Higher-Order Programming

Functions as Data

In Haskell (and other functional languages), functions can be treated as "ordinary data"—they can be passed as arguments to other functions, returned as results, stored in lists, etc., etc.

Taking a function as an argument:

\[
\text{thrice} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}
\]
\[
\text{thrice} f x = f (f (f x))
\]

\[
\text{plus2} :: \text{Int} \rightarrow \text{Int}
\]
\[
\text{plus2} x = x + 2
\]

\[
\text{foo} :: \text{Int}
\]
\[
\text{foo} = \text{thrice} \text{plus2} 10 \quad -- \text{foo} = 16
\]

Returning a function as a result:

\[
\text{plusn} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]
\[
\text{plusn} n = f
\]
\[
\begin{align*}
\text{where} & \quad f x = n + x \\
\text{plus5} & :: \text{Int} \rightarrow \text{Int} \\
\text{plus5} & = \text{plusn} 5
\end{align*}
\]

\[
\begin{align*}
\text{bar1} & = \text{plus5} 10 \quad -- \text{bar1} = 15 \\
\text{-- Or just...} \\
\text{bar2} & = (\text{plusn} 5) 10 \quad -- \text{bar2} = 15
\end{align*}
\]

The type constructor \(-\rightarrow\) is right-associative — i.e.,

\[
\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

means the same as

\[
\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})
\]

That is, a function of type \(\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\) can be thought of as a function that takes an integer and returns a function from integers to integers!

So we can write \(\text{plusn}\) in a simpler way:

\[
\text{plusn'} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \quad -- \text{i.e., } \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]
\[
\text{plusn'} n x = n + x \quad -- \text{i.e., } \text{plusn'} = (+)
\]

Each time we use \(\text{plusn}\), we can choose whether to apply it to two integers to get an integer or to "partially apply" it to just one integer, yielding a function.

Putting these together...

\[
\text{thriceplus2} :: \text{Int} \rightarrow \text{Int}
\]
\[
\text{thriceplus2} = \text{thrice} \text{plus2} \quad -- \text{partial application!}
\]

\[
\text{baz} :: \text{Int} \rightarrow \text{Int}
\]
\[
\text{baz} = \text{thrice} \text{thriceplus2} \quad -- \text{again!!}
\]

\[
\text{-- Check: What is (baz 0)?}
\]
### Polymorphism

Many of Haskell's predefined functions are polymorphic:

- `(++) :: [a] -> [a] -> [a]`
- `id :: a -> a`
- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `[] :: [a] -- interesting!`

Quick check: what is the type of `tag1`?

\[\text{tag1 } x = (1, x)\]

### Polymorphic Data Structures

Polymorphic functions — functions that can operate on any type of data — are often associated with polymorphic data structures — structures that can contain any type of data.

The previous examples involved lists and tuples. In particular, here are the types of the list and tuple constructors:

- `(,) :: a -> b -> (a, b)`
- `(:) :: a -> [a] -> [a]`

We can also define new polymorphic data structures...

### A User-Defined Polymorphic Data Structure

The type variable `a` on the left-hand-side of the `=` tells Haskell that `Maybe` is a polymorphic data type:

\[\text{data } \text{Maybe } a = \text{Nothing} \mid \text{Just } a\]

Note the types of the constructors:

- `Nothing :: Maybe a`
- `Just :: a -> Maybe a`

Thus:

- `Just 3 :: Maybe Int`
- `Just "x" :: Maybe String`
- `Just (3, True) :: Maybe (Int, Bool)`
- `Just (Just 1) :: Maybe (Maybe Int)`

### Maybe May Be Useful

The most common use of Maybe is with a function that "may" return a useful value, but may also fail.

For example, the division operator `div` in Haskell will cause a run-time error if its second argument is zero. Thus we may wish to define a safe division function, as follows:

\[\text{safeDivide :: Int -> Int -> Maybe Int}
\]
\[\text{safeDivide } x 0 = \text{Nothing}
\]
\[\text{safeDivide } x y = \text{Just } (x \text{ `div` } y)\]

The "a" in the type of `length` is a placeholder that can be replaced with any type when `length` is applied.

\[\text{length } [1, 2, 3] \Rightarrow 3\]
\[\text{length } ["a", "b", "c"] \Rightarrow 3\]
\[\text{length } [[1], [], [2, 3]] \Rightarrow 3\]
Abstraction Over Recursive Definitions

Recall from Section 4.1:

\[
\text{transList} :: [\text{Vertex}] \to [\text{Point}]
\]
\[
\text{transList} \; \emptyset \; = \; \emptyset
\]
\[
\text{transList} \; (p:ps) \; = \; \text{trans} \; p \; : \; \text{transList} \; ps
\]

(where \text{trans} converts ordinary cartesian coordinates into screen coordinates).

Also, from Chapter 3:

\[
\text{putCharList} :: [\text{Char}] \to [\text{IO ()}]
\]
\[
\text{putCharList} \; \emptyset \; = \; \emptyset
\]
\[
\text{putCharList} \; (c:cs) \; = \; \text{putChar} \; c \; : \; \text{putCharList} \; cs
\]

These definitions are very similar. Indeed, the only thing different about them (besides the variable names) is the function \text{trans} vs. the function \text{putChar}.

Abstraction Yields \text{map}

Since \text{trans} and \text{putChar} are the differing elements, they should be arguments to the abstraction. In other words, we would like to define a function — let's call it \text{map} — such that \text{map trans} behaves like \text{transList} and \text{map putChar} behaves like \text{putCharList}.

No problem:

\[
\text{map} \; f \; \emptyset \; = \; \emptyset
\]
\[
\text{map} \; f \; (x:xs) \; = \; f \; x \; : \; \text{map} \; f \; xs
\]

Now it is easy to redefine \text{transList} and \text{putCharList} in terms of \text{map}:

\[
\text{transList} \; xs \; = \; \text{map} \; \text{trans} \; xs
\]
\[
\text{putCharList} \; cs \; = \; \text{map} \; \text{putChar} \; cs
\]

map is Polymorphic

The great thing about \text{map} is that it is polymorphic. Its most general (or principal) type is:

\[
\text{map} \; :: \; (a \to b) \to [a] \to [b]
\]

Whatever type is instantiated for the type variable \text{a} must be the same at both instances of \text{a}, and similarly for \text{b}.

For example, since \text{trans} :: \text{Vertex} \to \text{Point}, we have

\[
\text{map} \; \text{trans} \; :: \; [\text{Vertex}] \to [\text{Point}]
\]

and since \text{putChar} :: \text{Char} \to \text{IO ()},

\[
\text{map} \; \text{putChar} \; :: \; [\text{Char}] \to [\text{IO ()}]
\]

Digression: Arithmetic Sequences

Haskell provides a convenient special syntax for lists of numbers obeying simple rules:

\[
[1 \ldots 6] \Rightarrow [1,2,3,4,5,6]
\]
\[
[1,3 \ldots 9] \Rightarrow [1,3,5,7,9]
\]
\[
[5,4 \ldots 1] \Rightarrow [5,4,3,2,1]
\]
\[
[2.4, 2.1 \ldots 0.3] \Rightarrow [2.4, 2.1, 1.8, 1.5, \text{etc.}]
\]

Another Example of Map

\[
\text{circles} :: [\text{Shape}]
\]
\[
\text{circles} \; = \; \text{map} \; \text{circle} \; [2.4, 2.1 \ldots 0.3]
\]

Now let's draw them...
Another useful higher-order function:

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _ _ = []
```

For example:

```haskell
zip [1,2,3] [True,False,False] ⇒ [(1,True), (2,False), (3,False)]
```

Quick check: What does `zip [1..3] [1..5]` yield?

```
Drawing Colored Shapes
```

```haskell
colCircles :: [(Color,Shape)]
colCircles = zip [Red,Blue,Green,
                         Cyan,Red,Magenta,
                         Yellow,White]
circles
```

```
Main Action
```

```haskell
g = do w <- openWindow "Bulls eye" (600,600)
drawShapes w colCircles
k <- getKey w
closeWindow w
main = runGraphics g
```

```
The Result
```

```
When to Define Higher-Order Functions
```

```
Recognizing repeating patterns is the key, as we did for map. As another example, consider:
```

```haskell
listSum [] = 0
listSum (x:xs) = x + listSum xs
listProd [] = 1
listProd (x:xs) = x * listProd xs
```

Note the similarities. Also note the differences (0 vs. 1 and + vs. *): it is these that will become parameters to the abstracted function.
Fold

Abstracting out the differences ($op$ and $init$) leaves this common part:

- $fold \ op \ init \ [] = init$
- $fold \ op \ init \ (x:xs) = x \ 'op' \ fold \ op \ init \ xs$

We recover $listSum$ and $listProd$ by instantiating $fold$ with the appropriate parameters:

- $listSum \ xs = fold \ (+) \ 0 \ xs$
- $listProd \ xs = fold \ (*) \ 1 \ xs$

Note that $fold$ is polymorphic:

- $fold :: (a -> b -> b) -> b -> [a] -> b$

Two Folds Are Better Than One

The $fold$ function is predefined in Haskell, but it is actually called $foldr$, because it “folds from the right.” That is:

- $foldr \ op \ init \ (x1 : x2 : \ldots : xn : []) \Rightarrow x1 \ 'op' \ (x2 \ 'op' \ (\ldots (xn \ 'op' \ init) \ldots))$

There is another predefined function $foldl$ that “folds from the left”:

- $foldl \ op \ init \ (x1 : x2 : \ldots : xn : []) \Rightarrow (\ldots((init \ 'op' \ x1) \ 'op' \ x2)\ldots) \ 'op' \ xn$

Why two folds? Because sometimes using one can be more efficient than the other. For example:

- $foldr \ (++) \ [] \ [x,y,z] \Rightarrow x \ ++ \ (y \ ++ \ z)$
- $foldl \ (++) \ [] \ [x,y,z] \Rightarrow (x \ ++ \ y) \ ++ \ z$

The former is considerably more efficient than the latter (as discussed in the book); but this is not always the case — sometimes $foldl$ is more efficient than $foldr$. Choose wisely!

Another Application of Fold

We have seen the function $sequence_\_\_$, which takes a list of actions of type $IO()$ and produces a single action of type $IO()$.

We can define $sequence_\_\_$ in terms of $\gg$ and $foldl$ as follows:

- $sequence_\_ \ :: \ [IO()] \rightarrow IO()$
- $sequence_\_ \ acts = foldl \ (\gg) \ (return \ ()) \ acts$

Reversing a List

Obvious but inefficient (why?):

- $reverse \ [] = []$
- $reverse \ (x:\!:xs) = reverse \ xs \ ++ \ [x]$

Much better (why?):

- $reverse \ xs = rev \ [] \ xs$
  where $rev \ acc \ [] = acc$
  $rev \ acc \ (x:xs) = rev \ (x:acc) \ xs$

This looks a lot like $foldl$. Indeed, we can redefine $reverse$ as:

- $reverse \ xs = foldl \ revOp \ [] \ xs$
  where $revOp \ a \ b = b : a$