IN 2100: Solutions — Problem Set 2

Peter Ölveczky

February 2, 2022

Exercise 2

1. Yes, there is nothing in the definition that forbids this.
2. Add in Definition 1 that the sets $\{\Sigma_{w,s}\}$ must be *mutually disjoint*. That is, $(w, s) \neq (w', s')$ implies that $\Sigma_{w,s} \cap \Sigma_{w',s'} = \emptyset$.

Exercises 3 and 4

TA explains.

Exercise 5

`NAT-ADD`: the first equation obviously does not contribute to any infinite computation. In the recursive call, the second equation, the first argument position of `+` "decreases" from `s(M)` in the left-hand side to `M` in the right-hand side.

`NAT<`: The first two equations obviously simplify the expression; in the last equation *both* arguments are decreased in the recursive call.

Exercise 6

This one was not entirely trivial, but I found a loop. I use for simplicity mathematical fonts:

$f(s(s(0)), s(s(0))) \rightsquigarrow f(s(0), s(0) + s(s(0))) + f(s(s(0)), s(0)) \rightsquigarrow \cdots$ equations for '+' $\cdots \rightsquigarrow$
$f(s(0), s(s(s(0)))) + f(s(s(0)), s(0)) \rightsquigarrow f(0, 0 + s(s(s(0)))) + f(s(s(s(0))), 0) + f(s(s(0)), s(0))$
$\rightsquigarrow$ (equation for '+') $f(0, s(s(s(0)))) + f(s(s(s(0))), 0) + f(s(s(0)), s(0))$
$\rightsquigarrow \ldots + f(s(s(0)), s(s(0)) + 0) + \ldots \rightsquigarrow \ldots + f(s(s(0)), s(s(s(0)))) + \ldots$

That is, we started with a term $f(s(s(0)), s(s(0)))$ and ended up with a term $\ldots + f(s(s(0)), s(s(0))) + \ldots$. It should then be obvious that those steps can be repeated from the latter $f(s(s(0)), s(s(0)))$, and so on, leading to an infinite loop. Chapter 4 will explain this more carefully. Honestly, this is easier to understand if you use normal notation for numbers: $f(2, 2)$ reduces to an expression that contains $f(2, 2)$.

Exercise 7

This is actually quite simple. As always, let's try the simplest solution, and see what happens if we try two equations with left-hand sides:

```
eq square(0) = ... .
eq square(s(N)) = ... .
```

The second one is just $(N + 1)^2$ is mathematical notation, and we know from school that this equals $N^2 + 2N + 1$. Of course, if we do not have multiplication, we have to write everything using square and addition, so this expression becomes $N^2 + (N + N) + 1$. The final $+1$ corresponds to an `s`, so we get:

```
fmod SQUARE is protecting NAT-ADD .
 op square : Nat -> Nat .
 var N : Nat .
 eq square(0) = 0 .
 eq square(s(N)) = s(square(N) + (N + N)) .
 endfm
```

Let's see if it works. What is the square of 4?

```
Maude> red square(s(s(s(s(0))))) .
reduce in SQUARE : square(s(s(s(s(0))))) .
rewrites: 33 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))
Maude>
```

Exercise 8

TA explains.

Exercise 9

```
set include BOOL off .
load nat-mult.maude
load less-than.maude

fmod NAT1 is protecting NAT-MULT . protecting NAT< .
  ops _monus_ diff min : Nat Nat -> Nat .
  ops _<=_ _>=_  _>_  _==_ : Nat Nat -> Boolean .

  vars M N : Nat .
  eq s(M) == 0 = false .
  eq 0 == s(M) = false .
  eq M == M = true .
  eq s(M) == s(N) = M == N .
  eq M <= N = (M < N) or (M == N) .
  eq M >= N = N <= M .
  eq M > N = N < M .
  ...
endfm
```

and

```
set include BOOL off .
load boolean.maude

fmod BOOLEAN1 is protecting BOOLEAN .
  op _implies_ : Boolean Boolean -> Boolean [prec 61] .
  op if_then_else_fi : Boolean Boolean Boolean -> Boolean .
  vars X Y : Boolean .
  eq true implies X = X .
  eq false implies X = true .
  eq if true then X else Y fi = X .
  eq if false then X else Y fi = Y .
endfm
```

Exercise 10

```
  vars N N' : Nat .
  vars L L' : List .

  eq insertFront(N, nil) = nil N .
  eq insertFront(N, L N') = insertFront(N, L) N' .
  eq N occursIn nil = false .
  eq N occursIn L N' = if (N == N') then true else (N occursIn L) fi .
  eq max(nil) = 0 .                    *** Default/error value
  eq max(L N) = max(max(L), N) .
  eq isSorted(nil) = true .
  eq isSorted(nil N) = true .
  eq isSorted(L N N') = (N <= N') and isSorted(L N) .
```

Exercise 11

1. $2 > 1\ 2 > 1\ 1\ 2 > 1\ 1\ 1\ 2 > \cdots$
2. TA explains
3. ```
   fmod LIST-NAT1-LEXI is
     protecting LIST-NAT1 .
     op _greaterThan_ : List List -> Boolean .   *** Lexicographic comparison
     vars N N' : Nat .
     vars L L' : List .

     *** Base cases:
     eq nil greaterThan L = false .
     eq (L N) greaterThan nil = true .

     *** Both lists are non-empty:
     eq (L N) greaterThan (L' N') =
        if (first(L N) > first(L' N')) then true
        else (if (first(L N) < first(L' N')) then false
              else (rest(L N) greaterThan rest(L' N')) fi) fi .
   endfm
   ```
   ```

Exercise 12

```
bintree(
  bintree(empty, s(s(0)),
          bintree(empty, s(s(s(0))), empty)),
  s(s(s(s(0)))),
  bintree(bintree(empty, s(s(s(s(s(0))))), empty),
          s(s(s(s(s(s(0)))))),
          bintree(empty, s(s(s(s(s(s(s(s(s(0)))))))), empty)))
```

Exercise 13

```
  eq isSearchTree(BT) = isSorted(inorder(BT)) .
```