HW 3 **Prof. Caldwell**
Due: 1 September 2010 **COSC 3015**

1 Today's Lecture

We noted in class that there are two languages at work when using Haskell. There is the *computation language* of executable terms (programs) and the *type language* for describing types. For full Haskell, these languages are very rich. IN this note we just describe the core of these languages that include the types of pairs and functions.

1.1 The Core Language of Types

If $TVar = \{a, b, c, \cdots\}$ is the collection of type variables, then the language of types is give by the following grammar:

$$\begin{array}{rl} \mathcal{T} & ::= x \ \mid \ (T, T') \ \mid \ T \to T' \\ & \text{where } x \in TVar \text{ is a type variable, and} \\ & \qquad T, T' \in \mathcal{T} \text{ are previously constructed type expressions.} \end{array}$$

Thus, standalone type variables are type expressions, and if T and T' are type expressions, then so are (T, T') (denoting the type of pairs whose first elements have type T and whose second elements have type T'. Also, if T and T' are type expressions, then so is $T \to T'$ (which denoted the type of functions from type T to type T'.)

You can think of $(\_, \_)$ as the constructor that maps two type expressions T and T' to the expression (T, T') denoting the type of pairs. Similarly, $\to$ is the (infix) constructor for function type expressions mapping type expressions T and T' to the function tye expression $T \to T'$.

We adopt the convention that $T_1 \to T_2 \to T_3$ is to be interpreted as $T_1 \to (T_2 \to T_3)$. We say that the type constructor $\to$ *associates to the right.*.

Of course the full Haskell language has a much richer language of types than the core presented here, but these are fundamental.

1.2 The Core Computation Language

If $Var = \{x, y, z, \cdots\}$ is the set of variable name in the computation language, then Haskell's core computation language (call it $\mathcal{H}$) is defined as follows:

$$\begin{array}{rl} \mathcal{H} & ::= x \ \mid \ (t, t') \ \mid \ (\backslash x \to t) \ \mid \ t \ t' \\ & \text{where } x \in Var \text{ is a computation variable, and} \\ & \qquad t, t' \in \mathcal{H} \text{ are previously constructed Haskell terms.} \end{array}$$

We sometimes call executable Haskell expressions *terms*. If t and t' are terms then (t, t') is the pair term whose first element is t and whose second element if t'. The term $(\backslash x \to t)$ is called a lambda abstraction, a lambda term or simply an abstraction. This bit of syntax allows us to write a term whose value is a function (without having to assign it a name.) If t and t' are terms then the term $t\ t'$ denotes function application – apply t to t'. It only makes computational sense if t is a function.

We adopt the convention that the application $t_1\ t_2\ t_3$ is to be interpreted as $(t_1\ t_2)\ t_3$. We say that function application *associates to the left.*

1.3 Linking the Computation Language and the Type Language

If t is a term and T is a type expression we write $t :: T$ to mean that the computational term t has type T. This notion is the basis for type checking which we will go into in more depth later in the course.

1.4 Equality of functions

Recall from class that functions are equal if and only if they return equal results on all inputs (this equality is called *extensionality.*)

More formally, we can write:

Definition 1.1. (extensionality) If $f, g :: a \to b$ then they are defined to be (extensionally) equal as follows:

$$f = g \stackrel{\text{def}}{=} \forall x{:}a.\ f(x) = g(x)$$

So, we can prove two functions f and g are equal by choosing an arbitrary x of type a and showing $f(x) = g(x)$.

For example, if $f(x) = |x|$ (the absolute value) and $g(x) = x$ then, $f \neq g$ when we consider them as functions in the type $\mathbb{Z} \to \mathbb{Z}$ since $f(-2) = 2$ and $g(-2) = -2$. But, if we think of these functions as elements[1] of $\mathbb{N} \to \mathbb{N}$, they are equal. To see this, choose an arbitrary $x \in \mathbb{N}$ and argue that $f(x) = g(x)$ *i.e.* that $|x| = x$. But this is trivially true when $x \geq 0$, which follows because $x \in \mathbb{N}$.

Recall the following Haskell definitions.

Definition 1.2. plus

$$\begin{aligned} &plus :: (Integer, Integer) \to Integer \\ &plus(x, y) = x + y \end{aligned}$$

Definition 1.3. plusc

$$\begin{aligned} &plusc :: Integer \to (Integer \to Integer) \\ &plusc\ x\ y = x + y \end{aligned}$$

Definition 1.4. curry

$$\begin{aligned} &curry :: ((a, b) \to c) \to (a \to (b \to c)) \\ &curry\ f\ x\ y = f\ (x, y) \end{aligned}$$

Definition 1.5. uncurry

$$\begin{aligned} &uncurry :: (a \to (b \to c)) \to ((a, b) \to c) \\ &uncurry\ f\ (x, y) = f\ x\ y \end{aligned}$$

In class we proved the following theorem:

Theorem 1.1.

$$curry\ \ plus = plusc$$

[1] Recall that the natural numbers are $\mathbb{N} = \{0, 1, 2, \cdots\}$.

Proof: Note that both $curry\ plus$ and $plusc$ have the type $Integer \to (Integer \to Integer)$ *i.e.* they are functions mapping an $Integers$ to a function of type $Integer \to Integer$. This means we can use extensionality to prove they are equal as functions. We must show the following.

$$\forall x : Integer.\ curry\ \ plus\ x = plusc\ x$$

Assume x is an arbitrary $Integer$. Then we must show

$$curry\ \ plus\ x = plusc\ x$$

But $curry\ \ plus\ x$ and $plusc\ x$ are functions of type $Integer \to Integer$. To show they are equal we use extensionality a second time, to show:

$$\forall y : Integer.\ curry\ \ plus\ x\ y = plusc\ x\ y$$

We choose an arbitrary y of type $Integer$ and show the following

$$curry\ \ plus\ x\ y = plusc\ x\ y$$

Starting with the left side of the equality we get the following:

$$curry\ \ plus\ x\ y \overset{\langle\langle def.\ of\ curry\rangle\rangle}{=} plus\ (x, y) \overset{\langle\langle def.\ of\ plus\rangle\rangle}{=} x + y$$

On the right side of the equality, we have the following:

$$plusc\ x\ y \overset{def.\ of\ plusc}{=} x + y$$

Since both sides of the equality are equal to $x + y$ we see that the functions are equal.
□

2 Homework Problems

Problem 2.1. Prove the following theorem using extensionality.

Theorem 2.1. [uncurry-plusc]

$$uncurry\ plusc = plus$$

Hint: The functions $(uncurry\ plusc)$ and $plus$ have the type $(Integer, Integer) \to Integer$. Extensionality for functions f and g of this type can most conveniently be written as

$$\forall (x, y) :: (Integer, Integer).\ f(x, y) = g(x, y)$$

Problem 2.2. Consider the following function definitions:

Definition 2.1. flip

$$flip :: (a \to b \to c) \to (b \to a \to c)$$
$$flip\ f\ x\ y = f\ y\ x$$

Prove the following theorem using extensionality.

Theorem 2.2. [flip plusc]

$$flip\ plusc = plusc$$

Hints: What is the types of $flip\ plusc$ and $plusc$. You will need the fact that addition is commutative *i.e.* $x + y = y + x$.