Dynamic Technology Holdings

meet the λambdas

Workshop I

Learning how to Compute . . . without a Computer!

Notes & Study Guide

Andrea Falconi
andrea.falconi@gmail.com

May, 2009

Workshop I

Learning how to Compute . . . without a Computer!

"A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine."

—Alan Turing

Goals.

- ☐ To introduce informally, through concrete examples fundamental concepts of "computing with functions".
- ☐ To start developing a taste for functional abstraction.

Topics. Functions and types. Function composition. Induction and recursion. Recursive definitions of functions. Functional evaluation model. Lazy evaluation. Importance of proof.

Description. Don't let the arcane terms above scare you away! Instead of seeking precise definitions, we will examine several practical examples through which we will get an operational grasp of those concepts. We will "visualize" functions and types, come up with an easy recipe to define things in "terms of themselves", and see how this leads to an extremely simple and clean program evaluation model — so simple that in fact all we need to work out a program's output is pencil and paper! In the process, we will note that the output of a function can become the input of another and so we have a way to compose small computations into bigger ones as if by "piping"; moreover the inputs and outputs of this pipeline can be produced lazily (on-demand), which leads to an efficient and powerful modularization mechanism. Finally, we will mention the important role that proof can (actually should!) play in programming.

Study Guide.

(1) After the workshop we will post the session notes and program examples on the wiki. Review them while the concepts are still fresh in your mind. At the same time, you should read §1 from the *Craft* which covers the topics discussed in the session.
(2) Start familiarizing yourself with Haskell. Read §1 from *RwH* and use `ghci` to run the examples yourself.
(3) Read §3 from the *Craft*. It overlaps with §1 from *RwH* and should be useful to cross-check your understanding of the concepts.
(4) Do all the exercises in §3 from the *Craft*. Write down your function definitions and use pencil & paper to compute them on some test inputs. Only when you're confident a definition works, you should run your tests in `ghci`. Are you getting the same results? If not, try to understand whether your definition is incorrect or you got wrong some of the evaluation steps you simulated on paper. How could you ensure the absolute *absence* of errors in your definitions? Is testing enough?

Our task for the day is to write a program that, given a list of persons, tells you whether or not there's at least one person who likes functional programming and has green eyes. We will first outline a possible solution in imperative land: tell the computer what to do by sequencing commands that change the state of the program data. Then, after noting how easy it is to lose our train of thought in state changes, we will start thinking of a programming world where state changes are banned and see how we could implement the same program in that ethereal setting. In the process, we will make friends with functions and types as well as induction and recursion.

Main Course. So here's our implementation in the rising star object-oriented language *Coffee@7Sharp*.[1] (Obvious details omitted from the listing.)

```
class Color
    ...
    bool isGreen()

class Person
    ...
    bool likesFP()
    Color eyeColor()

class Finder
    List<Person> whoLikesFP(List<Person> ppl)
        for (k = 0; k < ppl.size; ++k)
            Person p = ppl.get(k);
            if (!p.likesFP) ppl.remove(k);
        return ppl;

    bool someoneGreenEyed(List<Person> ppl)
        for (k = 0; k < ppl.size; ++k)
            Person p = ppl.get(k);
            if (!p.eyeColor.isGreen) return true;
        return false;
```

We have a `Color` class to represent an enumeration of colors (`Black`, `Green`, etc.) and to tell us whether the `Color` at hand is `Green`. A `Person` is created with an eye `Color` and a `bool` to tell whether or not he/she likes functional programming. The `whoLikesFP` method filters out of the list those folks that don't like functional programming, whereas `someoneGreenEyed` tells you whether at least one `Person` in the input list has green eyes.

[1]A collaboration of software giants Mega Bubbles and Fading Sun with renowned hackers from the sect of the Purple Serpent.

Side Dishes. So now given a list of `Person` objects we can tell whether someone likes functional programming and has green eyes: we call `whoLikesFP` and then `someoneGreenEyed`. Easy peasy? Unfortunately, unforseen side-effects (yes, bugs) are lying in wait. In fact, besides the usual "suspects" (yes, the code above should check for `null` pointers!), it seems that sometimes, depending on which input list we pass in, we get the wrong answer. For example, look at the following:

```
void main()
    Joe   = new Person(likesFP = true,  eyeColor = Black),
    Tom   = new Person(likesFP = false, eyeColor = Green),
    Sally = new Person(likesFP = true,  eyeColor = Green);

    List<Person> ppl = [ Joe, Tom, Sally ];

    List<Person> fpFolks = Finder.whoLikesFP(ppl);

    // Now:      ppl = [ Joe, Sally ]      fpFolks = [ Joe ]

    bool answer = Finder.someoneGreenEyed(fpFolks);

    print(answer);         // ⇝ false
```

But Sally likes functional programming and has green eyes! So why don't we get `true`? Ah, of course! The problem is that `whoLikesFP` *may change* the state of the input list and, moreover, it may do so while the list is being iterated — elements may be removed. In the example above, when the iteration reaches Tom, it also removes him from the list which now shrinks to 2 elements, but 2 happens to be the value of the iteration counter `k` and so we exit the loop before we get to see Sally. Also note that the effects of those state changes may then *propagate*, leading to more unexpected results — e.g. if we called `print(ppl)` after `whoLikesFP`, Tom would not be printed out.

So even a conceptually trivial task, like filtering out of the list persons that don't like functional programming, can easily become fairly involved to program because we also have to think about how the state changes we carry out in one place are going to affect other pieces of code. Moreover, to fully understand a piece of code we have to keep track of all the possible values of variables, objects, etc. operated on by the code at each computation step. Going back to the `whoLikesFP` method, we see that when `k` is 1 the size of `ppl` is 3, but after removing Tom the size *becomes* 2; now `k` is incremented and *becomes* 2, which is the *new* value of `ppl.size` and so the loop exits. The bottom line is that we have to consider all possible combinations of state changes, but the number of combinations is usually fairly large even for small code fragments. Invariably, things crop up that we had not seen or thought about initially and they usually lead to those unforeseen side-effects that we've come to love and call bugs.

To Be or not to Be-come

Knowing how "temperamental" imperative programs can get because of side-effects induced by state changes, we begin to wonder if there is a way to eliminate or at least contain side-effects. Well, if side-effects are caused by state changes, we could ban state changes from our programming model. In that case, we would avoid runtime "surprises" like those we've just seen and also a piece of code could always be understood in isolation as there would be nothing outside of it that can possibly affect it nor would it be possible for it to cause side-effects. This means we have to move away from seeing variables as "buckets" for values where the bucket may contain *different* values at different times.

Equal is Equal. Look back at the `for` loop in the `whoLikesFP` method above. Does the statement $k = 0$ really mean that k is *equal* to 0? Indeed just after the first loop iteration k *becomes* 1. In fact, we're using the equal sign to mean "assignment" and treating k as a "bucket" where to store a value; the state of the bucket (i.e. the contained value) may change as statements are executed. It stands to reason that to avoid state changes we have to get rid of assignments. But how can we compute without assigning values to variables? Do we even have variables?

Let's start by doing the obvious and interpret the equal sign as meaning that the expression on the left hand side is *identical* to the one on the right hand side. This means that writing something like $k = 2$ is just another way to say that, in the enclosing evaluation context, k is another name for 2 and is not possible to then give another "meaning" to k, such as $k = 11$. Likewise, once we say ppl = [Joe, Tom, Sally], ppl will always refer to that list and the list may not be changed — e.g. we can't remove Tom. Any of these statements with an equal sign in between two expressions is called an *equation* and equations are central to functional programming.

Equals by Equals. So we have equations, but how do we use them? Say we have a trivial program that just multiplies 5 by 2 and then adds 1; every time the program is run, it produces 11. We could model this situation with an equation: $\mathit{result} = 5 \times 2 + 1$. A slightly more useful program would be one where the computation is parameterized, for example $\mathit{result} = 5 \times x + 1$, so that you could pass an input number x to the program and get back that number times 5 plus 1. But what mechanism should the program use to compute the result? For example, if we start from:

$$\begin{aligned} &x = 2 \\ &\mathit{result} = 5 \times x + 1 \end{aligned}$$

how would the program arrive at 11 — i.e. the right answer? Well, it should follow the same rote procedure we would follow if computers had not been invented and all we had was a paper sheet and a pencil: *rewrite* the second equation by replacing all occurrences of x with 2 and then evaluate the resulting expression to obtain $\mathit{result} = 11$. Here's the computation, step by step:

$$\begin{aligned}
\mathit{result} &= 5 \times x + 1 \\
&= 5 \times 2 + 1 && \text{by equation } x = 2 \\
&= 10 + 1 && \text{by arithmetic} \\
&= 11 && \text{by arithmetic.}
\end{aligned}$$

So we can see computation as a plain matter of *rewriting expressions* by replacing equals with equals until the expressions are reduced to the desired result. Then a program is nothing more than an expression and running the program consists of evaluating that expression on the provided input. Although a simple model, it surprisingly scales up to programs of any size!

Variables Vary Not. Note that in functional programming we do use variables (e.g. x above), although they're quite a different beast than those found in an imperative language. In fact, the meaning of a variable in an expression is that of a placeholder for a "typical" value of a certain kind, one that makes sense in the context of the expression in which the variable appears — e.g. you could replace x with 2 or 1.456 in the previous expression, but you would not replace it with Tom, would you?

A variable may not be "assigned" different values overtime, but can be bound to a certain value for the sake of evaluating an expression. In other words, every occurrence of the variable in the expression is replaced with the given value — like in the above example, where 2 is *substituted* for x. This fact that variables are just placehodlers and may not vary[2] is essential to enable that kind of *equational reasoning* we've just seen, where equals can be replaced by equals. So we can "safely" evaluate an expression and replace it with its value to yield an equivalent program (i.e. one that produces exactly the same output for the same input) as there are no side-effects in rewriting the expression. (This fact is referred to as *referential transparency.*)

[2] In the sense that a variable once bound to a value may not be "assigned" another value while the enclosing expression is being evaluated.

We made the point that functional programming can avoid us the pain of unforeseen side-effects and simplifies our programming model.[3] Now it's time to see in more detail how computation unfolds from the program's input to the output. The key idea is to specify "recipes" to transform an input value into an output value; each recipe being defined in terms of equations whose expressions manipulate the input value to produce the output. No state changes are involved, rather we specify that to a given input value *corresponds* a specific output value. These recipes can be chained together so that the program's input value is transformed in subsequent steps, until the final output is produced. Computation proceeds by using those recipes' equations to rewrite the initial program expression until the output is produced. So let's roll up our sleeves and start rewriting our program from scratch, one little step at a time.

Basic Recipes. To begin, we specify a *rule* to *associate* a Bool to *each* Person, depending on whether they like functional programming:

$$\begin{tikzcd}
\textbf{Person} \arrow[r, "\mathsf{likesFP}"] & \textbf{Bool} \\
\text{Joe} \arrow[r] & \text{True} \\
\text{Tom} \arrow[r] & \text{False} \\
\text{Sally} \arrow[uur] &
\end{tikzcd}$$

$$\begin{aligned}
\textbf{likesFP}\ \ \text{Joe} &= \text{True} \\
\textbf{likesFP}\ \ \text{Tom} &= \text{False} \\
\textbf{likesFP}\ \ \text{Sally} &= \text{True}
\end{aligned}$$

We give the rule a name (likesFP) and specify which Bool corresponds to each Person. This concept is illustrated by the diagram on the left, whereas on the right you can see the corresponding Haskell code. As you can see, likesFP is defined by three equations each stating that the expression on the left can be replaced with the value on the right. So, for example, we can reduce the expression (likesFP Tom || likesFP Sally) to True as follows:[4]

$$\begin{aligned}
&\mathsf{likesFP}\ \text{Tom} \mid\mid \mathsf{likesFP}\ \text{Sally} \\
&\qquad = \text{False} \mid\mid \mathsf{likesFP}\ \text{Sally} && \text{by } 2^{\text{nd}} \text{ likesFP equation} \\
&\qquad = \mathsf{likesFP}\ \text{Sally} && \text{by } \mid\mid \text{ definition} \\
&\qquad = \text{True} && \text{by } 3^{\text{rd}} \text{ likesFP equation.}
\end{aligned}$$

We can repeat the same procedure to craft another rule to associate an eye Color to a Person:

[3] Indeed, it could be argued that the most attractive benefits are the ability to reason about programs and to derive programs, but we'll leave those topics for an advanced course.

[4] Yes, you guessed it, || is the Haskell "or" operator.

$$\begin{tikzcd}
\textbf{Person} \arrow[r, "eyeColor"] & \textbf{Color} \\
\text{Joe} \arrow[r] & \text{Black} \\
\text{Tom} \arrow[r] & \text{Green} \\
\text{Sally} \arrow[ru] & \text{Orange} \\
 & \dots
\end{tikzcd}$$

$$\begin{aligned}
&\textbf{eyeColor}\ \text{Joe} &&= \text{Black} \\
&\textbf{eyeColor}\ \text{Tom} &&= \text{Green} \\
&\textbf{eyeColor}\ \text{Sally} &&= \text{Green}
\end{aligned}$$

A small variation on the theme above is the following rule to tell us whether or not a color is Green:

$$\begin{tikzcd}
\textbf{Color} \arrow[r, "isGreen"] & \textbf{Bool} \\
\text{Green} \arrow[r] & \text{True} \\
\text{Black} \arrow[r] & \text{False} \\
\text{Orange} \arrow[ru] & \\
\dots \arrow[ruu, dashed] &
\end{tikzcd}$$

$$\begin{aligned}
&\textbf{isGreen}\ \text{Green} &&= \text{True} \\
&\textbf{isGreen}\ \_ &&= \text{False}
\end{aligned}$$

As you can see the rule associates True to Green and False to any other Color; in Haskell you can use the underscore character as a wildcard to “match” any other value that you haven’t specified in the previous equations.

The Function Club. You’ve probably already noticed that we’ve done the same thing over and over again to specify a rule: given an input value, decide what is the corresponding output value; every time we specified, by means of equations, an input/output transformation that to *each* given input, associated *one and only one* definite output. Should we give this construction a snappy name? How about . . . yes, *function*!

All the functions defined above use an equation for each input/output association that defines the function. In other words, we enumerated all possible associations. This is not always practical or even possible, especially if we have a very large number of input values to cater for. A simple example can make this evident; let’s say we want to define a function isOdd to tell us whether or not an integer is odd and let’s try using the enumeration approach:

$$\begin{aligned}
&\text{isOdd}\ 0 &&= \text{False} \\
&\text{isOdd}\ 1 &&= \text{True} \\
&\text{isOdd}\ 2 &&= \text{False} \\
&\text{isOdd}\ 3 &&= \text{True} \\
&\dots
\end{aligned}$$

As you can imagine, we’re not going to have an easy time if we go down this road. However, as you’ve probably guessed already, there’s a simple way out: write down how the function operates on a typical input value. Not surprisingly, like most languages, Haskell too has equality and modulo operators; so we can easily redefine isOdd as

$$\textbf{isOdd}\ x = (x\ `\textbf{mod}`\ 2) == 1$$

where x is a name that stands for a typical input value — or function *argument*, if you prefer. Note that conceptually isOdd does not differ from the functions we defined previously, it's still a rule to associate an input value to an output value; it's just that we've figured out a more succinct way of defining the rule — i.e. we're not explicitly enumerating all possible associations, but using parameterized expressions instead. In particular, the procedure to evaluate the function on a given input is still the same: use expression rewriting to obtain the result — obviously we have to substitute the placeholder argument with the actual input value. Here's an example then:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{isOdd}\ 4} \\
= & (4\ \text{`}\mathsf{mod}\text{`}\ 2) == 1 & \text{by substitution} \\
= & 0 == 1 & \text{by } \mathsf{mod} \text{ definition} \\
= & \text{False} & \text{by } == \text{ definition.}
\end{array}
$$

Who's Your Type. We've said that a function transforms an input value into an output value, but where do these values come from? Can we use *any* value? Obviously not; for example what is the meaning of an expression like (isOdd Tom)? Yes, that would be a real odd thing to do indeed to pass Tom as an argument to the isOdd function because the function is defined in terms of computations that involve integers, not persons. So to avoid non-sense, when we define a function, we also need to specify what is the collection of values that can be used as inputs and to which collection of values the produced outputs belong.[5]

In functional programming, we refer to a collection of values as a *type* and we give it a name. Indeed, if you look at the function diagrams above, we've implicitly used this concept of type in our definitions as we were "naturally" grouping inputs and outputs when defining functions and we even gave them names: Person, Color, and Bool. Haskell provides several built-in types such as Bool (whose only two values are True and False) and Integer (whose values are the integer numbers). Of course, you can also define your own types such as Person and Color, which for the purposes of our program we define as simple enumerations[6]

```
data Person = Joe   | Tom   | Sally
data Color  = Black | Green | Orange | Yellow | Blue | White
```

so that Person is a collection with three values (Joe, Tom, and Sally) whereas Color has six. In Haskell you state that a value belongs to a type by using two semicolons like so: Joe :: Person, which we read as "Joe has type Person". To state that a function f transforms input values of type α into output values of type β, we provide a function's *type signature* like so: $\mathsf{f} :: \alpha \to \beta$. (We use the arrow for the sake of pretty printing; however, when using a normal text editor you would have to enter the two characters `->`.) The type signatures of all the functions we've defined so far are:

[5] Haskell can infer those types for you and do compile time checking, so in practice you don't have to specify the types of the function's inputs and outputs. From a conceptual standpoint however, for a function to be well defined, you have specify its input/output types.

[6] There are of course much more sophisticated ways to define your own types, but we will not need to use these constructions for now.

likesFP	::	Person	→	Bool	**eyeColor**	::	Person	→	Color
isGreen	::	Color	→	Bool	**isOdd**	::	Integer	→	Bool

In conclusion, to define a function we have to:

- ☐ Decide what is the type of the inputs the function will operate on and what is the type of the produced outputs — i.e. establish the function's type signature.
- ☐ Come up with a rule to associate each input to a unique output — this rule is defined in terms of equations and often we give a generic way to transform a typical input into an output so to avoid enumerating all possible input/output associations.

No Country for Old Method. At this point you might be wondering if a function is the same as method in an imperative language; after all, they both take in typed inputs and deliver typed outputs. However, there's a subtle (at first sight) difference that makes the two extremely diverse species: a function *always* returns the same output for the same input, a method may not. Moreover, evaluating a function has no side-effects, whereas a method call may (and often does) change the state of the program. Here's a simple example to clarify these points: say we're programming a video game and have a class to track if a character has entered the scene at least once. You bet, we're writing it in our favorite object-oriented language, *Coffee@7Sharp*.

```
class Tracker
    Person who;
    bool   seen = false;

    bool seen(Person character)
        if (character == who)   seen = true;
        return seen;
```

You create an instance of the `Tracker` with a specific Person to track and the `seen` field is initially set to `false`. As characters enter the scene, you pass them in to the `seen` method, which sets the `seen` field to `true` when the character to track is passed in. Simple enough. Now, look at this code snippet:

```
spotJoe = new Tracker(who = Joe);
// Now:  spotJoe.seen = false

x = spotJoe.seen(Sally);   // x = false
spotJoe.seen(Joe);         // Now:  spotJoe.seen = true
y = spotJoe.seen(Sally);   // y = true

z = x || y;                // z = true
```

The `seen` method returns both `false` and `true` for the same argument, Sally. Hence, it can't be a function as to Sally should correspond *exactly one* value. Moreover, the method call changes the state of the `spotJoe` object, which means after the call we have a program that is potentially not equivalent to the one before the call — i.e. may return different results for the same input as is the case for Sally. (In short, we lose referential transparency.) The implication is that, in imperative land, we can't just happily rewrite expressions by replacing equals with equals — it won't work in general, we have to take into account all state changes as well. Looking at the snippet above, we can't say that $x = y$ (just because their right hand side is the same) and therefore the expression $z = x \parallel y$ can be simplified to $z = x$, yielding $z = \mathit{false}$. On the other hand, this line of reasoning is perfectly valid in (pure) functional land and can be used to get rid of unnecessary computation steps — to conclude that $z = \mathit{false}$, we don't need to evaluate y (which means we're sparing a function call) nor do we need to evaluate the "or" expression (so we're sparing another call).

In imperative land, we're allowed to make unrestricted use of mutable data and I/O in our definitions; so, in general, there are no guarantees that a method/procedure/etc. be a faithful representation of a mathematical function. On the other hand, in the (pure) functional world we've described so far,[7] a function really is a mathematical function.

Cooking a Meal. If you look at the $\mathit{eyeColor}$ and $\mathit{isGreen}$ functions we defined earlier on, you see that we could take a Person (Tom say) and use the $\mathit{eyeColor}$ function to get his eye Color (Green); then we could pass on this output to $\mathit{isGreen}$ which would tell us, well, if the Color is Green (True). Look below:

$$\begin{tikzcd}
\textbf{Person} \arrow[r, "\mathit{eyeColor}"] & \textbf{Color} \arrow[r, "\mathit{isGreen}"] & \textbf{Bool} \\
\text{Joe} \arrow[r] & \text{Black} \arrow[dr] & \text{True} \\
\text{Tom} \arrow[r] & \text{Green} \arrow[ur] & \text{False} \\
\text{Sally} \arrow[ur] & \text{Orange} & \\
 & \dots &
\end{tikzcd}$$

Just by following the arrows we realize that the trick works really for any Person (not just Tom) and so we've defined a new function that associates True or False to each Person, depending on whether or not they have green eyes. Here's how we could define it in Haskell:

```
hasGreenEyes Joe   = isGreen (eyeColor Joe)
hasGreenEyes Tom   = isGreen (eyeColor Tom)
hasGreenEyes Sally = isGreen (eyeColor Sally)
```

[7] In our workshops we'll only be studying concepts pertaining to purely functional languages; so we'll keep on using (as we've tacitly done so far) the word functional with the meaning of "purely functional".

If we evaluate ($\mathsf{hasGreenEyes}$ Joe), we get False:

$$\begin{aligned}
&\mathsf{hasGreenEyes}\ \text{Joe} \\
&\qquad = \mathsf{isGreen}(\mathsf{eyeColor}\ \text{Joe}) && \text{by } 1^{st}\ \mathsf{hasGreenEyes} \text{ equation} \\
&\qquad = \mathsf{isGreen}\ \text{Black} && \text{by } 1^{st}\ \mathsf{eyeColor} \text{ equation} \\
&\qquad = \text{False} && \text{by } 2^{nd}\ \mathsf{isGreen} \text{ equation.}
\end{aligned}$$

This way of chaining the two functions together worked because the output type of $\mathsf{eyeColor}$ is the *same* as the input type of $\mathsf{isGreen}$ — i.e. Color type. In other words, $\mathsf{eyeColor}$ produces a value that $\mathsf{isGreen}$ knows how to handle. Obviously, if we try to chain the two functions the other way around, we get nonsense. For example, we could take a Color (say Black) and evaluate $\mathsf{isGreen}$ on it to obtain a Bool value (False); however, now we can't pass on this value to $\mathsf{eyeColor}$ ($\mathsf{eyeColor}$ False) as the function is defined for Person inputs, not Bool values. Also, this chaining procedure is not specific to $\mathsf{eyeColor}$ and $\mathsf{isGreen}$; in fact, we could just as well combine isOdd above with the function f that doubles its argument and then adds one to it:

```
f :: Integer → Integer
f x  =  2*x + 1

isOddF :: Integer → Bool
isOddF x  =  isOdd (f x)
```

$$\begin{tikzcd}
\text{Integer} \arrow[d, "\mathsf{f}"] & \textbf{Integer} \arrow[dd, "\mathsf{isOddF}"] \\
\text{Integer} \arrow[d, "\mathsf{isOdd}"] & \\
\text{Bool} & \textbf{Bool}
\end{tikzcd}$$

If we evaluate (isOddF 5), we get True:

$$\begin{aligned}
&\mathsf{isOddF}\ 5 \\
&\qquad = \quad \mathsf{isOdd}\ (\mathsf{f}\ 5) && \text{by substitution} \\
&\qquad = \quad \mathsf{isOdd}\ (2 * 5 + 1) && \text{by substitution} \\
&\qquad = \quad \mathsf{isOdd}\ 11 && \text{by arithmetic} \\
&\qquad = \quad (11\ `\mathsf{mod}`\ 2) == 1 && \text{by substitution} \\
&\qquad = \quad 1 == 1 && \text{by } \mathsf{mod} \text{ definition} \\
&\qquad = \quad \text{True} && \text{by} == \text{definition.}
\end{aligned}$$

Surely, in general we can repeat this procedure for any other two functions, provided that the output type of the one is the same as the input type of the other. Specifically, whenever we have two functions

$$f :: \alpha \to \beta \qquad g :: \beta \to \gamma$$

where α, β, and γ stand for arbitrary types such as Integer, Bool, or Person, we can always define a new one

$$\begin{aligned} &h :: \alpha \to \gamma \\ &h\ x\ =\ g\ (f\ x) \end{aligned}$$

This new function h is called the *composite* of g with f; the procedure we followed to obtain it is referred to as *function composition*. Haskell provides a function composition operator, the dot '.' character — for the sake of pretty-printing, we will use a small circle $\circ$ in place of the dot. We can use this operator to make the definition of a composite function a little shorter as follows: $h\ =\ g \circ f$. (We can read it as "g after f", "g following f", "g composed with f", or just "g of f".) We can use this operator to rewrite the composite functions we defined earlier:

$$\mathbf{hasGreenEyes}\ =\ \mathbf{isGreen}\ \circ\ \mathbf{eyeColor}$$

$$\mathbf{isOddF}\ =\ \mathbf{isOdd}\ \circ\ \mathbf{f}$$

So we always follow this recipe: put the name of the composite function on the left hand side of the equation; on the right side, put the second function in the chain, followed by the $\circ$ operator, followed by the first function in the chain. As you can imagine, we can chain as many functions as we like using composition — of course, provided that input/output types match. You can think of it as a processing pipeline where each processing step is represented by a function and the composition operator acts as a pipe in between functions, collecting the output of one function and passing it on as an input to the next function in the pipeline. If the first function in the pipeline is f_1, the second is f_2, and so on, we can define a $pipeline$ function as follows:

$$pipeline\ =\ \mathbf{f}_n\ \circ\ \cdots\ \circ\ \mathbf{f}_2\ \circ\ \mathbf{f}_1$$

This gives us a simple, yet extremely effective mechanism to chain our recipes (functions) together into a bigger one for an entire "meal" (program) and, at the same time, allows us to decompose the task of cooking the meal into smaller parts (functions).

Chasing the Tail

Functions and function composition sound like a very interesting tool to build processing pipelines. Let's put all this to good use right away then to assemble a pipeline that we can feed with a list of persons to detect whether someone has green eyes — recall the `someoneGreenEyed` method in our *Coffee@7Sharp* implementation. The idea is to decompose the computation in small, very focused steps, each carried out by a function; this way we can concentrate on one small and well defined task at a time (i.e. function definition) and then put them all together to get the desired result — so each function should just "do one thing and do it well". To your pipe wrenches, Sirs.

Note about Notation. Before looking at our plan of action, let's quickly discuss some handy notation we'll be using for lists. First off, Haskell lets you specify a list of values by enumerating them within square brackets: [1, 2, 3] is a list of three Integer values and [Tom, Sally] is a list of two Person values. A list with no elements in it is called the *empty list* and denoted by [ ]. As an alternative to the above notation, you can specify a list by separating the values with a colon and terminate the expression with the empty list — e.g. the two lists above can be equally written as 1:2:3:[ ] and Tom:Sally:[ ]. You can also mix the two notations as long as you keep the square brackets to the right — e.g. Joe:[Tom, Sally] is the same as [Joe, Tom, Sally]. Each list is itself a value; to which type does it belong then? Well, if you have any type, say Person, you can build a new list type out of it (denoted by [Person]) and its values are the lists of values of that type — e.g. [Joe, Sally]::[Person], [Joe, Tom]::[Person], [1]::[Integer], [1, 2]::[Integer].

Plan of Action. As a picture is worth a thousand words, here's one to describe the simple idea behind our plan for implementing the processing pipeline:

$$\begin{tikzcd}
{[\text{Joe, Tom, Sally}]} \arrow[d, "\mathsf{mapEyeColor}"] & \\
{[\mathit{eyeColor}\ \text{Joe}, \mathit{eyeColor}\ \text{Tom}, \mathit{eyeColor}\ \text{Sally}]} \arrow[d, "\mathsf{mapIsGreen}"] & = [\text{Black, Green, Green}] \\
{[\mathit{isGreen}\ \text{Black}, \mathit{isGreen}\ \text{Green}, \mathit{isGreen}\ \text{Green}]} \arrow[d, "\mathsf{foldOr}"] & = [\text{False, True, True}] \\
\text{False} \mid\mid \text{True} \mid\mid \text{True} & = \text{True}
\end{tikzcd}$$

$$\textbf{someoneGreenEyed} = \textbf{foldOr} \circ \textbf{mapIsGreen} \circ \textbf{mapEyeColor}$$

If we start from a list of Person values and want to find out whether someone is green eyed, the first obvious thing to do is to figure out the eye Color of each Person in the list. We can easily do this by applying the $\mathsf{eyeColor}$ function to each Person in the list and so we end up with a list of Color values. Then, we need to tell which ones are Green; again we can apply the $\mathsf{isGreen}$ function to each Color in the list to get a list of Bool values. If at least one of these is True, then we know that at least one Color must have been Green and so at least one Person must be green eyed. By the same token, if all values turn out to be False, then no Person in the pipeline's input list can be green eyed. Hence, we can use the following functions to break down the computation and then assemble the pipeline:

- ☐ $\mathsf{mapEyeColor} :: [\text{Person}] \to [\text{Color}]$ associates a Person list to a list where each Person is replaced with their eye Color.
- ☐ $\mathsf{mapIsGreen} :: [\text{Color}] \to [\text{Bool}]$ associates a Color list to a list where each Color c is replaced with the value of $\mathsf{isGreen}\ \mathsf{c}$.
- ☐ $\mathsf{foldOr} :: [\text{Bool}] \to \text{Bool}$ tells whether at least one Bool in the input list is True; this is easily done by “folding” the || operator into the list.

Turn ’em All. To implement the $\mathsf{mapEyeColor}$ function, we have to find a way to turn each Person in the input list into a Color by using the $\mathsf{eyeColor}$ function. We need a way to describe the kind of associations pictured below:

$$\begin{tikzcd}
\textbf{[Person]} \arrow[r, "\mathsf{mapEyeColor}"] & \textbf{[Color]} \\
[\ ] \arrow[r] & [\ ] \\
[\text{Joe}] \arrow[r] & [\text{Black}] \\
[\text{Tom, Joe}] \arrow[r] & [\text{Green, Black}] \\
[\ldots] \arrow[r, dashed] & [\ldots]
\end{tikzcd}$$

We could try to define the function by enumeration:

$$\begin{aligned}
\mathsf{mapEyeColor}\ [\text{Joe}] &= [\text{Black}] \\
\mathsf{mapEyeColor}\ [\text{Tom, Joe}] &= [\text{Green, Black}] \\
&\ \ \cdots
\end{aligned}$$

However, we wouldn’t go very far as there are too many input lists to enumerate; we need a cunning trick, but how do we pull it out of our hat? Since all we can do is rewrite expressions, maybe we should try some rewrites and see if we can get some ideas from the examples. Let’s start from the most simple case we can possibly think of: the empty list. This is trivial as there are no values to map and so it goes without saying that the result should be the empty list too.

$$\mathsf{mapEyeColor}\ [\ ] = [\ ]$$

If we consider a one-value list, we also know what the function should return:

$$\mathsf{mapEyeColor}\ [\text{Joe}] = [\mathsf{eyeColor}\ \text{Joe}]$$

Now the right hand side can be seen as the empty list with ($\mathsf{eyeColor}$ Joe) added to the front, hence we can rewrite it as $[\mathsf{eyeColor}\ \text{Joe}] = \mathsf{eyeColor}\ \text{Joe} : [\,]$. But we know that ($\mathsf{mapEyeColor}\ [\,] = [\,]$) and so we can replace the empty list with ($\mathsf{mapEyeColor}\ [\,]$). All together we have:

$$\begin{array}{lll}
\mathsf{mapEyeColor}\ [\text{Joe}] & & \\
 & = [\mathsf{eyeColor}\ \text{Joe}] & \text{desired output} \\
 & = \mathsf{eyeColor}\ \text{Joe} : [\,] & \text{by list notation} \\
 & = \mathsf{eyeColor}\ \text{Joe} : \mathsf{mapEyeColor}\ [\,] & \text{by def on } [\,].
\end{array}$$

Let's see if we can do something similar with a two-value list; so we add Tom to the front of [Joe] and try to repeat what we've done above. In this case too, it's pretty clear what the function should deliver:

$$\mathsf{mapEyeColor}\ [\text{Tom, Joe}] = [\mathsf{eyeColor}\ \text{Tom}, \mathsf{eyeColor}\ \text{Joe}]$$

The right hand side can be seen as the list [$\mathsf{eyeColor}$ Joe] to which the value ($\mathsf{eyeColor}$ Tom) has been added. Also we know that $\mathsf{mapEyeColor}\ [\text{Joe}] = [\mathsf{eyeColor}\ \text{Joe}]$. If we combine all these facts together, we get

$$\begin{array}{lll}
\mathsf{mapEyeColor}\ [\text{Tom, Joe}] & & \\
 & = [\mathsf{eyeColor}\ \text{Tom}, \mathsf{eyeColor}\ \text{Joe}] & \text{desired output} \\
 & = \mathsf{eyeColor}\ \text{Tom} : [\mathsf{eyeColor}\ \text{Joe}] & \text{by list notation} \\
 & = \mathsf{eyeColor}\ \text{Tom} : \mathsf{mapEyeColor}\ [\ \text{Joe}] & \text{by def on [Joe].}
\end{array}$$

It definitely looks promising and, in fact, if we add yet another Person to the front of the list and repeat the procedure, we see that a pattern begins to emerge:

$$\begin{array}{ll}
\mathsf{mapEyeColor}\ [\text{Sally, Tom, Joe}] & \\
 & = [\mathsf{eyeColor}\ \text{Sally}, \mathsf{eyeColor}\ \text{Tom}, \mathsf{eyeColor}\ \text{Joe}] \\
 & = \mathsf{eyeColor}\ \text{Sally} : [\mathsf{eyeColor}\ \text{Tom}, \mathsf{eyeColor}\ \text{Joe}] \\
 & = \mathsf{eyeColor}\ \text{Sally} : \mathsf{mapEyeColor}\ [\text{Tom, Joe}]
\end{array}$$

Ah! This is like chasing the tail of the input list until we get to the bottom, i.e. the empty list, and have converted all input values in the process. Specifically, what we've done on non-empty lists is:

- ☐ Take the head of the input list and apply $\mathsf{eyeColor}$ to it; the resulting value is going to be the head of the result list.
- ☐ The tail of the result list is given by computing $\mathsf{mapEyeColor}$ on the tail of the input list.

So if we let x stand for the head of any non-empty Person list and xs for the tail, we can generalize our definition as follows:

```
mapEyeColor [ ]      = [ ]
mapEyeColor (x : xs) = eyeColor x : mapEyeColor xs
```

which gives a simple recipe to compute the result of mapEyeColor on a given input list; in words, to evaluate mapEyeColor on a list ppl of Person values, do the following:

(1) if ppl is empty, the result is the empty list too;
(2) otherwise evaluate eyeColor on the head of ppl and add the result to the front of the list obtained by repeating this procedure on the tail of ppl.

Note that mapEyeColor is defined in terms of itself (second equation); when this happens we say that the definition is *recursive*. Such definitions imply that to compute the output you have to keep on rewriting the function itself on a subset of the input list. Of course, this rewriting should eventually come to an end, which happens when the subset of the input list we're considering has no elements left — i.e. is empty. We can clearly see how this works if we evaluate the function.

```
mapEyeColor [Sally, Tom, Joe]
 = mapEyeColor (Sally : [Tom, Joe])
 = eyeColor Sally : mapEyeColor [Tom, Joe]
 = eyeColor Sally : mapEyeColor (Tom : [Joe])
 = eyeColor Sally : eyeColor Tom : mapEyeColor [Joe]
 = eyeColor Sally : eyeColor Tom : mapEyeColor (Joe : [ ])
 = eyeColor Sally : eyeColor Tom : eyeColor Joe : mapEyeColor [ ]
 = eyeColor Sally : eyeColor Tom : eyeColor Joe : [ ]
 = [eyeColor Sally, eyeColor Tom, eyeColor Joe]
```

At this point it shouldn't be hard to see that we can go through exactly the same process to craft the definition of mapIsGreen, which you can see below.

```
mapIsGreen [ ]      = [ ]
mapIsGreen (x : xs) = isGreen x : mapIsGreen xs
```

Squeeze 'em All. The last hurdle ahead of us is the definition of foldOr. This function is supposed to take a Bool list and return True if and only if at least one value is True; if none is True, then False should be returned. We can see from the picture above that an easy way of achieving this is to fold the || operator in between each element of the list so that, when evaluated, the expression "squeezes" all the list values into one. How do we express that though? Well, we've seen that going through a few concrete examples may help to find the way, so we'll try to follow the same line of reasoning as above.

The first example input we consider is the empty list. In this case there's no element which can possibly be True as the list is empty! Hence, we should return False.

$$\texttt{foldOr}\ [\,] \;=\; \text{False}$$

Now we move on to consider a list with just one Bool, for example [True]. Evidently in this case we should return True, so `foldOr` [True] = True. By looking at what we did earlier on, our guess is that we should try to rewrite the right hand side in such a way that `foldOr` appears there too; that would give us a mechanism to process in turn every element of the input list by means of successive, recursive rewrites.

`foldOr` [True]		
	= True	desired output
	= True || False	by || def
	= True || `foldOr` [ ]	by def on [ ].

Umm ... the trick is getting old quickly! We have a sneaky suspicion that if we try this for a two-value list, we're gonna end up with something very similar.

`foldOr` [False, True]		
	= True	desired output
	= False || True	by || def
	= False || `foldOr` [True]	by def on [True].

Also, even if add yet another value to the front of the list, the trick still works.

`foldOr` [True, False, True]	
= True	desired output
= True || True	by || def
= True || `foldOr` [False, True]	by def on [False, True].

We have a situation pretty similar to `mapEyeColor` in that we're still chasing the tail of the input list until we get to the empty list. However, in this case we're not producing a list but rather a single value, which is obtained by "or-ing" the head of the input list with the Bool value resulting from evaluating `foldOr` on the tail of the input list. To generalize this procedure to any list, we let `x` stand for the head of a non-empty Bool list and `xs` for the tail, so that we easily arrive at:

```
foldOr  [  ]     = False
foldOr  (x:xs)   = x  ||  foldOr  xs
```

In plain English our recipe says that to evaluate `foldOr` on a list `bools` of Bool values, we do the following:

(1) if `bools` is empty, the result is False;
(2) otherwise we take the head of `bools` and "or" it with the Bool value obtained by repeating this procedure on the tail of `bools`.

Again, we have a recursive definition as `foldOr` appears on both sides of the second equation above, which means that to compute the output given a non-empty list, we have to keep on rewriting the function on the tail of the input until the tail becomes the empty list, in which case the first equation says we should stop and return False. Of course, the best way to see this is by evaluating the function.

$$
\begin{aligned}
&\texttt{foldOr}\ [\text{False}, \text{True}, \text{False}] \\
&\qquad = \texttt{foldOr}\ (\text{False} : [\text{True}, \text{False}]) \\
&\qquad = \text{False} \mathbin{||} \texttt{foldOr}\ [\text{True}, \text{False}] \\
&\qquad = \text{False} \mathbin{||} \texttt{foldOr}\ (\text{True} : [\text{False}]) \\
&\qquad = \text{False} \mathbin{||} \text{True} \mathbin{||} \texttt{foldOr}\ [\text{False}] \\
&\qquad = \text{False} \mathbin{||} \text{True} \mathbin{||} \texttt{foldOr}\ (\text{False} : [\,]) \\
&\qquad = \text{False} \mathbin{||} \text{True} \mathbin{||} \text{False} \mathbin{||} \texttt{foldOr}\ [\,] \\
&\qquad = \text{False} \mathbin{||} \text{True} \mathbin{||} \text{False} \mathbin{||} \text{False} \\
&\qquad = \text{True}
\end{aligned}
$$

The Domino Effect

Looking back at the recursive functions we've defined so far, we see that conceptually we used the same procedure to arrive at their definition. First, decide what to do with the empty list — i.e. declare which output value the function associates to the empty list. For example $\mathsf{mapEyeColor}\ [\,] = [\,]$, whereas $\mathsf{foldOr}\ [\,] = \text{False}$. Then consider an arbitrary non-empty input list; use its head x and its tail xs to write an expression involving x and the value obtained by computing the function on xs. For example, $\mathsf{mapEyeColor}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{eyeColor}\ \mathsf{x} : \mathsf{mapEyeColor}\ \mathsf{xs}$ and $\mathsf{foldOr}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{x} \mathbin{\|} \mathsf{foldOr}\ \mathsf{xs}$. Surely we can factor this out in a "template recipe" to define recursive functions over lists; if f is the name of the function we want to define, then we need to give two equations as follows:

$$\begin{aligned} \mathsf{f}\ [\,] &= \text{some atomic value} \\ \mathsf{f}\ (\mathsf{x} : \mathsf{xs}) &= \text{some terminating expression involving } \mathsf{x} \text{ and } \mathsf{f}\ \mathsf{xs} \end{aligned}$$

where an atomic value is one on which no further computation can be done (e.g. [ ], [Tom, Sally], False, 3, etc.) and a terminating expression is one that yields an atomic value in a finite number of steps — like adding a value to the front of a list, "or-ing" two values, etc. More general patterns are possible, but we'll stick to the above for the moment.

Rest Assured. Although it seems a perfectly reasonable template to use, we have to clear some loose ends. First off, does the procedure even yield a function? And if so, does it yield just one or many? We sort of convinced ourselves through the examples that the template actually works in that everytime we associated to an input list a definite output value — a list in the case of the $\mathsf{map}*$ functions and a Bool in the case of foldOr. However, we only "checked" a few inputs, what *assurance* do we have that it will work for *all* inputs? It would be impossible to test the defintions on every possible input list as there is an *infinite* number of those. Moreover, here we're talking about further generalizing the procedure beyond the $\mathsf{map}*$ and foldOr functions to define *arbitrary* recursive functions on *arbitrary* lists.

Another issue is that of termination; we noticed already that because the same function appears on both sides of the second equation, we'll have to keep on rewriting the same expression when the function is evaluated. What guarantees do we have that this process will *always* come to an end? In other words, how do we rule out once and for all the possibility that we might end up in an infinite loop where we keep on rewriting forever? Intuitively we have a feel that this is not possible because every time we do a rewrite we also "consume" (taking a finite number of steps) the head of the input list and so we pass a smaller list on to f for the next evaluation. Eventually the input list will shrink to the empty list at which point the first equation tells us to return a specific value and so we stop rewriting. But how do we *prove* it?

As the majority of the definitions we're going to give are based on the template above (in fact we'll do this so often that it'll soon become a rote procedure), we would like to be *absolutely certain* that the mechanism works out of the box. If a flaw (bug!) were discovered two years down the line, we'd be in a rather interesting situation where we'd have to potentially fix a large amount of code! (Yes, many, many sleepless nights!) However, there's actually a *mathematical proof*[8] that shows the template is sound. (Yes, we can rest assured.) So we can see that proof can be a terrific weapon in our programming arsenal: it can give us assurance that the code will work as expected — i.e. no bugs! One of the reasons the functional paradigm is so attractive is that it makes it easy to reason mathematically about code, something not exactly trivial to do in imperative land.

Knocking Down the Dominoes. Without having to venture in the deep waters of pure maths, we can still get a fairly good appreciation of the line of reasoning we'd use to prove the above claims about soundness and termination. In turn, this will give us a further insight into the mechanism to craft recursive definitions. All we need to do really is make explicit something we've taken for granted every time we moved from the concrete examples to the general case in the $\mathsf{map}{*}$ and foldOr definitions: we implicitly *assumed* that rewriting the expression on the tail of the input list would yield the expected result — as it was the case for the concrete definitions we initially gave on some specific lists. In fact, in letting

$$\mathsf{foldOr}\ (\mathsf{x} : \mathsf{xs}) = \ \mathsf{x} \ || \ \mathsf{foldOr}\ \mathsf{xs}$$

the implicit assumption is that foldOr will operate correctly on xs (i.e. will return True if and only if at least one value in xs is True); the assumption is justified by the fact that indeed this is what foldOr does in the case of the empty list.[9] (A similar argument goes for $\mathsf{mapEyeColor}$ and $\mathsf{mapIsGreen}$.) This is like knocking down a chain of dominoes; if we are able to knock the first one down (give the correct defintion of f on the empty list) and assume that every piece in the chain will fall (assume f "works" on any list xs), then if we append another piece to the chain (consider $\mathsf{x} : \mathsf{xs}$) this will also have to fall, provided we put it in a suitable position with respect to the last piece (give the correct definition of f on $\mathsf{x} : \mathsf{xs}$ in terms of x and $\mathsf{f}\ \mathsf{xs}$).

Indeed, we can easily convince ourselves that this line of reasoning works whenever we have a "chain" of arbitrary things whereby we can get hold of the first one and we know how to "move" to the next thing in the chain. To clarify, consider a bunch of people holding their hands to form a chain:

$$\text{Joe} \longrightarrow \text{Sally} \longrightarrow \text{Tom} \longrightarrow \text{Jenny} \longrightarrow \ldots$$

We know that Joe is the first person in the chain. To tell who follows Joe, you look at who's holding his right hand and see Sally; to tell who follows Sally, you look at who's holding her right hand and see Tom ... well, hopefully you got the hang of it! So given any person we can always tell who follows them. Now for the interesting bit. Let's say we're given these two pieces of information:

[8] See for example §1.2, §4.2 in [1] — not for the faint-harted though!

[9] And, of course, we also managed to define it on three concrete lists ([True], [False, True], [True, False, True]) in such a way that it still worked correctly, which is what led us to make the assumption.

(1) Joe, the first person in the chain, is a programmer.
(2) However you pick a person in the chain, who follows that person is in the same job as her/him.

If asked whether everybody in the chain is a programmer then, after a little thought, we'd probably say yes. What is interesting though is the train of thought we'd follow to get to the conclusion. In fact, we'd probably start from the beginning of the chain and consider Joe, whom we know by (1) to be a programmer. The person following Joe is Sally, whom we know by (2) to be in the same job as Joe; therefore, she must be a programmer too. Tom follows Sally and so by (2) he's in the same job as her; therefore Tom is a programmer too. We can obviously carry on this process until we see that everybody is actually a programmer.[10] However, if we had $10,000$ people in the chain, we'd probably stop after the first few persons and notice that the property of being a programmer is passed down from Joe to every person in the chain checked so far because of (2). Then it seems a perfectly reasonable assumption to make that this process of passing down the property would carry on until we reach the last person in the chain; making the assumption allows us to arrive at the conclusion without having to check each and every person in the chain. In other words, to get to the conclusion we only need to check that Joe is a programmer and that we have a way to pass down this property to every person coming after him, that is if we assume that a person `p` is a programmer, then we can verify that the person following `p` is a programmer too — which we did by using (2).

We can dub this way of reasoning as *induction principle* and summarize it as follows:[11]

- ☐ We have a chain of things (e.g. chain of people above) whereby we know which is the first one (e.g. Joe) and, given a certain thing `t` in the chain, we can get to the thing following `t` (e.g. the person following `t` is the one holding `t`'s right hand).
- ☐ We know a certain property `P` holds true for the first thing in the chain. (E.g. Joe is a programmer.)
- ☐ If we assume that `P` holds for an arbitrary thing `t` in the chain, then we can verify that it also holds for the thing following `t`. (E.g. if a person `t` is a programmer, then the person following `t` must be a programmer too.)
- ☐ Given all the above, we can conclude that all the things in the chain enjoy the property `P`. (E.g. everybody is a programmer.)

Now consider all the lists having values of a given type — e.g. all the lists of Bool values. We can see easily chain them by always considering the empty list to be the first thing in the chain and if we have a list `xs` we construct one following it by adding a new value `x` to its head — i.e. `x : xs`. (For example, $\text{True} : [\text{False}, \text{True}]$

[10] Tacitly assuming there are a finite number of people in the chain.
[11] Indeed induction can come in different flavors and is not exactly formulated this way; however, we're just trying to get the hang of it here!

follows [False, True].) This means we can readily recast the induction principle in terms of lists, which in this case we dub as *structural induction over lists*:[12]

- ☐ We consider all the lists of type [α], α being a given type (e.g. Bool).
- ☐ We know a certain property P holds true for the empty list, [ ].
- ☐ If we assume that P holds for an arbitrary list xs, then we can verify that it also holds for (x : xs), x being arbitrary too.
- ☐ Given all the above, we can conclude that all the lists of type [α] enjoy the property P.

Finally, we can clearly see how the template for recursive definitions we gave above rests on induction. In fact, we start from the empty list to give a suitable function definition (the property to pass down) and then we basically set up a mechanism to pass the definition down to arbitrary lists. Look at `foldOr` again; we want to define it in such a way that the function returns True if and only if the input list contains at least one True value. This is the function specification and is obviously the property we want to pass down, that is we want to define `foldOr` so that it behaves according to this spec on any list. We know that if we let `foldOr` [ ] = False, then the function sticks to the spec. So we assume it behaves correctly on an arbitrary list xs of Bool values and we consider a generic list following xs, which we do by adding an arbitrary Bool value x to the front of the list. Now if x is True we want `foldOr` (x : xs) = True, otherwise we're happy with whatever Bool value (`foldOr` xs) turns out to be as we assumed it to be the right one. This obviously means that if we let `foldOr` (x : xs) = x || `foldOr` xs, the function still behaves according to the spec on (x : xs), that is an arbitrary list following xs. Then we may conclude that the function actually behaves according to the spec on *any* list of Bool values. Rejoice, we've just produced our first (informal) *proof of correctness*! Another way to look at it is that the `foldOr` definition is a formal spec of the required behavior and so the defintion is in actual fact an executable (formal) spec! In any case, we can now better appreciate how functional programming may give us an edge to get correct code out of the door.

Finishing Touches. Time to get back to our program. All we need to complete the implementation is some functionality to filter out of a Person list those who don't like functional programming — something equivalent to the `whoLikesFP` method in our initial object-oriented implementation, but without the bugs! We have already defined a function to tell us whether or not a given Person likes functional programming: `likesFP` :: Person → Bool. (For example, `likesFP` Joe = True.) Let's then use this function and the template above to define a new function `filterLikesFP` :: [Person] → [Person]. The first thing to decide is what to do with the empty list; this is a no brainer as there's nothing to filter and so we should return the empty list.

```
filterLikesFP [ ]  =  [ ]
```

[12] Smells funny? You're quite right ... we should at least first explain why now all of a sudden each element in our chain may be followed by many — e.g. False : [False, True] also follows [False, True]! But again, we're going to leave rigorous, formal thinking for another day and concentrate on developing a decent intuition first.

The next step is to define the function on a non-empty list of Person values $(x : xs)$. We have to come up with an expression involving x and (filterLikesFP xs) that reduces to a Person list as this is what the function returns. Now, if the Person at the head of the list likes functional programming (i.e. likesFP x = True), then we have to include this value in the result list; otherwise we have to discard it. Also, because of the way recursion works, we can assume that filterLikesFP will do exactly the same on xs and so it will deliver a list ps containing only those Person values p in xs for which likesFP p = True. Hence our expression should be something like this: if likesFP x = True then add x to the front of the list ps returned by (filterLikesFP xs); otherwise, just return ps. In Haskell, we can code this by using *guard* conditions on the defining equation like so:

```
filterLikesFP (x : xs)
                    | likesFP x  = x : filterLikesFP xs
                    | otherwise =     filterLikesFP xs
```

Finally we have all the bits and pieces needed to re-implement the program, whose entire listing is below.

```
module Workshop1 where

data Person = Joe | Tom | Sally
data Color  = Black | Green | Orange | Yellow | Blue | White

likesFP Joe   = True
likesFP Tom   = False
likesFP Sally = True

eyeColor Joe   = Black
eyeColor Tom   = Green
eyeColor Sally = Green

isGreen Green = True
isGreen  _    = False

mapEyeColor [ ]      = [ ]
mapEyeColor (x:xs) = eyeColor x : mapEyeColor xs

mapIsGreen [ ]      = [ ]
mapIsGreen (x:xs) = isGreen x : mapIsGreen xs

foldOr [ ]      = False
foldOr (x:xs) = x || foldOr xs

someoneGreenEyed = foldOr ∘ mapIsGreen ∘ mapEyeColor
```

```
filterLikesFP [ ]      = [ ]
filterLikesFP (x:xs)
     | likesFP x = x : filterLikesFP xs
     | otherwise =     filterLikesFP xs

ppl = [Joe, Tom, Sally]
answer = (someoneGreenEyed ∘ filterLikesFP) ppl
main = putStr (show answer ++ "\n")
```

You can find the file containing the source code (`Workshop1.hs`) on our wiki. To run the program use the following command line: `runghc Workshop1.hs`; it will print True, which is the right answer! Indeed we could even *prove* that the function $\mathsf{f} = \mathsf{someoneGreenEyed} \circ \mathsf{filterLikesFP}$ *always* arrives at the right answer. That is, if there exist one Person in the input list who likes functional programming and has green eyes, then True is returned. Conversely, if no Person who likes functional programming has green eyes, then False is returned. We have discovered a marvellous proof of this, which the margin is too short to contain and thus we leave it as a challenge to the (very) adventurous reader.[13]

[13] Just paraphrasing Fermat's claim about his famous Last Theorem, the proof of which actually took 400 years of forehead banging. So you're welcome to attempt a proof, but don't get disappointed if you don't find it in the next couple of hours!

Laziness That Pays

With our first functional program under our belt already, we want to finish off by pointing out some advanced features of certain functional languages (Haskell being one) that have the potential to make programs more efficient and modular. In the ethereal programming world of mathematical functions we've described so far, where equals stay equal, if a result can be arrived at following different evaluation paths, then, from a conceptual standpoint, which one we choose to follow is sort of irrelevant because they all are equivalent, meaning that they will have to lead to the same result,[14] given the same input. However, in the world of bits, which computations are actually executed (i.e. actual sequence of machine instructions) often matters as it may result in dramatic efficiency gains both in terms of memory used and number of steps (hence time) to complete the computation. The crux of the matter is that a given functional expression may admit more than one evaluation sequence, i.e. you can perform the evaluation steps in different orders to get to the same result. In pure functional programs we don't specify an execution order (like we do for imperative programs), so if many orderings are possible, which one will be used? How is the actual execution order going to affect performance?

You Only Get What You Need. Consider a simple computation to see if at least one out of a list of integers is odd. To make things straightforward, let's say we only have two integers, 5 and 10; we could use the functions we already know and write:

$$\text{answer} \;=\; \mathsf{foldOr}\ [\ \mathsf{isOdd}\ 5,\ \mathsf{isOdd}\ 10\ ]$$

Seasoned imperative programmers would probably expect the compiler to produce code to the effect that the following sequence of steps are executed at runtime:

(1) Evaluate (isOdd 5) to produce True.
(2) Evaluate (isOdd 10) to produce False.
(3) Allocate the list [True, False].
(4) Evaluate foldOr on the above list.

Although this is a possible evaluation order, it is not the only one. In fact, we could equally well evaluate the expression as follows:

$$\begin{array}{ll}
\mathsf{foldOr}\ [\ \mathsf{isOdd}\ 5,\ \mathsf{isOdd}\ 10\ ] & \\
= \mathsf{foldOr}\ (\ \mathsf{isOdd}\ 5 : [\ \mathsf{isOdd}\ 10\ ]) & \text{by starting with list's head} \\
= \mathsf{isOdd}\ 5\ \|\ \mathsf{foldOr}\ [\ \mathsf{isOdd}\ 10\ ] & \text{by evaluating } \mathsf{foldOr} \\
= \text{True}\ \|\ \mathsf{foldOr}\ [\ \mathsf{isOdd}\ 10\ ] & \text{by evaluating } \mathsf{isOdd}\ 5 \\
= \text{True} & \text{by evaluating } \|
\end{array}$$

As you can see, our strategy in this case is to always evaluate the function at the outermost position in the expression, starting from the left; this evaluation

[14] As long as the computation actually yields one; in other words, we're evaluating a terminating expression.

strategy is commonly referred to as *lazy evaluation* and is what Haskell uses under the bonnet.[15] It is similar in concept to short-circuiting of boolean expressions in an imperative language: only evaluate what is needed for the computation to make progress. We can clearly see this in the evaluation sequence above:

(1) To kick off the intial foldOr rewrite, we only need the head of the input list.
(2) After rewriting foldOr, to proceed we need to evaluate the isOdd function on 5.
(3) To evaluate the resulting "or" expression, only the left hand side is needed as that is enough for the $||$ function to produce its output.

Then the remainder of the expression (i.e. foldOr [isOdd 10]) is never evaluated, which means we have reached the result and spared unneeded computation steps at the same time!

As you can immagine, this is good news if we're planning to make extensive use of function pipelines like the one in our program above. For example, we see that if we were to evaluate

$$(\mathsf{someoneGreenEyed} \circ \mathsf{filterLikesFP})\ [\text{Sally, Tom, Joe}]$$

then only the head of the list would be needed to arrive at the answer (remember? Sally likes functional programming and has green eyes) and, moreover, the various functions in the pipeline need not be rewritten on the entire list. Here's the step by step evaluation — to save space we let $\mathsf{xs} = \mathsf{filterLikesFP}$ [Tom, Joe] after the first few steps:

$$\begin{aligned}
&(\mathsf{someoneGreenEyed} \circ \mathsf{filterLikesFP})\ [\text{Sally, Tom, Joe}]\\
&= \mathsf{someoneGreenEyed}\ (\mathsf{filterLikesFP}\ [\text{Sally, Tom, Joe}])\\
&= \mathsf{someoneGreenEyed}\ (\mathsf{filterLikesFP}\ (\text{Sally} : [\text{Tom, Joe}]))\\
&= \mathsf{someoneGreenEyed}\ (\text{Sally} : \mathsf{filterLikesFP}\ [\text{Tom, Joe}])\\
&= \mathsf{someoneGreenEyed}\ (\text{Sally} : \mathsf{xs})\\
&= \mathsf{foldOr}\ (\mathsf{mapIsGreen}\ (\mathsf{mapEyeColor}\ (\text{Sally} : \mathsf{xs})))\\
&= \mathsf{foldOr}\ (\mathsf{mapIsGreen}\ (\mathsf{eyeColor}\ \text{Sally} : \mathsf{mapEyeColor}\ \mathsf{xs})))\\
&= \mathsf{foldOr}\ (\mathsf{isGreen}\ (\mathsf{eyeColor}\ \text{Sally}) : \mathsf{mapIsGreen}\ (\mathsf{mapEyeColor}\ \mathsf{xs}))\\
&= \mathsf{isGreen}\ (\mathsf{eyeColor}\ \text{Sally})\ ||\ \mathsf{foldOr}\ (\mathsf{mapIsGreen}\ (\mathsf{mapEyeColor}\ \mathsf{xs}))\\
&= \mathsf{isGreen}\ \text{Green}\ ||\ \mathsf{foldOr}\ (\mathsf{mapIsGreen}\ (\mathsf{mapEyeColor}\ \mathsf{xs}))\\
&= \text{True}\ ||\ \mathsf{foldOr}\ (\mathsf{mapIsGreen}\ (\mathsf{mapEyeColor}\ \mathsf{xs}))\\
&= \text{True}
\end{aligned}$$

[15] There's actually more to it, but we'll skim over the details for now.

Out of the Loop. Contrast the above with the way we chained method calls to get the answer in our imperative version of the program. On the one hand, we see how the list items are "streamed" one at a time into the function pipeline: each function transforms the item and passes it onto the next funtion in the pipeline. So the list is only iterated once. On the other hand, in the imperative program, we first call `whoLikesFP` and then `someoneGreenEyed` but both have to iterate through the list each time; we could make this more efficient by squashing the two methods into one so that we have just a single loop, but we'd be doing that at the cost of modularity as the single big method would be less generic and less reusable. The problem is that each loop is "sealed" into a method and this limits the way in which we can decompose our code to make it more modular. What we would need is a way to separate the loop body (the computation on one item) from the iteration boundary imposed by the loop itself, which is exactly what lazy evaluation gives us.

Through this simple example we've seen how lazy evaluation can afford us new ways to decompose problems and make the code more modular, but this is just the tip of the iceberg! Although we won't have time to tackle it in depth, we will point out in due course how this mechanism can be used to greatly simplify and better modularize the design in the case of producer/consumer scenarios, expecially so when the producer may run indefinitely resulting in a potentially infinite output stream.[16]

[16] Indeed, several other applications are possible, but we won't even have the time to mention them!

The Condensed Ideas

Time to sum up the concepts we should remember. It's been a long discussion, but what made it long is the fact that we went through a lot of examples to develop a natural intuition of the concepts. So here's the short version of the entire story.

Functional Programming. A paradigm that sees computation as the evaluation of mathematical functions. In the *purely functional* model, equations are used to define computation rules whereby the equal sign is used to state that two expressions are identical and computation becomes a plain matter of rewriting expressions by replacing equals with equals until the expressions are reduced to the desired result. Evaluation is then by necessity side-effect free. All this makes programming more closely related to constructing mathematical objects on which one can reason using normal mathematical techniques.

Functions and Types. A *type* is a collection of values and a *function* is a rule that, given an input and an output type, associates one definite value of the output type to each value of the input type. To define a function we have to:

- ☐ Decide what is the type of the inputs the function will operate on and what is the type of the produced outputs — i.e. establish the function's *type signature.*
- ☐ Come up with a rule to associate each input to a unique output — this rule is defined in terms of equations and often we give a generic way to transform a typical input into an output so to avoid enumerating all possible input/output associations.

Function Composition. Given two functions, whenever the output type of the one is the same as the input type of the other, we can compute the one on an input value and pass the produced output on as an input to the other. Specifically, whenever we have two functions

$$f :: \alpha \to \beta \qquad g :: \beta \to \gamma$$

where α, β, and γ stand for arbitrary types such as Integer, Bool, or Person, we can always define a new one

$$\begin{aligned} &h :: \alpha \to \gamma \\ &h\ x \ = \ g\ (f\ x) \end{aligned}$$

This new function h is called the *composite* of g with f; the procedure we followed to obtain it is referred to as *function composition.* Using the function composition operator, h can be defined by the simple equation $h \ = \ g \circ f$. We can chain as many functions as we like using composition, provided that input/output types match. This gives us a simple, yet extremely effective mechanism to decompose a computation into small, very focused steps, each carried out by a function and then put them all together to get the desired result.

Induction and Recursion. A function can be defined in terms of itself; when this happens we say that the definition is *recursive*. Recursion gives us an elegant and powerful mechanism to process the elements of a list and a template to define recursive functions over lists is as follows. If f is the name of the function we want to define, then we need to give two equations:

$$\begin{aligned} \mathsf{f}\ [\,] &= \text{some atomic value} \\ \mathsf{f}\ (\mathsf{x} : \mathsf{xs}) &= \text{some terminating expression involving } \mathsf{x} \text{ and } \mathsf{f}\ \mathsf{xs} \end{aligned}$$

Functions so defined can be *proved* correct by means of a mathematical technique known as as *structural induction over lists*:

- □ Consider all the lists of type $[\alpha]$, α being a given type (e.g. Bool).
- □ Verify that a certain property P holds true for the empty list, $[\,]$.
- □ Assume that P holds for an arbitrary list xs and verify that it also holds for $(\mathsf{x} : \mathsf{xs})$, x being arbitrary too.
- □ Given all the above, it follows that all the lists of type $[\alpha]$ enjoy the property P.

Evaluation Model. A program is a function and running the program consists of evaluating that function on the provided input to yield an output. As functions are defined by equations, evaluation proceeds by simply interpreting those equations as left-to-right *rewite rules*, replacing a left hand side with a right hand side after *substituting* the inputs for the expression's variables; this process carries on until primitive values are reached and no further rewrites are possible. *Referential transparency* is a key tenet of purely functional programs: evaluating an expression and replacing it with its value yields an *equivalent* program (same outputs for same inputs) as there are no side-effects in rewriting the expression. As a consequence, if more than one evaluation sequence is possible for a given (terminating) expression, each will have to lead to the same result and hence several evaluation strategies may be possible. *Lazy evaluation* is a strategy whereby evaluation always begins with the function at the outermost position in the expression, starting from the left; the implication is that any part of an expression that is not needed for the computation to progress is never evaluated. If properly mastered, lazy evaluation can greately increase modularity as well as help improve performance.

References

[1] P.G. Hinman. *Fundamentals of Mathematical Logic*. AK Peters, Ltd., 2005.