Five Fits of Functional Programming

Conor McBride

February 1, 2012

Introduction

This five lecture topic introduces functional programming, using the Haskell programming language [Peyton Jones et al., 2003]. For further organisational information and details of the assignments, consult `http://personal.cis.strath.ac.uk/~conor/FFFP`.

0.1 Teaching Materials

These notes, program files, and so on, are subject to revision, and are thus maintained using the revision control system `darcs`[1]. You will need your own local copy of my repository. To make one, open a terminal window, change directory to somewhere suitable, then issue the incantation

```
darcs get http://personal.cis.strath.ac.uk/~conor/FFFP
```

That command will download and create your own `FFFP` directory.

Every so often, I'll notify you of updates, which you can get hold of by changing directory to somewhere inside `FFFP` and issuing the incantation

```
darcs pull
```

which will check if I've made any changes and ask you if you want to download them. In response, press `y` repeatedly to download the changes one at a time, or `a` to say 'yes to all'.

0.2 Help

We have an email help service, `fp-help@cis.strath.ac.uk`, staffed by postgraduate students, available for sensible questions relating to the touch business of learning functional programming at Strathclyde. You can post to fp-help only from a CIS account. Before you do so, please read our guide `http://local.cis.strath.ac.uk/teaching/ug/classes/CS.314/fp-help.html`. You may also find it useful to read the companion guide `http://local.cis.strath.ac.uk/teaching/ug/classes/CS.314/fp-helpers.html`.

Think twice before you ask for help, because thinking twice often solves the problem anyway! Email fp-help rather than emailing me alone: I certainly receive fp-help, but so do other helpful people, so you increase your chances of a sensible outcome. Knocking on my door is always an option. The worst that can happen is that I ask you to make an appointment.

[1] which happens to be implemented in Haskell

Fit 1

What thing *is* that?

- understand datatypes as sets of things
- understand functions as machines which are things which map things to things
- understand computation as finding out which thing an expression means

1.1 Expressions and Values

What are functional programs? What data do they manipulate? How do they *run*? The answers to these questions are quite different from the corresponding answers for object-oriented languages like Java. When you're used to Java, functional programming an seem rather alien, so expect to take some time to adjust. Some aspects, however, are quite familiar.

*What **is*** $2 + 2$ *?* $\quad 4$

*What **is*** $(4 + 2) * (10 - 3)$ *?* $\quad 6 * (10 - 3)$, *which **is*** $6 * 7$, *which **is*** 42

You've been evaluating *arithmetic* expressions for years. Evaluation works by replacing some component of an expression by a simpler component of equal value, e.g. replacing $(4 + 2)$ by 6, keeping the surrounding context the same. You keep going until you reach a numerical value. The value is the *thing* the expression *means*. Evaluation answers the question 'what thing *is* that?'.

Functional programming uses *evaluation* as its mechanism for computation. In a Java program, you direct computation by saying what things *do*: in a Haskell program, you direct computation by saying what things *are*.

1.2 Types and Values

You're used to working with numerical values $2 + 2 = 4$. You're probably fine with Boolean values $3 \leq 5 = \mathsf{True}$. Haskell chops up the world of values into *types*. Type distinctions are useful in lots of ways, but crucially, we use them to indicate the ways we expect components to be compatible. We write

$$3 :: \mathsf{Int} \qquad \mathsf{True} :: \mathsf{Bool}$$

to say that '3 is an integer' and 'True' is a Boolean value. You can read :: as 'is a' or 'has type'. In Haskell, values and types are separate. Values live to the left of :: and are used in actual computation; types live to the right of :: and are used by the compiler to check that computations make sense, but are thrown away at run time. The two do not mix: apart from any other distinctions, you can tell them apart by *where they are*.

Haskell lets us invent our own types — *algebraic datatypes*. Here's an example.

```
data List a             -- the type constructor
  = Nil                 -- a value constructor
  | Cons a (List a)     -- another value constructor
```

What does this do? It doesn't *do* anything. What does this *say*? The head of the declaration, between **data** and = (a place for types) asserts the existence of some types; the 'body', after the =, asserts the existence of some values.

Which types exist? The declaration head means that List a is a type whenever a is a type — as it happens, the type of lists whose elements have type a. As Int and Bool are types, we acquire types List Int and List Bool, but then we can build List (List Int), the type of lists whose elements are themselves lists of integers, List (List Bool), List (List (List Int)), and so on. Types are plugged together like LEGO™ bricks, and we've just invented a new component, the *type constructor*, List, with one parameter. Meanwhile, Bool is a type constructor with no parameters.

Which values exist in these types? The declaration body tells us the only possible ways we can use to build values in our new types. We give a bunch of options, separated by the | symbol. Here, there are two *value* constructors, Nil and Cons. Each packs up a (possibly empty) bunch of components to make a value in List a, and the declaration says what types those components must have. Nil packs up *no* components, to make a List a, whatever a might be — Nil represents an empty list. Meanwhile to construct a List a with at least one element, Cons takes an element of a and another List a — the first element (or 'head') and the list of the rest (or 'tail'). Meanwhile, True and False are the value constructors for Bool, and neither packs up any components.

```
Nil :: List Int
Cons 3 Nil :: List Int   -- because 3 :: Int and Nil :: List Int
Cons 2 (Cons 3 Nil) :: List Int   -- because 2 :: Int and Cons 3 Nil :: Int
Cons 1 (Cons 2 (Cons 3 Nil)) :: List Int   -- because..?
```

Note that

Cons (Cons 1 Nil) (Cons 2 Nil) $\not::$ List Int   because   Cons 1 Nil $\not::$ Int

Names. Type Constructors and Value Constructors have Names beginning with a Capital Letter, so you know them when you see them.

Let's have some more examples.

$\mathsf{Nil} :: \mathsf{List\ Bool}$ -- Nil always makes a List a but it doesn't determine a
$\mathsf{Cons\ True\ Nil} :: \mathsf{List\ Bool}$ -- because True :: Bool and Nil :: List Bool
$\mathsf{Cons\ False\ (Cons\ True\ Nil)} :: \mathsf{List\ Bool}$ -- because..?

$\mathsf{Nil} :: \mathsf{List\ (List\ Int)}$
$\mathsf{Cons\ Nil\ Nil} :: \mathsf{List\ (List\ Int)}$ -- because Nil :: List Int and Nil :: List (List Int)
$\mathsf{Cons\ (Cons\ 1\ Nil)\ (Cons\ Nil\ Nil)} :: \mathsf{List\ (List\ Int)}$
-- because Cons 1 Nil :: List Int and Cons Nil Nil) :: List (List Int)

The Nil example shows that you should not expect a value to have only one type. Nil :: List a for *any* a. Cons Nil Nil :: List (List a) for *any* a, making use of two different types for Nil.

You should, however, be able to *check* whether a value has a given type: check that the value constructor is one of the options allowed for the given type constructor, then check that you have the right number of components, and that they each have the type demanded by the **data** declaration, once you instantiate the parameters properly. Let's try to check

$$\mathsf{Cons\ 1\ (Cons\ True\ Nil)} :: \mathsf{List\ Int} \qquad ?$$

Yes, Cons is allowed by List Int, and with $a = \mathsf{Int}$, we must now check that 1 :: Int (yes!) and that Cons True Nil :: List Int. Yes, Cons is allowes by List Int, and with $a = \mathsf{Int}$, we must now check that True :: Int (**no!**) and that Nil :: List Int (yes!). But it only takes one wrong thing to spoil it all: that example does *not* typecheck, and just as well.

1.3 Functions are Things, Too

A function is a mechanism which computes one thing from another, mapping an input thing (or 'argument') to an output thing (or 'result'). In Haskell, a function is a value[1] Haskell has a special type constructor for functions:

$$a \to b$$

is a type if a and b are types — it's the type of functions mapping values of type a to values of type b. For example, we might have a function which adds up the integers in a given list.

$\mathsf{total} :: \mathsf{List\ Int} \to \mathsf{Int}$

We would expect to *apply*[2] total to some List Int thing, and get an Int back. In Haskell, you apply a function to its argument by writing the function next to the argument (with some space in between). That is,

$$\text{if } f :: a \to b \text{ and } x :: a, \text{ then } f\ x :: b.$$

So

$$\mathsf{total} :: \mathsf{List\ Int} \to \mathsf{Int} \text{ and } \mathsf{Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))} :: \mathsf{List\ Int},$$
$$\text{therefore } \mathsf{total\ (Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil)))} :: \mathsf{Int}.$$

[1] Where in Java, an object may have methods but a method is not an object. We say that in Haskell, functions are 'first-class citizens'.

[2] An older name for functional programming is 'applicative' programming.

Names. Note that the names of *functions* begin with a small letter, so you can easily tell them apart from Value Constructors.

You can see that $\mathsf{total}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ \mathsf{Nil})))$ is not a *value* of type Int, because it isn't a numeral. Rather, it's an *expression* of type Int, involving an application of the function total which has not yet been computed. We should hope that this expression has a value $6 :: \mathsf{Int}$. Which reminds me of another important thing:

$$\boxed{\textbf{If an expression has a type, its value has the same type.}}$$

How do we define things (including functions)? We declare their type, then we give a bunch of *evaluation rules*, like this:

$$\begin{array}{lll}
\mathsf{total} :: \mathsf{List\ Int} \to \mathsf{Int} & & \\
\mathsf{total\ Nil} & = 0 & \text{-- rule 1} \\
\mathsf{total}\ (\mathsf{Cons}\ x\ xs) & = x + \mathsf{total}\ xs & \text{-- rule 2}
\end{array}$$

These evaluation rules explain how to compute expressions in which total is applied, saying what $\mathsf{total}\ v$ must be for each possible input value v. See? We know that a $\mathsf{List\ Int}$ value must either be Nil or something of the form $\mathsf{Cons}\ x\ xs$: those two *patterns* cover all the possibilities.

$$\begin{array}{ll}
\mathsf{total}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ \mathsf{Nil}))) & \text{-- rule 2} \\
= 1 + \mathsf{total}\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ \mathsf{Nil})) & \text{-- rule 2} \\
= 1 + (2 + \mathsf{total}\ (\mathsf{Cons}\ 3\ \mathsf{Nil})) & \text{-- rule 2} \\
= 1 + (2 + (3 + \mathsf{total\ Nil})) & \text{-- rule 1} \\
= 1 + (2 + (3 + 0)) & \\
= 1 + (2 + 3) & \\
= 1 + 5 & \\
= 6 &
\end{array}$$

So, we're not storing a running total in memory, then modifying the total by adding stuff to it. We're following the rules to figure out what things are.

Suppose, for example, I want to explain how to concatenate or 'append' two lists. Can I define a function with *two* inputs? No! I can do something even better! I can define a function which makes a function!

$$\mathsf{append} :: \mathsf{List}\ a \to (\mathsf{List}\ a \to \mathsf{List}\ a)$$

Note that append works for any element type, so long as we're consistent about it. I indicate this generality by writing a *type variable*, a, for the element type. Just as with Nil and Cons, you can use them for any specific instance of a. For example, (with $a = \mathsf{Int}$), we can have

$$\mathsf{append}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil})) :: \mathsf{List\ Int} \to \mathsf{List\ Int}$$

being the function which appends that particular list of integers onto any other list of integers, so that

$$(\mathsf{append}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil})))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})) :: \mathsf{List\ Int}$$

That is to say, we can express very general operations, then specialise them a little at a time. We'll see more of that in a moment.

Parentheses or not? We arrange the grouping to make functions-which-make-functions look like functions with multiple inputs. $a \to b \to c$ means $a \to (b \to c)$. We may actually write

$$\mathsf{append} :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a$$

Correspondingly, $f\ x\ y$ means the repeated application $(f\ x)\ y$. That is, if $f :: a \to b \to c$ and $x :: a$ and $y :: b$, then $f\ x\ y :: c$. For example, we may drop the parentheses in the above and just say

$$\mathsf{append}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})) :: \mathsf{List}\ \mathsf{Int}$$

Note that if we want to give a large expression (more than one symbol) as a function's argument, we need to put it in parentheses. For example

$$\mathsf{total}\ (\mathsf{append}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil}))) :: \mathsf{Int}$$

should give 10, whereas without the parentheses,

$$\mathsf{total}\ \mathsf{append}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})) \quad \text{-- type error}$$

means

$$(\mathsf{total}\ \mathsf{append})\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})) \quad \text{-- obvious type error}$$

which is clearly bad, because append is a function, not a list of integers, so (total append) is nonsense. Note that application groups more tightly than any infix operator, so $f\ x + y$ means $(f\ x) + y$.

Meanwhile. Here's a definition of append.

$$\begin{array}{l}
\mathsf{append} :: \mathsf{List}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a \\
\mathsf{append}\ \mathsf{Nil}\qquad\qquad\ ys = ys \\
\mathsf{append}\ (\mathsf{Cons}\ x\ xs)\ ys = \mathsf{Cons}\ x\ (\mathsf{append}\ xs\ ys)
\end{array}$$

Let's evaluate

$$\begin{array}{l}
\mathsf{total}\ (\mathsf{append}\ (\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil}))) \\
\ = \mathsf{total}\ (\mathsf{Cons}\ 1\ (\mathsf{append}\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil}))) \\
\ = 1 + \mathsf{total}\ (\mathsf{append}\ (\mathsf{Cons}\ 2\ \mathsf{Nil}))\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})) \\
\ = 1 + \mathsf{total}\ (\mathsf{Cons}\ 2\ (\mathsf{append}\ \mathsf{Nil}\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})))) \\
\ = 1 + (2 + \mathsf{total}\ (\mathsf{append}\ \mathsf{Nil}\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil})))) \\
\ = 1 + (2 + \mathsf{total}\ (\mathsf{Cons}\ 3\ (\mathsf{Cons}\ 4\ \mathsf{Nil}))) \\
\ = 1 + (2 + (3 + \mathsf{total}\ (\mathsf{Cons}\ 4\ \mathsf{Nil}))) \\
\ = 1 + (2 + (3 + (4 + \mathsf{total}\ \mathsf{Nil}))) \\
\ = 1 + (2 + (3 + (4 + 0))) \\
\ = 1 + (2 + (3 + 4)) \\
\ = 1 + (2 + 7) \\
\ = 1 + 9 \\
\ = 10
\end{array}$$

See what happens at the beginning? We can't make any progress with the total application until we've done a bit of appending, but we don't need to compute the append (Cons 1 (Cons 2 Nil)) (Cons 3 (Cons all the way to a *value*. Rather, we can get away with computing it just enough that we can see which total evaluation rule to use.

1.4 Lab Exercise

In a terminal window, change to your `FFFP` directory and issue the command

```
make ex1
```

After a certain amount of mechanical jiggery-pokery, you should see a display showing

$$\boxed{\text{append (Cons 1 (Cons 2 Nil)) (Cons 3 (Cons 4 Nil))}}$$

with the boxed 'in focus' expression highlighted by a black background.

Navigation. If the expression in focus is an application, you can zoom in on the *function* by pressing $\boxed{\leftarrow}$. Try it now!

$$\boxed{\text{append (Cons 1 (Cons 2 Nil))}}\text{ (Cons 3 (Cons 4 Nil))}$$

If the expression in focus is an application, you can zoom in on the *argument* by pressing $\boxed{\rightarrow}$. Try it!

$$\text{append }\boxed{\text{(Cons 1 (Cons 2 Nil))}}\text{ (Cons 3 (Cons 4 Nil))}$$

If the focus is inside an application (in either function or argument position), you can zoom out by pressing $\boxed{\uparrow}$. Try it twice, and you should get back to the start.

$$\boxed{\text{append (Cons 1 (Cons 2 Nil)) (Cons 3 (Cons 4 Nil))}}$$

Rewriting. Try pressing $\boxed{\text{b}}$. You should see.

$$\boxed{\text{Cons 1 (append (Cons 2 Nil) (Cons 3 (Cons 4 Nil)))}}$$

Move the focus to

$$\text{Cons 1 }\boxed{\text{(append (Cons 2 Nil) (Cons 3 (Cons 4 Nil)))}}$$

and press $\boxed{\text{b}}$ again. What happens?

$$\text{Cons 1 }\boxed{\text{(Cons 2 (append Nil (Cons 3 (Cons 4 Nil))))}}$$

Each time you press $\boxed{\text{b}}$, you rewrite the term in focus by one of the evaluation rules for append.

In fact, lots of letter-keys apply rewrite rules to the term in focus, provided it matches the left-hand-side. Here are all the definitions in play. Note that each rule is labelled with a letter in a comment: Haskell uses -- to signal 'comment until end of line'. That letter is the key to press to make the rule fire.

Keep rewriting until you reach a value, then zoom all the way out and press $\boxed{\text{return}}$, to get the next problem.

At any time, you can quit by pressing $\boxed{\text{escape}}$.

Your mission is to work through all the evaluation problems. You get the 10% for finishing the lot! The method for confirming that you've done will be specified in the lab.

By the way, look at insert. You'll notice that the rule for Node splits into two cases, one if $x \leq y$, the other if not: whenever you see a call to insert which matches that Node-pattern, you will need to decide which of those two cases applies.

Good luck!

```
data List a
  = Nil
  | Cons a (List a)
    deriving Show

data Tree a
  = Leaf
  | Node (Tree a) a (Tree a)
    deriving Show

append :: List a → List a → List a
append Nil         ys = ys                      -- [a]
append (Cons x xs) ys = Cons x (append xs ys)   -- [b]

insert :: Int → Tree Int → Tree Int
insert x Leaf                        = Node Leaf x Leaf        -- [c]
insert x (Node lt y rt) | x ≤ y      = Node (insert x lt) y rt -- [d]
                        | otherwise  = Node lt y (insert x rt) -- [e]

foldList :: t → (a → t → t) → List a → t
foldList n c Nil         = n                       -- [f]
foldList n c (Cons x xs) = c x (foldList n c xs)   -- [g]

makeTree :: List Int → Tree Int
makeTree = foldList Leaf insert   -- [h]

glueIn :: List a → a → List a → List a
glueIn xs y zs = append xs (Cons y zs)   -- [i]

flatten :: Tree a → List a
flatten Leaf           = Nil                                  -- [j]
flatten (Node lt x rt) = glueIn (flatten lt) x (flatten rt)   -- [k]

compose :: (b → c) → (a → b) → a → c
compose f g a = f (g a)   -- [l]

sort :: List Int → List Int
sort = compose flatten makeTree   -- [m]
```

Fit 2

Am I *seeing* things?

2.1 Last time's one-minute-paper quiz

Which of the following does not have a type?

a Cons Nil Nil	b Cons Nil (Cons Nil Nil)
c Cons (Cons 1 Nil) (Cons 1 Nil)	d Cons (Cons 1 Nil) Nil

Let's work through it. Firstly, let's have the types of the pieces. The declaration

```
data List a
  = Nil
  | Cons a (List a)
```

tells us that

```
Nil  ::           List a
Cons :: a → List a → List a
```

which is to say that Nil and Cons build lists of type List a for *any* type a of elements. Each time we use Nil or Cons, the element type a is up for grabs. It's like LEGO. We can plug Nil into any hole whose type matches List a for some a. We can plug Cons (? :: a) (? :: List a) into any hole whose type matches List a, making two new holes, one of type a, one of type List a. We match types by choosing values for the type variables. *Think: upper case type constructors **rigid**, lower case type variables **flexible**.*

We also know[1]

```
1 :: Int
```

We need to check whether each of the above expressions can be given a type. We can work a little bit at time, going 'outside in'. Let's try Cons Nil Nil.

Can we plug Cons Nil Nil into a hole ? :: x? Looking at the type of Cons, yes, taking x = List y, $a = y$ for some y

Cons (? :: y) (? :: List y) :: List y

[1]a convenient oversimplification

provided we can plug Nil into each of a hole of type y and a hole of type $\mathsf{List}\ y$. Looking at the type of Nil, the latter is always possible

$$\mathsf{Cons}\ (?::y)\ \mathsf{Nil} :: \mathsf{List}\ y$$

but the former can only happen if we choose $y = \mathsf{List}\ z$, for some z. We end up with

$$\mathsf{Cons}\ \mathsf{Nil}\ \mathsf{Nil} :: \mathsf{List}\ (\mathsf{List}\ z)$$

$\mathsf{Cons}\ \mathsf{Nil}\ \mathsf{Nil}$ is a list of lists whose single element is the empty list.

Moving on, let's try to plug $\mathsf{Cons}\ \mathsf{Nil}\ (\mathsf{Cons}\ \mathsf{Nil}\ \mathsf{Nil})$ into $?::x$. The first step is the same.

$$\mathsf{Cons}\ (?::y)\ (?::\mathsf{List}\ y) :: \mathsf{List}\ y$$

and plugging Nil into the first hole, we again get that $y = \mathsf{List}\ z$ for some z

$$\mathsf{Cons}\ \mathsf{Nil}\ (?::\mathsf{List}\ (\mathsf{List}\ z)) :: \mathsf{List}\ (\mathsf{List}\ z)$$

and we know we can plug $\mathsf{Cons}\ \mathsf{Nil}\ \mathsf{Nil}$ into the remaining hole

$$\mathsf{Cons}\ \mathsf{Nil}\ (\mathsf{Cons}\ \mathsf{Nil}\ \mathsf{Nil}) :: \mathsf{List}\ (\mathsf{List}\ z)$$

For option (c), let's try plugging $\mathsf{Cons}\ (\mathsf{Cons}\ 1\ \mathsf{Nil})\ (\mathsf{Cons}\ 1\ \mathsf{Nil})$ into $?::x$. As ever,

$$\mathsf{Cons}\ (?::y)\ (?::\mathsf{List}\ y) :: \mathsf{List}\ y$$

and we can plug a $\mathsf{Cons}??$ into the first hole, just as before, provided we take $y = \mathsf{List}\ z$ for some z

$$\mathsf{Cons}\ (\mathsf{Cons}\ (?::z)\ (?::\mathsf{List}\ z))\ (?::\mathsf{List}\ (\mathsf{List}\ z)) :: \mathsf{List}\ (\mathsf{List}\ z)$$

Now we plug 1 in the leftmost hole, and that means $z = \mathsf{Int}$.

$$\mathsf{Cons}\ (\mathsf{Cons}\ 1\ (?::\mathsf{List}\ \mathsf{Int}))\ (?::\mathsf{List}\ (\mathsf{List}\ \mathsf{Int})) :: \mathsf{List}\ (\mathsf{List}\ z)$$

Nil goes in the next hole, no problem.

$$\mathsf{Cons}\ (\mathsf{Cons}\ 1\ \mathsf{Nil})\ (?::\mathsf{List}\ (\mathsf{List}\ \mathsf{Int})) :: \mathsf{List}\ (\mathsf{List}\ z)$$

But now we have to put $\mathsf{Cons}\ 1\ \mathsf{Nil}$ into $?::\mathsf{List}\ (\mathsf{List}\ \mathsf{Int})$. Start with the $\mathsf{Cons}??$.

$$\mathsf{Cons}\ (\mathsf{Cons}\ 1\ \mathsf{Nil})\ (\mathsf{Cons}\ (?::\mathsf{List}\ \mathsf{Int})\ (?::\mathsf{List}\ (\mathsf{List}\ \mathsf{Int}))) :: \mathsf{List}\ (\mathsf{List}\ z)$$

and the Nil fits

$$\mathsf{Cons}\ (\mathsf{Cons}\ 1\ \mathsf{Nil})\ (\mathsf{Cons}\ (?::\mathsf{List}\ \mathsf{Int})\ \mathsf{Nil}) :: \mathsf{List}\ (\mathsf{List}\ z)$$

but can we put 1 into $?::\mathsf{List}\ \mathsf{Int}$. No.[2] This one's not happening. If you look in the element positions, the elements which must have the same type are 1 and $\mathsf{Cons}\ 1\ \mathsf{Nil}$, so no chance.

Just to finish the job let's check that $\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 1\ \mathsf{Nil})$ fits into $?::x$. Start as usual

$$\mathsf{Cons}\ (?::y)\ (?::\mathsf{List}\ y) :: \mathsf{List}\ y$$

and now, plugging 1 in the first hole, we see $y = \mathsf{Int}$

$$\mathsf{Cons}\ 1\ (?::\mathsf{List}\ \mathsf{Int}) :: \mathsf{List}\ \mathsf{Int}$$

and yes, $\mathsf{Cons}\ 1\ \mathsf{Nil}$ fits into $?::\mathsf{List}\ \mathsf{Int}$, so

$$\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 1\ \mathsf{Nil}) :: \mathsf{List}\ \mathsf{Int}$$

[2] In fact, Haskell overloads numerical constants, and says "You can if $\mathsf{List}\ \mathsf{Int}$ is a numeric type." But it isn't. Note to self, next time, set this exercise using True, not 1. Logical constants aren't overloaded.

2.2 This Week's Mission

- Learn to play LEGO with types.
- Learn to split a pattern variable into constructor cases.
- Learn about Boolean guards.
- Learn to use Haskell's built in list type $[x]$.

The plan is to write the sorting algorithm from last week's lab, using pattern matching, Boolean guards, and Haskell's built in list type $[x]$.

2.3 The Magic Word

There's a magic word which you should shout out when you see somebody implementing *functions* for one type with the same shape as *constructors* for another. For example, if you see a functions like

```
emptyBunch   :: BunchOf x
putOneMoreIn :: x → BunchOf x → BunchOf x
```

that's a bit like Nil and Cons, isn't it? So shout out the magic word

ALGEBRA

What's so magic? Well, when you have an ALGEBRA, you can turn one thing into another. For example, if we had the above, we could write a function to make a BunchOf x from a List x.

```
listToBunch :: List x → BunchOf x
listToBunch Nil           = emptyBunch
listToBunch (Cons x xs) = putOneMoreIn x (listToBunch xs)
```

Fit 3

How do I put things *together*?

- understand how types explain the way expressions together
- figure out how to assign types to missing pieces in top-down construction

Synthesis.

building expressions; figuring out the types of the missing bits; definition and re-use (some higher-order examples?);

ghci encounters

Fit 4

Does being a thing actually *do* anything?

Strategy Trees.

Boolean strategy trees and their interpretation as functions; spatial and temporal composition; computing components; CS106 flashbacks;

an introduction to monads which doesn't mention them at all

Fit 5

Can I have *two* things at once?

Multicore Haskell.
embarrassing parallelism; threads and queues; divide and conquer; threadscope; when do we slow things down? when do we speed things up?

Bibliography

Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.