(X,[2,1]).
```

3. (a) (12 points)
What is the type of the following ML function? How did you infer the type you selected?

```
fun f(x,[]) = 0
  | f(x,(a,b)::t) = x a + x b + f(x,t);
```

(b) (13 points)
What does the ML function `f`, defined below, compute? (`g` is used as a subroutine).

```
fun g [x] = []
  | g(h::t) = h::g t;

fun f [x] = x
  | f(h::t) = f(g t);
```

4. (25 points)
A well-known children's game is "tic-tac-toe." A three by three grid contains `x`'s and `o`'s. Three `x`'s is a row, horizontally, vertically, or diagonally is a winner. Assume we represent, in Prolog, a tac-tac-toe board by a list containing three sub-lists. Each sublist contains 3 elements, which can be an `x`, an `o`, or a `b` (representing a blank position). Thus the grid

x		o
o	x	x
o		x

would be represented as `[[x,b,o],[o,x,x],[o,b,x]]`.

Write Prolog rules that define the relation `winner(L)`. `L` is a list of lists representing a tic-tac-toe board as defined above. Given that `L` is ground (already bound to a value), `winner` should succeed if `L` represents a winning position for `x` *or* if `L` can be transformed into a winning position for `x` by transforming a single `b` into an `x`. That is, `winner` should recognize boards that `x` has already won or can win on his next move.

5. (25 points)
A *deque* is a double-ended queue; that is, a queue that allows elements to be added or removed on either end. Define an ML abstract type definition (an `abstype`) for a polymorphic deque. It should provide the following:
   (i) `null`
      A null deque containing no elements.
   (ii) `empty(d)`
      A boolean function that tests whether deque `d` is empty.
   (iii) `enterleft(v,d)`
      Return a deque with a new element `v` added to the left end of deque `d`.
   (iv) `enterright(v,d)`
      Return a deque with a new element `v` added to the right end of deque `d`.
   (v) `rmleft(d)`
      Return a deque with the leftmost element of deque `d` removed.
   (vi) `rmright(d)`
      Return a deque with the rightmost element of deque `d` removed.
   (vii) `left(d)`
      Return the leftmost element of deque `d`.
   (viii) `right(d)`
      Return the rightmost element of deque `d`.