A crash course in Haskell

Functions and Programming

Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function.

10:41 AM - 31 Mar 2011

Carmack on Functions

What is Haskell?

A typed, lazy, purely functional programming language

Haskell = \lambda-calculus ++
• better syntax ✔
• types ✔
• built-in features
  o booleans, numbers, characters
  o records (tuples)
  o lists
  o recursion
  o ...

Why Haskell?

Haskell programs tend to be simple and correct

QuickSort in Haskell
sort :: (Ord a) => [a] -> [a]

sort [] = []
sort (x:xs) = sort ls ++ [x] ++ sort rs
  where
    ls = [ l | l <- xs, l <= x ]
    rs = [ r | r <- xs, x < r ]

**Goals for this week**

1. Understand the code above
2. Understand what **typed**, **lazy**, and **purely functional** means (and why it’s cool)

**Haskell vs λ-calculus: similarities**

(1) **Programs**

A program is an **expression** (not a sequence of statements)
It evaluates to a value (it does not perform actions)

- $\lambda$:
  
  $$(\lambda x \to x) \text{apple} \quad -- =\to \text{apple}$$

- Haskell:
  
  $$(\lambda x \to x) "\text{apple}" \quad -- =\to "\text{apple}"$$

(2) **Functions**

Functions are first-class values:

- can be passed as arguments to other functions
- can be returned as results from other functions
- can be partially applied (arguments passed one at a time)

$$(\lambda x \to (\lambda y \to x (x y))) (\lambda z \to z + 1) 0 \quad -- =\to ???$$

*But:* unlike $\lambda$-calculus, not everything is a function!

(3) **Top-level bindings**

Like in Elsa, we can name terms to use them later

**Elsa:**

let T = \(x\ y \rightarrow x\)
let F = \(x\ y \rightarrow y\)

let PAIR = \(x\ y \rightarrow b \rightarrow\) ITE b x y
let FST = \(p \rightarrow p\ T\)
let SND = \(p \rightarrow p\ F\)

eval fst:
  FST (PAIR apple orange)
  \(\Rightarrow\) apple

Haskell:

haskellIsAwesome = True

pair = \(x\ y \rightarrow b \rightarrow\) if b then x else y
fst = \(p \rightarrow p\ haskellIsAwesome\)
snd = \(p \rightarrow p\ False\)

  -- In GHCi:
> fst (pair "apple" "orange")   -- "apple"

The names are called top-level variables

Their definitions are called top-level bindings
Better Syntax: Equations and Patterns

You can define function bindings using **equations**:

```plaintext
pair x y b = if b then x else y  -- same as: pair = |x y b -> ...
fst p   = p True                 -- same as: fst = |p -> ...
snd p   = p False                -- same as: snd = |p -> ...
```

A single function binding can have **multiple** equations with different **patterns** of parameters:
pair x y True = x  -- If 3rd arg matches True,
                    -- use this equation;
pair x y False = y  -- Otherwise, if 3rd arg matches False,
                    -- use this equation.

At run time, the first equation whose pattern matches the actual arguments is chosen

For now, a **pattern** is:

- a **variable** (matches any value)
- or a **value** (matches only that value)

Same as:

```python
pair x y True = x  -- If 3rd arg matches True,
                    -- use this equation;
pair x y b = y  -- Otherwise, use this equation.
```

Same as:

```python
pair x y True = x
pair x y _ = y
```
QUIZ

Which of the following definitions of `pair` is incorrect?

A. \( \text{pair } x \ y = \lambda b \rightarrow \text{if } b \text{ then } x \ \text{else } y \)

B. \( \text{pair } x = \lambda y \ b \rightarrow \text{if } b \text{ then } x \ \text{else } y \)

C.

\[
\begin{align*}
\text{pair } x \ _ \ \text{True} &= x \\
\text{pair } _ \ y \ _ &= y
\end{align*}
\]

D.

\[
\begin{align*}
\text{pair } x \ y \ b &= x \\
\text{pair } x \ y \ \text{False} &= y
\end{align*}
\]

E. all of the above
Equations with guards

An equation can have multiple guards (Boolean expressions):

```
cmpSquare x y | x > y*y    = "bigger :)
               | x == y*y   = "same :|
               | x < y*y    = "smaller :
```

Same as:

```
cmpSquare x y | x > y*y    = "bigger :)
               | x == y*y   = "same :|
               | x < y*y    = "smaller :
               | otherwise  = "smaller :(
```
Recursion

Recursion is built-in, so you can write:

```plaintext
sum n = if n == 0
    then 0
    else n + sum (n - 1)
```

or you can write:

```plaintext
sum 0 = 0
sum n = n + sum (n - 1)
```
The scope of variables

Top-level variable have **global** scope, so you can write:

message = if haskellIsAwesome -- this var defined below
    then "I love CSE 130"
    else "I'm dropping CSE 130"

haskellIsAwesome = True

(f 3) evaluate to?

Or you can write:

--- What does f compute?

f 0 = True
f n = g (n - 1) -- mutual recursion!

\[ f \]
\[ 3 \]
\[ \rightarrow \]
\[ \leftarrow \] g 2
\[ \leftarrow \] g 1
\[ \leftarrow \] g 0
\[ \leftarrow \] false

--- Is this allowed?
haskellIsAwesome = True

haskellIsAwesome = False -- changed my mind

Local variables

You can introduce a new (local) scope using a \texttt{let} –expression:

\begin{verbatim}
sum 0 = 0
sum n = let n' = n - 1
       in n + sum n'  -- the scope of n' is the term after in
\end{verbatim}

Syntactic sugar for nested \texttt{let} –expressions:
sum 0 = 0
sum n = let
    n'   = n - 1
    sum' = sum n'
in n + sum'

If you need a variable whose scope is an equation, use the where clause instead:

cmpSquare x y | x > z   = "bigger :)")"
               | x == z   = "same :|
               | x < z    = "smaller :("

    where z = y*y
Types

What would Elsa say?

```
let WEIRDO = ONE ZERO
```

What would Python say?

```
def weirdo():
    return 0(1)
```
What would Java say?

```java
void weirdo() {
    int zero;
    zero(1);
}
```

In Haskell every expression either has a type or is ill-typed and rejected statically (at compile-time, before execution starts)

- like in Java
- unlike λ-calculus or Python

```haskell
weirdo = 1 0 -- rejected by GHC
```