COMP 317: Semantics of Programming Languages

Summary of Lecture 9: Maude

UNIVERSITY OF LIVERPOOL

This lecture introduces Maude; the material is covered in detail in Chapter 4 of [the draft notes].

Listen to the lecture given on 18/2/16:

Maude is a *logical* language for specifying Abstract Data Types (ADTs) and a *functional* programming language. Functional programming languages provide notations for the programmer to define and evaluate functions. We'll use Maude to define our various denotation functions such as [[*N*]] that takes a binary numeral *N* as input and returns as output the integer that *N* denotes. Since functional programming languages also evaluate functions, we can get Maude to evaluate results such as [[`1101`]], to get the result 13: i.e., Maude can compute the integer 13 as the denotation of the binary numeral `1101`. This will give us an interpreter for SImpL: we can use Maude to execute the semantics of SImpL programs.

Maude specifications are in modules, which are begun with the keywork `fmod` (followed by the name of the module, then the keyword `is`), and ended with the keywork `endfm`. Inside a Maude specification we can declare sorts, operations, and equations. We will use the following example as illustration through the next few lectures. Module `UNARY` specifies unary numerals:

```
fmod  UNARY  is

   sort  Numeral .

   op  0 : -> Numeral .

   op  succ : Numeral -> Numeral .

endfm
```

Recall an abstract data type consists of a set of abstract data values, and some operations that work on those values. In Maude we can give a name to a set of abstract data values with a sort declaration. In `UNARY`, we declare one sort, `Numeral`. This is the set of unary numerals. There is one constant operation, `0`, and one operation `succ` that takes one numeral as input and returns a numeral as output. *Terms* are built by applying operations to arguments. Since `0` takes no arguments, it is a term; applying `succ` to this gives us the term `succ(0)`, and so on, giving us the terms

```
0, succ(0),  succ(succ((0)),  succ(succ(succ(0))),  ...
```

These are the unary numerals.

We can give an inductive definition of addition in the module `UNARY_ARITHMETIC` which imports `UNARY` using the keyword `protecting`:

```
fmod  UNARY_ARITHMETIC  is

   protecting UNARY .

   op  plus : Numeral Numeral -> Numeral .

   vars  M N : Numeral .
   eq  plus(0, N)  =  N .
   eq  plus(succ(M), N)  =  succ(plus(M, N)) .
endfm
```

Now, for example, `plus(succ(0), succ(succ(0)))` is a term of sort numeral, and the equations tell us that this term is equal to `succ(succ(succ(0)))`:

```
plus(succ(0), succ(succ(0)))  =  succ(plus(0, succ(succ(0))))  =  succ(succ(succ(0))) .
```

Grant Malcolm