COSC 3015: Lecture 5

Lecture given by Prof. Caldwell and scribed by Sunil Kothari

September 16, 2008

1 Some Notes on Haskell Syntax

Function names, variable names starts with lowercase letter - followed by zero or more digits, underscores (_) upper or lowercase letters, forward quotes ($\acute{()}$). For example, $x1, x', x_1, x''$.

Datatype and datatype constructor names start with uppercase letters and are otherwise like functions and variable names. For example, *Bool, String, Int , True, False.*

Keywords case, class, data, default, deriving, do, else, if, import, in , infix, infixl, infixr, instance, let, module, newtype, of , then, type, where.

1.1 Layout rule

Each function definition must begin in the same column.

```
a = b + c
  where
    b = 1
    c = 2

d a = a* 2

a = b + c
  where
    {b = 1; c = 2}

d a = a* 2
```

1.2 Comments

Line comments start with "–" and continue to the end of the line. For example,
$fx = x + 1 - -dumbfunction$
Nested comments are of the form $\{- \cdots -\}$

1.3 Local Definition

The following code has the same effect as the ex. 1

```
a = let b = 1
        c = 2
       in
         b + c
```

Here's another version of it.

```
a =  b + c
  where
        b = 1
        c = 2
```

Meaning of let-in. $let\ x = t_1\ in\ t_2$ means $t_2[x := t_1]$ where,

- x is locally defined variable
- t_1 is local declaration of x
- t_2 body of the let-declaration

and $t_2[x := t_1]$ - evaluate expression t_2 where all free occurrences of x get the value of expression t_1. Note that ":=" is a capture-avoiding substitution

```
let x = 1 in
  let x = 2 in
   x
```

$\rightsquigarrow 2$

we can elaborate on it

$\forall x : Int, \forall x : nat.x \geq 0$. Choose arb. y $\in$ int and show $\forall x : Int, \forall x : nat.x \geq 0[x := y] \rightsquigarrow \forall x : nat, x \geq 0$

1.4 if-then-else

A code fragment in an imperative language

```
if x > 0 then
   y :=  x
else
    y:= (-x)
```

In C and C++ there is a conditional expression

```
y:= x >0 ? x: -x
```

i.e. it returns a value of x or -x depending on whether $x > 0$

The type of if-then-else, for example \ b -¿ \ t1 -¿ \ t2 -¿ if b then t1 else t2 is $Bool \rightarrow a \rightarrow a \rightarrow a$.

Note: if True then 0 else "foo" is not well-typed.

Evaluation rules for if-then-else

$$\text{if True then t1 else t2} \rightsquigarrow \text{t1 if False then t1 else t2} \rightsquigarrow \text{t2}$$

If we evaluate *if (True && False) then 0 else 1* then the interpreter responds with 1.

fact k = if k ¡0 then error "fact of a negative number not defined" else if k == 0 then 1 else k * fact (k-1)

```
>:t fact
(Num a, Ord a) => a -> a


fact1 k
 | k ==0 = 1
 | k >0  = k * fact1(k-1)


>:t (Num a, Ord a) => a -> a
```

1.5 Standard Prelude

It's a default library which gets loaded when you start-up. Haskell supports datatype declarations -

Enumerated Types - In Haskell, Bool is defined as a datatype with two constructor *data Bool = True | False*

```
>:t True
 True::Bool
```

But in the prelude it is defined as

data Bool = True | False deriving (Eq, Ord, Enum, Read, Show, Bounded)

Here, deriving tells Haskell to derive all the operations for all the named type classes

```
>:t (show True)
  (show True)::String
```

1.6 Infix operators

```
(&&),(||) :: Bool -> Bool -> Bool

   True  && x = x
   False && _ = False


   True  || x = True
   False || x = x
```

```
> False && infinity
```

In Haskell, we have lazy evaluation so if we try to evaluate the above expression it evaluates to False. But in eager evaluation it will loop forever. There is a little bit of asymmetry here - Infinity and False will not terminate but evaluate to infinity.

```
not ::Bool -> Bool
not True = false
not False = True

otherwise::Bool
not otherwise = True.
```