Folding Unfolded

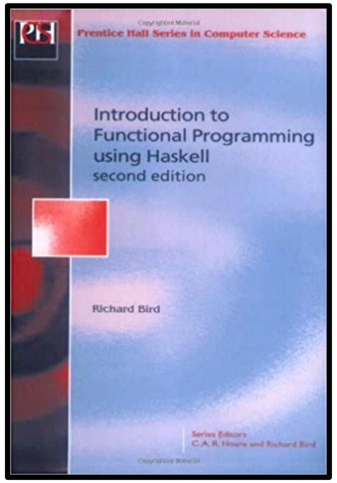
Polyglot FP for Fun and Profit
Haskell and Scala

See how **recursive functions** and **structural induction** relate to **recursive datatypes**

Follow along as the **fold abstraction** is introduced and explained

Watch as **folding** is used to simplify the definition of **recursive functions** over **recursive datatypes**

Part 1 - through the work of



Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

Graham Hutton
@haskellhutt

A tutorial on the universality and expressiveness of fold

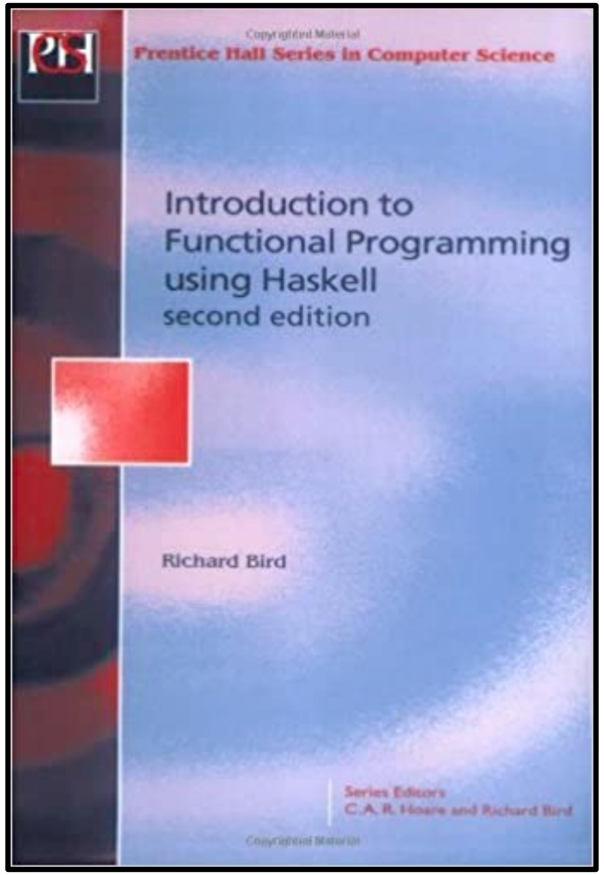
GRAHAM HUTTON
University of Nottingham, Nottingham, UK
http://www.cs.nott.ac.uk/~gmh

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

This slide deck is almost entirely centered on material from **Richard Bird**'s fantastic book, **Introduction to Functional Programming using Haskell**.

I hope he'll forgive me for relying so heavily on his work, but I couldn't resist using extensive excerpts from his compelling book to introduce and explain the concept of **folding**.



Richard Bird

http://www.cs.ox.ac.uk/people/richard.bird/

https://en.wikipedia.org/wiki/Richard_Bird_(computer_scientist)

3.1 Natural Numbers

The natural numbers are the numbers 0, 1, 2 and so on, used for counting. The type Nat is introduced by the declaration

$$\mathbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

Nat is our first example of a **recursive** **datatype** **declaration**. The definition says that Zero is a value of Nat, and that $\mathit{Succ}\ n$ is a value of Nat whenever n is. In particular, the constructor Succ (short for 'successor'), has type $\mathit{Nat} \rightarrow \mathit{Nat}$. For example, each of

$$\mathit{Zero}, \mathit{Succ}\ \mathit{Zero}, \mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})$$

is an element of Nat. As an element of Nat the number 7 would be represented by

$$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))))))$$

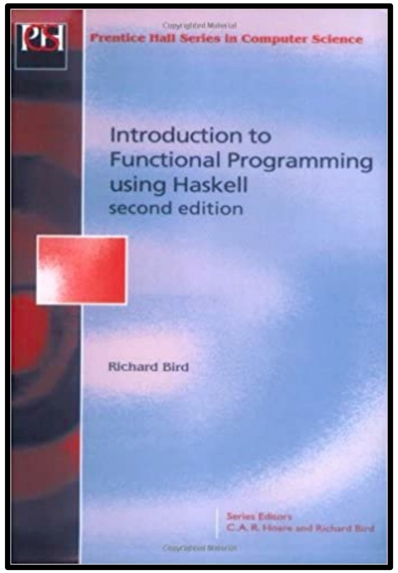
Every natural number is represented by a unique value of Nat. On the other hand, not every value of Nat represents a well-defined natural number. In fact Nat also contains the values $\bot$, $\mathit{Succ}\ \bot$, $\mathit{Succ}\ (\mathit{Succ}\ \bot)$, and so on. These additional values will be discussed later.

Let us see how to program the basic arithmetic and comparison operations on Nat. Addition can be defined by

$$\begin{aligned} (+) \quad &:: \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat} \\ m + \mathit{Zero} &= m \\ m + \mathit{Succ}\ n &= \mathit{Succ}\ (m + n) \end{aligned}$$

This is a **recursive definition**, defining $+$ by pattern matching on the second argument. Since every element of Nat, apart for $\bot$, is either Zero or of the form $\mathit{Succ}\ n$, where n is an element of Nat, the two patterns in the equations for $+$ are disjoint and cover all numbers apart from $\bot$.

…



Richard Bird

Here is how $Zero + Succ\ (Succ\ Zero)$ would be evaluated:

$$
\begin{aligned}
& Zero + Succ\ (Succ\ Zero) \\
= & \quad \{ \text{second equation for } +, \text{ i.e. } m + Succ\ n = Succ\ (m + n) \} \\
& Succ\ (Zero + Succ\ Zero) \\
= & \quad \{ \text{second equation for } +, \text{ i.e. } m + Succ\ n = Succ\ (m + n) \} \\
& Succ\ (Succ\ (Zero + Zero)) \\
= & \quad \{ \text{first equation for } +, \text{ i.e. } m + Zero = m \} \\
& Succ\ (Succ\ Zero)
\end{aligned}
$$

…it is not a practical proposition to introduce natural numbers through the datatype Nat: arithmetic would be just too inefficient. In particular, calculating $m + \mathrm{n}$ would require $(n + 1)$ evaluation steps. On the other hand, counting on your fingers is a good way to understand addition.

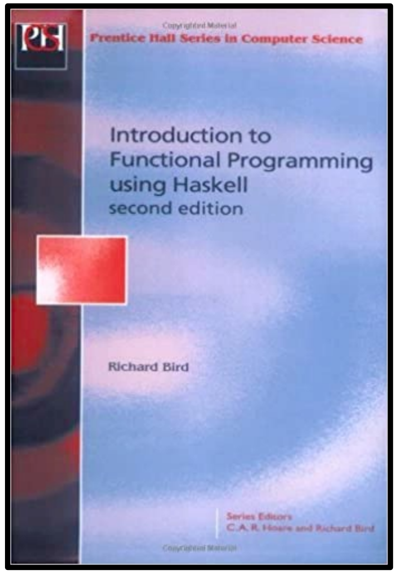
Given $+$, we can define $\times$:

$$
\begin{aligned}
& (\times) && :: \quad Nat \rightarrow Nat \rightarrow Nat \\
& m \times Zero && = \quad Zero \\
& m \times Succ\ n && = \quad (m \times n) + m
\end{aligned}
$$

Given $\times$, we can define exponentiation $(\uparrow)$ by

$$
\begin{aligned}
& (\uparrow) && :: \quad Nat \rightarrow Nat \rightarrow Nat \\
& m \uparrow Zero && = \quad Succ\ Zero \\
& m \uparrow Succ\ n && = \quad (m \uparrow n) \times m
\end{aligned}
$$

…



Richard Bird

On the next slide we show the definitions of $+$, $\times$, and $\uparrow$ again, and have a go at implementing the three operations in **Scala**, together with some tests.

```
data Nat = Zero | Succ Nat
```

```scala
sealed trait Nat
case class Succ(n: Nat) extends Nat
case object Zero extends Nat
```

$$
\begin{aligned}
(+) &:: Nat \to Nat \to Nat \\
m + Zero &= m \\
m + Succ\ n &= Succ\ (m + n)
\end{aligned}
$$

$$
\begin{aligned}
(\times) &:: Nat \to Nat \to Nat \\
m \times Zero &= Zero \\
m \times Succ\ n &= (m \times n) + m
\end{aligned}
$$

$$
\begin{aligned}
(\uparrow) &:: Nat \to Nat \to Nat \\
m \uparrow Zero &= Succ\ Zero \\
m \uparrow Succ\ n &= (m \uparrow n) \times m
\end{aligned}
$$

```scala
val `(+)`: Nat => Nat => Nat =
  m => {
    case Zero    => m
    case Succ(n) => Succ(m + n)
  }
```

```scala
val `(×)`: Nat => Nat => Nat =
  m => {
    case Zero    => Zero
    case Succ(n) => (m × n) + m
  }
```

```scala
val `(↑)`: Nat => Nat => Nat =
  m => {
    case Zero    => Succ(Zero)
    case Succ(n) => (m ↑ n) × m
  }
```

```scala
implicit class NatOps(m: Nat){
  def +(n: Nat) = `(+)`(m)(n)
  def ×(n: Nat) = `(×)`(m)(n)
  def ↑(n: Nat) = `(↑)`(m)(n)
}
```

```scala
; assert(0 + 0 == 0) ; assert(Zero + Zero == Zero)
; assert(0 + 1 == 1) ; assert(Zero + Succ(Zero) == Succ(Zero))
; assert(1 + 0 == 1) ; assert(Succ(Zero) + Zero == Succ(Zero))
; assert(1 + 1 == 2) ; assert(Succ(Zero) + Succ(Zero) == Succ(Succ(Zero)))
; assert(2 + 3 == 5) ; assert(Succ(Succ(Zero)) + Succ(Succ(Succ(Zero))) == Succ(Succ(Succ(Succ(Succ(Zero))))))
```

```scala
; assert(0 * 0 == 0) ; assert((Zero × Zero) == Zero)
; assert(1 * 0 == 0) ; assert((Succ(Zero) × Zero) == Zero)
; assert(0 * 1 == 0) ; assert((Zero × Succ(Zero)) == Zero)
; assert(1 * 1 == 1) ; assert((Succ(Zero) × Succ(Zero)) == Succ(Zero))
; assert(1 * 2 == 2) ; assert((Succ(Zero) × Succ(Succ(Zero))) == Succ(Succ(Zero)))
; assert(2 * 3 == 6) ; assert((Succ(Succ(Zero)) × Succ(Succ(Succ(Zero)))) == Succ(Succ(Succ(Succ(Succ(Succ(Zero)))))))
```

```scala
; assert(Math.pow(1,0) == 1) ; assert( (Succ(Zero) ↑ Zero) == Succ(Zero) )
; assert(Math.pow(2,2) == 4) ; assert( (Succ(Succ(Zero)) ↑ Succ(Succ(Zero))) == Succ(Succ(Succ(Succ(Zero)))) )
```

The remaining arithmetic operation common to all numbers is subtraction $(-)$. However, subtraction is a **partial operation** on natural numbers. The definition is

$$
\begin{array}{lcl}
(-) & :: & \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat} \\
m - \mathit{Zero} & = & m \\
\mathit{Succ}\, m - \mathit{Succ}\, n & = & m - n
\end{array}
$$

This definition uses pattern matching on both arguments; taken together, the patterns are disjoint but not exhaustive. For example,

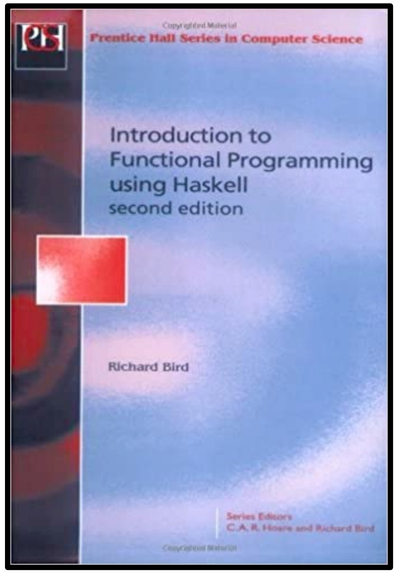
$$
\begin{array}{ll}
 & \mathit{Succ\, Zero} - \mathit{Succ}\, (\mathit{Succ\, Zero}) \\
= & \{ \text{second equation for } -\text{, i.e. } \mathit{Succ}\, m - \mathit{Succ}\, n \;=\; m - n \} \\
 & \mathit{Zero} - \mathit{Succ\, Zero} \\
= & \{ \text{case exhaustion} \} \\
 & \bot
\end{array}
$$

The hint 'case exhaustion' in the last step indicates that no equation for $(-)$ has a pattern that matches $(\mathit{Zero} - \mathit{Succ\, Zero})$. More generally, $m - n = \bot$ if $m < n$. The **partial nature** of subtraction on the natural numbers is the prime motivation for introducing the **integer** numbers; over the **integers**, $(-)$ is a **total operation**.

…Finally, here are two more examples of programming with Nat. The **factorial** and **Fibonacci** functions are defined by

$$
\begin{array}{lcl}
\mathit{fact} & :: & \mathit{Nat} \rightarrow \mathit{Nat} \\
\mathit{fact\, Zero} & = & \mathit{Succ\, Zero} \\
\mathit{fact}\, (\mathit{Succ}\, n) & = & \mathit{Succ}\, n \times \mathit{fact}\, n
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{fib} & :: & \mathit{Nat} \rightarrow \mathit{Nat} \\
\mathit{fib\, Zero} & = & \mathit{Zero} \\
\mathit{fib}\, (\mathit{Succ\, Zero}) & = & \mathit{Succ\, Zero} \\
\mathit{fib}\, (\mathit{Succ}\, (\mathit{Succ}\, n)) & = & \mathit{fib}(\mathit{Succ}\, n) + \mathit{fib}\, n
\end{array}
$$



Richard Bird

See the next slide for a **Scala** implementation of the – operation.

$$
\begin{aligned}
(-) \quad & :: \; \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
m - \mathit{Zero} \quad & = \; m \\
\mathit{Succ}\; m - \mathit{Succ}\; n \quad & = \; m - n
\end{aligned}
$$

No equation for $(-)$ has a pattern that matches $(\mathit{Zero} - \mathit{Succ\ Zero})$.
Similarly for $(\mathit{Zero} - \mathit{Succ}\ (\mathit{Succ\ Zero}))$, $(\mathit{Zero} - \mathit{Succ}\ (\mathit{Succ}(\mathit{Succ\ Zero})))$, etc.
More generally, $m - n = \bot$ if $m < n$.

```scala
val `(-)`: Nat => Nat => Nat =
  m => n => (m,n) match {
    case (_,Zero) => m
    case (Succ(x),Succ(y)) => x - y
  }
```

$m - n$ throws `scala.MatchError` if $m < n$.

```scala
; assert(0 - 0 ==  0) ; assert(Zero - Zero == Zero)
; assert(1 - 0 ==  1) ; assert(Succ(Zero) - Zero == Succ(Zero))
; assert(5 - 3 ==  2) ; assert(Succ(Succ(Succ(Succ(Succ(Zero))))) - Succ(Succ(Succ(Zero))) == Succ(Succ(Zero)))
; assert(3 - 5 == -2) ; assert(
                          Try {
                             Succ(Succ(Succ(Zero))) - Succ(Succ(Succ(Succ(Succ(Zero)))))
                           }.toString.startsWith(
                             "Failure(scala.MatchError: (Zero,Succ(Succ(Zero)))"
                           )
                        )
```

@philip_schwarz

On the next slide we show the definitions of $fact$, and fib again, and have a go at implementing the two functions in **Scala**, together with some tests.

$$
\begin{aligned}
&\mathit{fact} &&:: \mathit{Nat} \rightarrow \mathit{Nat}\\
&\mathit{fact}\ \mathit{Zero} &&= \mathit{Succ}\ \mathit{Zero}\\
&\mathit{fact}\ (\mathit{Succ}\ n) &&= \mathit{Succ}\ n \times \mathit{fact}\ n
\end{aligned}
$$

```
val fact: Nat => Nat = {
  case Zero     => Succ(Zero)
  case Succ(n) => Succ(n) × fact(n)
}
```

```
def factorial(n: Int): Int =
  if (n == 0) 1
  else n * factorial(n-1)

; assert(factorial(0) == 1) ; assert(fact(Zero) == Succ(Zero))
; assert(factorial(1) == 1) ; assert(fact(Succ(Zero)) == Succ(Zero))
; assert(factorial(2) == 2) ; assert(fact(Succ(Succ(Zero))) == Succ(Succ(Zero)))
; assert(factorial(3) == 6) ; assert(fact(Succ(Succ(Succ(Zero)))) == Succ(Succ(Succ(Succ(Succ(Succ(Zero)))))))
```

$$
\begin{aligned}
&\mathit{fib} &&:: \mathit{Nat} \rightarrow \mathit{Nat}\\
&\mathit{fib}\ \mathit{Zero} &&= \mathit{Zero}\\
&\mathit{fib}\ (\mathit{Succ}\ \mathit{Zero}) &&= \mathit{Succ}\ \mathit{Zero}\\
&\mathit{fib}\ \big(\mathit{Succ}\ (\mathit{Succ}\ n)\big) &&= \mathit{fib}(\mathit{Succ}\ n) + \mathit{fib}\ n
\end{aligned}
$$

```
val fib: Nat => Nat = {
  case Zero           => Zero
  case Succ(Zero)     => Succ(Zero)
  case Succ(Succ(n)) => fib(Succ(n)) + fib(n)
}
```

```
def fibonacci(n: Int): Int =
  if (n == 0 || n == 1) n
  else fibonacci(n-1) + fibonacci(n-2)

; assert(fibonacci(0) == 0) ; assert(fib(Zero) == Succ(Zero))
; assert(fibonacci(1) == 1) ; assert(fib(Succ(Zero)) == Succ(Zero))
; assert(fibonacci(2) == 1) ; assert(fib(Succ(Succ(Zero))) == Succ(Zero))
; assert(fibonacci(3) == 2) ; assert(fib(Succ(Succ(Succ(Zero)))) == Succ(Succ(Zero)))
; assert(fibonacci(4) == 3) ; assert(fib(Succ(Succ(Succ(Succ(Zero))))) == Succ(Succ(Succ(Zero))))
; assert(fibonacci(5) == 5) ; assert(fib(Succ(Succ(Succ(Succ(Succ(Zero)))))) == Succ(Succ(Succ(Succ(Succ(Zero))))))
; assert(fibonacci(6) == 8) ; assert(fib(Succ(Succ(Succ(Succ(Succ(Zero)))))) == Succ(Succ(Succ(Succ(Succ(Succ(Succ(Succ(Zero)))))))))
```

3.1.1 Partial numbers

Let us now return to the point about there being extra values in Nat. The values

$\bot$, $\mathit{Succ}\ \bot$, $\mathit{Succ}\ (\mathit{Succ}\ \bot)$, ...

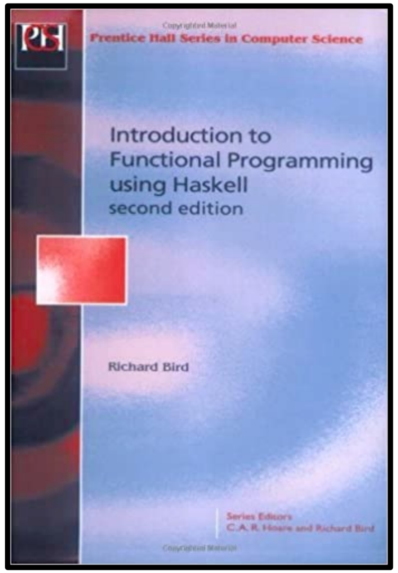
are all different and each is also a member of Nat. That they exist is a consequence of three facts:

i. $\bot$ is an element of Nat because every datatype declaration introduces at least one extra value, the **undefined** value of the type.
ii. constructor functions of a datatype are assumed to be **nonstrict**
iii. $\mathit{Succ}\ n$ is an element of **Nat**, whenever n is

To appreciate why these extra values are different from one another, suppose we define $\mathit{undefined} :: \mathit{Nat}$ by the equation $\mathit{undefined} = \mathit{undefined}$. Then

? $\mathit{Zero} < \mathit{undefined}$
{Interrupted!}
? $\mathit{Zero} < \mathit{Succ}\ \mathit{undefined}$
True
? $\mathit{Succ}\ \mathit{Zero} < \mathit{Succ}\ \mathit{undefined}$
{Interrupted!}
? $\mathit{Succ}\ \mathit{Zero} < \mathit{Succ}\ (\mathit{Succ}\ \mathit{undefined})$
True

One can interpret the extra values in the following way: $\bot$ corresponds to the natural number about which there is absolutely no information; $\mathit{Succ}\ \bot$ to the natural number about which the only information is that it is greater than Zero; $\mathit{Succ}\ (\mathit{Succ}\ \bot)$ to the natural number about which the only information is that it is greater than $\mathit{Succ}\ \mathit{Zero}$; and so on.



Richard Bird

There is also one further value of $\textit{Nat}$, namely the 'infinite' number:

$$\textit{Succ}\ (\textit{Succ}\ (\textit{Succ}\ (\textit{Succ}\ \ldots)))$$

This number can be defined by

$$\begin{aligned} \textit{infinity} &:: \textit{Nat} \\ \textit{infinity} &= \textit{Succ infinity} \end{aligned}$$

It is different from all the other numbers, because it is the only number x for which $\textit{Succ}\ m < x$ returns $\textit{True}$ for all finite numbers m. In this sense, $\textit{infinity}$ is the largest element of $\textit{Nat}$. If we request the value of $\textit{infinity}$, then we obtain

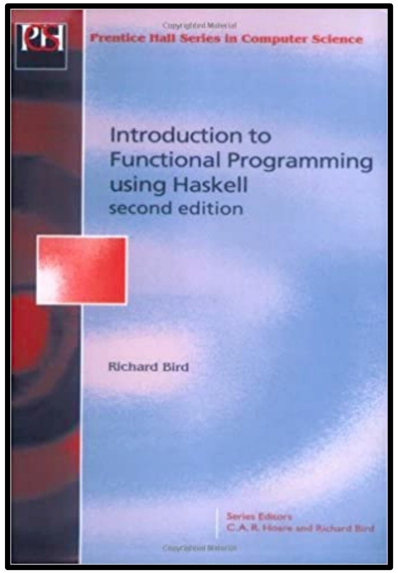
$? \ \textit{infinity}$
$\textit{Succ}\ (\textit{Succ}\ (\textit{Succ}\ (\textit{Succ}\ (\textit{Succ}\ \{\textit{Interrupted}!\}$

The number $\textit{infinity}$ satisfies other properties, in particular $n + \textit{infinity} = \textit{infinity}$, for all numbers n. The dual equation $\textit{infinity} + n = \textit{infinity}$ holds only for finite numbers n. We will see how to prove assertions such as these in the next section.

To summarise this discussion, we can divide the values of $\textit{Nat}$ into three classes:

- The **finite** numbers, those that correspond to well-defined natural numbers.
- The **partial** numbers, $\perp$, $\textit{Succ}\ \perp$, and so on.
- The **infinite** numbers, of which there is just one, namely $\textit{infinity}$.

We will see that this classification holds true of *all* **recursive types**. There will be the **finite** elements of the type, the **partial** elements, and the **infinite** elements. Although the infinite natural number is not of much use, the same is not true of the infinite values of other datatypes.
...



Richard Bird

Note that when in this slide deck we mention the concepts of $\bot$ and $infinity$, it is mainly in a **Haskell** context, as we did in the last two slides. In particular, we won't be modelling $\bot$ and $infinity$ in any of the **Scala** code you'll see throughout the deck.

3.2 Induction

In order to reason about the properties of **recursively defined** functions over a **recursive datatype**, we can appeal to a principle of ***structural induction***. In the case of Nat, the principle of **structural induction** can be defined as follows: In order to show that some property $P(n)$ holds for each finite number n of Nat, it is sufficient to show:

Case (Zero). That $P(\mathit{Zero})$ holds.

Case ($\mathit{Succ}\ n$). That if $P(n)$ holds, then $P(\mathit{Succ}\ n)$ holds also.

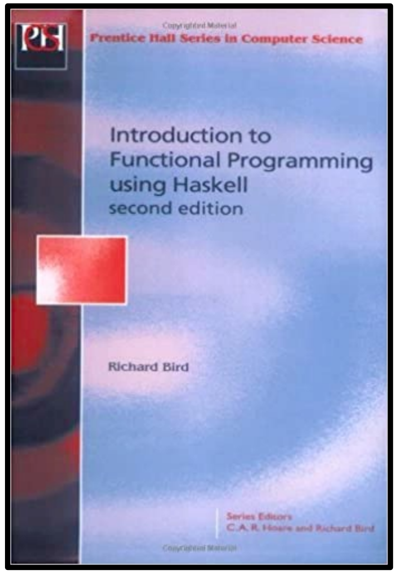
Induction is valid for the same reason that **recursive definitions** are valid: every finite number is either Zero or of the form $\mathit{Succ}\ n$, where n is a finite number. If we prove the first case, then we have shown that the property is true for Zero; If we also prove the second case, then we have shown that the property is true for $\mathit{Succ}\ \mathit{Zero}$, since it is true for Zero. But now, by the same argument, it is true for $\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})$, and so on.

The principle needs to be extended if we want to assert that some proposition is true for *all* elements of Nat, but we postpone discussion of this point for the following section.

As an example, let's prove that $\mathit{Zero} + n \ = \ n$ for all finite numbers n. Recall that $+$ is defined by

$$
\begin{aligned}
m + \mathit{Zero} &= m \\
m + \mathit{Succ}\ n &= \mathit{Succ}\ (m + n)
\end{aligned}
$$

The first equation asserts that Zero is a right unit of $+$. In general, e is a **left unit** of $\oplus$ if $e \oplus x = x$ for all x, and a **right unit** of x if $x \oplus e = x$ for all x. If e is both a **left unit** and a **right unit** of an operator $\oplus$, then it is called *the* **unit** of $\oplus$. The terminology is appropriate since only one value can be both a left and right **unit**. So, by proving that Zero is a **left unit** , we have proved that Zero is *the* **unit** of $+$.



Richard Bird

Proof. The proof is by induction on n. More precisely, we take for $P(n)$ the assertion that $Zero + n \ = n$. This equation is referred to as the **induction hypothesis**.

Case ($Zero$). We have to show $Zero + Zero = Zero$, which is immediate from the first equation defining $+$.

Case ($Succ\ n$). We have to show that $Zero + Succ\ n = Succ\ n$, which we do by simplifying the left-hand expression:

$$
\begin{aligned}
& Zero + Succ\ n \\
= \quad & \{ \text{second equation for } +, \text{ i.e. } m + Succ\ n \ = \ Succ\ (m + n) \} \\
& Succ\ (Zero + n) \\
= \quad & \{ \text{induction hypothesis} \} \\
& Succ\ n
\end{aligned}
$$

□

This example shows the format we will use for **inductive proofs**, laying out each case separately and using a □ to mark the end. The very last step made use of the **induction hypothesis**, which is allowed by the way **induction** works.

…

3.2.1 Full Induction

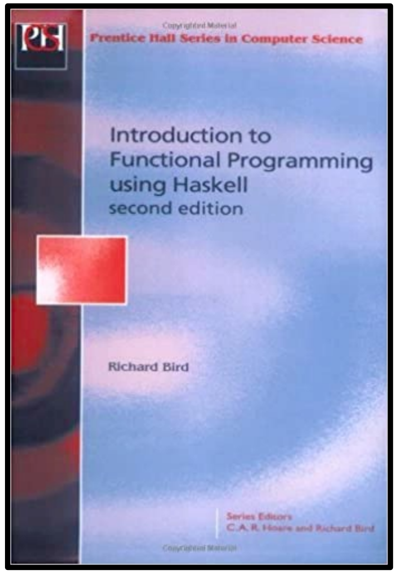
In the form given above, the **induction principle** for Nat suffices only to prove properties of the **finite** members of Nat. If we want to show that a property P also hold for every **partial number**, then we have to prove three things:

Case ($\bot$). That $P(\bot)$ holds.

Case ($Zero$). That $P(Zero)$ holds.

Case ($Succ\ n$). That if $P(n)$ holds, then $P(Succ\ n)$ holds also.

We can omit the second case, but then we can conclude only that $P(n)$ holds for every **partial** number. The reason the principle is valid is that is that every **partial number** is either $\bot$ or of the form $Succ\ n$ for some **partial number** n.



Richard Bird

To illustrate, let us prove the somewhat counterintuitive result that $m + n = n$ for all numbers m and all **partial numbers** n.

Proof. The proof is by **partial number induction** on n.

Case ($\bot$). The equation $m + \bot = \bot$ follows at once by case exhaustion in the definition of $+$. That is, $\bot$ does not match either of the patterns ***Zero*** or ***Succ*** n.

Case (***Succ*** n). For the left-hand side, we reason

$$
\begin{array}{ll}
 & m + \mathit{Succ}\ n \\
= & \quad \{\text{second equation for } +, \text{ i.e. } m + \mathit{Succ}\ n \ = \ \mathit{Succ}\ (m+n)\ \} \\
 & \mathit{Succ}\ (m + n) \\
= & \quad \{\text{induction hypothesis}\} \\
 & \mathit{Succ}\ n
\end{array}
$$

Since the right-hand side is also ***Succ*** n, we are done.

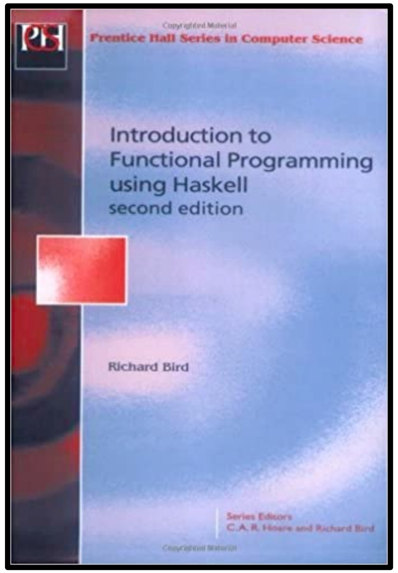
3.2.2 Program synthesis

In the proofs above we defined some functions and then used induction to prove a certain property. We can also view induction as a way to **synthesise** definitions of functions so that they satisfy the properties we want.

Let us illustrate with a simple example. Suppose we **specify** subtraction of natural numbers by the condition

$$(m + n) - n = m$$

for all m and n. The specification does not give a constructive definition of $(-)$, merely a property that it has to satisfy. However, we can do an **induction proof** on n of the equation above, but view the calculation as a way of generating a suitable definition of $(-)$.



Richard Bird

Unlike previous proofs, we reason with the equation as a whole, since simplification of both sides independently is not possible if we do not know what all the rules of simplification are.

Case (Zero). We reason

$$(m + \mathit{Zero}) - \mathit{Zero} = m$$
$$\equiv \quad \{\text{first equation for } +\text{, i.e. } m + \mathit{Zero} = m\}$$
$$m - \mathit{Zero} = m$$

Hence we can take $m - \mathit{Zero} = m$ to satisfy the case. The symbol $\equiv$ is used to separate steps of the calculation since we are calculating with mathematical assertions, not with values of a datatype.

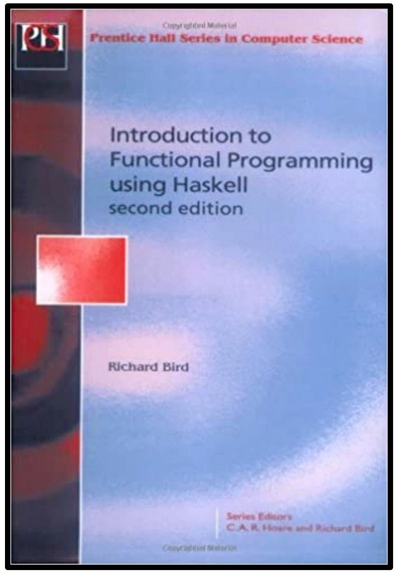
Case ($\mathit{Succ}\ n$). We reason

$$(m + \mathit{Succ}\ n) - \mathit{Succ}\ n = m$$
$$\equiv \quad \{\text{second equation for } +\text{, i.e. } m + \mathit{Succ}\ n = \mathit{Succ}\ (m + n)\}$$
$$\mathit{Succ}\ (m + n) - \mathit{Succ}\ n = m$$
$$\equiv \quad \{\text{hypothesis } (m + n) - n = m\}$$
$$\mathit{Succ}\ (m + n) - \mathit{Succ}\ n = (m + n) - n$$

Replacing $m + n$ in the last equation by m, we can take $\mathit{Succ}\ m - \mathit{Succ}\ n = m - n$ to satisfy the case. Hence we have derived

$$\begin{aligned} m - \mathit{Zero} &= m \\ \mathit{Succ}\ m - \mathit{Succ}\ n &= m - n \end{aligned}$$

This is the program for $(-)$ seen earlier.



Richard Bird

After that look at **structural induction**, it is finally time to see how **Richard Bird** introduces the concept of **folding**.

3.3 The fold function

Many of the **recursive definitions** seen so far have a common pattern, exemplified by the following definition of a function f:

$$
\begin{aligned}
&f &&:: \ \mathit{Nat} \rightarrow A \\
&f\ \mathit{Zero} &&= \ c \\
&f\ (\mathit{Succ}\ n) &&= \ h\ (f\ n)
\end{aligned}
$$

Here, A is some type, c is an element of A, and $h :: A \rightarrow A$. Observe that f works by taking an element of Nat and replacing Zero by c and Succ by h. For example, f takes

$$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})) \quad \text{to} \quad h\ (h\ (h\ c))$$

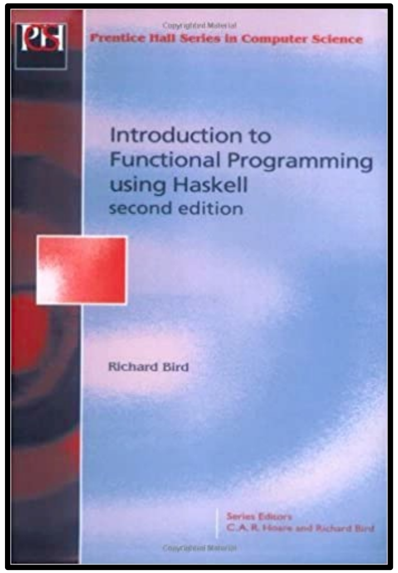
The two equations for f can be captured in terms of a single function, foldn, called the fold function for Nat. The definition is

$$
\begin{aligned}
&\mathit{foldn} &&:: \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \mathit{Nat} \rightarrow \alpha \\
&\mathit{foldn}\ h\ c\ \mathit{Zero} &&= \ c \\
&\mathit{foldn}\ h\ c\ (\mathit{Succ}\ n) &&= \ h\ (\mathit{foldn}\ h\ c\ n)
\end{aligned}
$$

In particular, we have

$$
\begin{aligned}
m + n &= \mathit{foldn}\ \mathit{Succ}\ m\ n \\
m \times n &= \mathit{foldn}\ (+\ m)\ \mathit{Zero}\ n \\
m \uparrow n &= \mathit{foldn}\ (\times\ m)\ (\mathit{Succ}\ \mathit{Zero})\ n
\end{aligned}
$$

It follows also that the **identity function** id on Nat satisfies $\mathit{id} = \mathit{foldn}\ \mathit{Succ}\ \mathit{Zero}$. A suitable fold function can be defined for every **recursive type**, and we will see other fold functions in the following chapters.



Richard Bird

@philip_schwarz

Just to reinforce the ideas on the previous slide, here are the original definitions of $+$, $\times$ and $\uparrow$, and next to them, the new definitions in terms of $foldn$.

And the next slide is the same but in terms of **Scala** code.

$$
\begin{aligned}
(+) &:: Nat \to Nat \to Nat \\
m + Zero &= m \\
m + Succ\ n &= Succ\ (m + n)
\end{aligned}
$$

$$
\begin{aligned}
(+) &:: Nat \to Nat \to Nat \\
m + n &= foldn\ Succ\ m\ n
\end{aligned}
$$

$$
\begin{aligned}
(\times) &:: Nat \to Nat \to Nat \\
m \times Zero &= Zero \\
m \times Succ\ n &= (m \times n) + m
\end{aligned}
$$

$$
\begin{aligned}
(\times) &:: Nat \to Nat \to Nat \\
m \times n &= foldn\ (+\ m)\ Zero\ n
\end{aligned}
$$

$$
\begin{aligned}
(\uparrow) &:: Nat \to Nat \to Nat \\
m \uparrow Zero &= Succ\ Zero \\
m \uparrow Succ\ n &= (m \uparrow n) \times m
\end{aligned}
$$

$$
\begin{aligned}
(\uparrow) &:: Nat \to Nat \to Nat \\
m \uparrow n &= foldn\ (\times\ m)\ (Succ\ Zero)\ n
\end{aligned}
$$

$$
\begin{aligned}
foldn &:: (\alpha \to \alpha) \to \alpha \to Nat \to \alpha \\
foldn\ h\ c\ Zero &= c \\
foldn\ h\ c\ (Succ\ n) &= h\ (foldn\ h\ c\ n)
\end{aligned}
$$

```
val `(+)`: Nat => Nat => Nat =
  m => {
    case Zero    => m
    case Succ(n) => Succ(m + n)
  }
```

```
val `(+)`: Nat => Nat => Nat =
  m => n => foldn(Succ,m,n)
```

```
val `(×)`: Nat => Nat => Nat =
  m => {
    case Zero    => Zero
    case Succ(n) => (m × n) + m
  }
```

```
val `(×)`: Nat => Nat => Nat =
  m => n => foldn((x:Nat) => x + m, Zero, n)
```

```
val `(↑)`: Nat => Nat => Nat =
  m => {
    case Zero    => Succ(Zero)
    case Succ(n) => (m ↑ n) × m
  }
```

```
val `(↑)`: Nat => Nat => Nat =
  m => n => foldn((x:Nat) => x × m, Succ(Zero), n)
```

```
def foldn[A](h: A => A, c: A, n: Nat): A =
  n match {
    case Zero => c
    case Succ(n) => h(foldn(h,c,n))
  }
```

In the examples above, each instance of $foldn$ also returned an element of $\mathbf{Nat}$. In the following two examples, $foldn$ returns an element of ($\mathbf{Nat}$, $\mathbf{Nat}$):

$$\begin{aligned} fact &:: \mathbf{Nat} \rightarrow \mathbf{Nat} \\ fact &= snd \cdot foldn\ f\ (\mathbf{Zero}, \mathbf{Succ}\ \mathbf{Zero}) \\ &\quad \textbf{where}\ f\,(m,n) = (\mathbf{Succ}\ m, \mathbf{Succ}\ (m) \times n) \end{aligned}$$

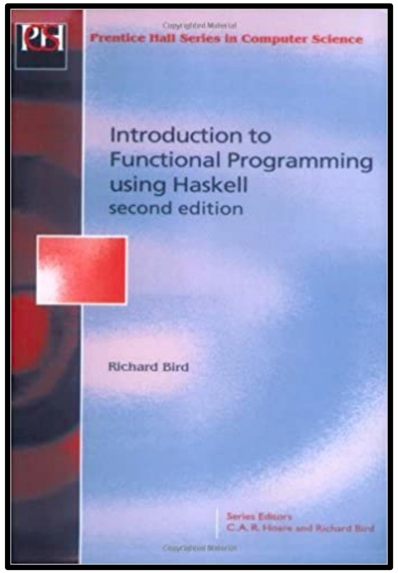
$$\begin{aligned} fib &:: \mathbf{Nat} \rightarrow \mathbf{Nat} \\ fib &= fst \cdot foldn\ g\ (\mathbf{Zero}, \mathbf{Succ}\ \mathbf{Zero}) \\ &\quad \textbf{where}\ g\,(m,n) = (n, m + n) \end{aligned}$$

The function $fact$ computes the **factorial** function and function fib computes the **Fibonacci** function. Each program works by first computing a more general result, namely an element of ($\mathbf{Nat}$, $\mathbf{Nat}$), and then extracts the required result. In fact,

$$\begin{aligned} foldn\ f\ (\mathbf{Zero}, \mathbf{Succ}\ \mathbf{Zero})\ n &= (n, fact\ n) \\ foldn\ g\ (\mathbf{Zero}, \mathbf{Succ}\ \mathbf{Zero})\ n &= \left(fib\ n, fib\ (\mathbf{Succ}\ n)\right) \end{aligned}$$

These equations can be **proved by induction**. The program for fib is more efficient than a direct **recursive definition**. The recursive program requires an **exponential number** of $+$ operations, while the program above requires only a **linear number**. We will discuss efficiency in more detail in chapter 7, where the programming technique that led to the invention of the new program for fib will be studied in a more general setting.

There are two advantages of writing **recursive definitions** in terms of $foldn$. Firstly, the definition is shorter; rather than having to write down two equations, we have only to write down one. Secondly, it is possible to prove general properties of $foldn$ and use them to prove properties of specific instantiations. In other words, rather than having to write down many **induction proofs**, we have only to write down one.



Richard Bird

@philip_schwarz

The next slide shows the original definitions of the **factorial** and **Fibonacci** functions, and next to them, the new definitions in terms of $foldn$.

And the slide after that is the same but in terms of **Scala** code.

$$\begin{aligned} fact &:: Nat \to Nat \\ fact\ Zero &= Succ\ Zero \\ fact\ (Succ\ n) &= Succ\ n \times fact\ n \end{aligned}$$

$$\begin{aligned} fact &:: Nat \to Nat \\ fact &= snd \cdot foldn\ f\ (Zero, Succ\ Zero) \\ &\quad \textbf{where}\ f\,(m,n) = (Succ\ m, Succ\ (m) \times n) \end{aligned}$$

$$\begin{aligned} fib &:: Nat \to Nat \\ fib\ Zero &= Zero \\ fib\ (Succ\ Zero) &= Succ\ Zero \\ fib\ \big(Succ\ (Succ\ n)\big) &= fib(Succ\ n) + fib\ n \end{aligned}$$

$$\begin{aligned} fib &:: Nat \to Nat \\ fib &= fst \cdot foldn\ g\ (Zero, Succ\ Zero) \\ &\quad \textbf{where}\ g(m,n) = (n, m + n) \end{aligned}$$

$$\begin{aligned} foldn &:: (\alpha \to \alpha) \to \alpha \to Nat \to \alpha \\ foldn\ h\ c\ Zero &= c \\ foldn\ h\ c\ (Succ\ n) &= h\ (foldn\ h\ c\ n) \end{aligned}$$

```
val fact: Nat => Nat = {
  case Zero     => Succ(Zero)
  case Succ(n) => Succ(n) × fact(n)
}
```

```
def fact(n: Nat): Nat = {

  def snd(pair: (Nat, Nat)): Nat =
    pair match { case (_,n) => n }

  def f(pair: (Nat, Nat)): (Nat, Nat) =
    pair match { case (m,n) => (Succ(m), Succ(m) × n) }

  snd( foldn(f, (Zero, Succ(Zero)), n) )
}
```

```
val fib: Nat => Nat = {
  case Zero             => Zero
  case Succ(Zero)       => Succ(Zero)
  case Succ(Succ(n)) => fib(Succ(n)) + fib(n)
}
```

```
def fib(n: Nat): Nat = {

  def fst(pair: (Nat, Nat)): Nat =
    pair match { case (n,_) => n }

  def g(pair: (Nat, Nat)): (Nat, Nat) =
    pair match { case (m,n) => (n, m + n) }

  fst( foldn(g, (Zero, Succ(Zero)), n) )
}
```

```
def foldn[A](h: A => A, c: A, n: Nat): A =
  n match {
    case Zero => c
    case Succ(n) => h(foldn(h,c,n))
  }
```

Now let's have a very quick look at the datatype for **lists**, and at **induction over lists**.

4.1.1 Lists as a datatype

A **list** can be constructed from scratch by starting with the **empty list** and successively adding elements one by one. One can add elements to the front of the list, or to the rear, or to somewhere in the middle. In the following datatype declaration, nonempty lists are constructed by adding elements to the front of the list:

$$\mathbf{data}\ \mathit{List}\ \alpha\ \ =\ \mathit{Nil}\ |\ \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

…The constructor (short for 'construct' – the name goes back to the programming language LISP) adds an element to the front of the list. For example, the list $[1,2,3]$ would be represented as the following element of $\mathit{List}\ \mathit{Int}$:

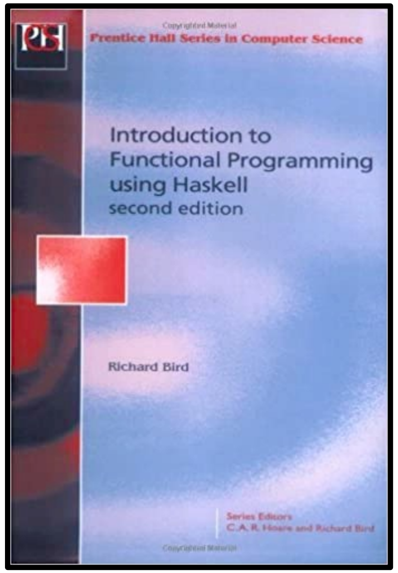
$$\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}\ ))$$

In functional programming, lists are defined as elements of $\mathit{List}\ \alpha$. The syntax $[\alpha]$ is used instead of $\mathit{List}\ \alpha$, the constructor Nil is written as $[\,]$, and the constructor Cons is written as infix operator $(:)$. Moreover, $(:)$ associates to the right, so

$$[1,2,3]\ =\ 1 : (2 : (3 : [\,]))\ =\ 1 : 2 : 3 : [\,]$$

In other words, the special syntax on the left can be regarded as an abbreviation for the syntax on the right, which is also special, but only by virtue of the fact that the constructors are given nonstandard names.

Like functions over other datatypes, functions over lists can be defined by pattern matching.



Richard Bird

Before moving on to the topic of **induction over lists**, **Richard Bird** gives an example of a function defined over **lists** using **pattern matching**, but the function he chooses is the **equality function**, whereas we are going to choose the **sum function**, just to keep things simpler.

$$
\begin{aligned}
&sum &&:: &&[Int] \rightarrow Int \\
&sum\ [\,] &&= &&0 \\
&sum\ (x : xs) &&= &&x + (sum\ xs)
\end{aligned}
$$

```
val sum : List[Int] => Int = {
  case Nil      => 0
  case x :: xs => x + sum(xs)
}
```

```
assert( sum( 1 :: (2 :: (3 :: Nil)) ) == 6)
```

Same as on the previous slide, but this time using Nat rather than Int, just for fun.

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

```
data Nat  = Zero | Succ Nat
```

```
(+)          :: Nat → Nat → Nat
m + Zero     =  m
m + Succ n   =  Succ (m + n)
```

```
sum               ::  List Nat → Nat
sum Nil           =   Zero
sum Cons x xs =   x + (sum xs)
```

```scala
sealed trait Nat
case class Succ(n: Nat) extends Nat
case object Zero extends Nat

sealed trait List[+A]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
case object Nil extends List[Nothing]

val `(+)`: Nat => Nat => Nat =
  m => { case Zero    => m
         case Succ(n) => Succ(m + n) }

implicit class NatSyntax(m: Nat){
  def +(n: Nat) = `(+)`(m)(n)
}

val sum: List[Nat] => Nat = {
  case Nil         => Zero
  case Cons(x, xs) => x + sum(xs)
}

assert( sum( Cons(
               Succ(Zero), // 1
               Cons(
                 Succ(Succ(Zero)), // 2
                 Cons(
                   Succ(Succ(Succ(Zero))), // 3
                   Nil))) )
        == Succ(Succ(Succ(Succ(Succ(Succ(Zero)))))) // 6
```

4.1.2 Induction over Lists

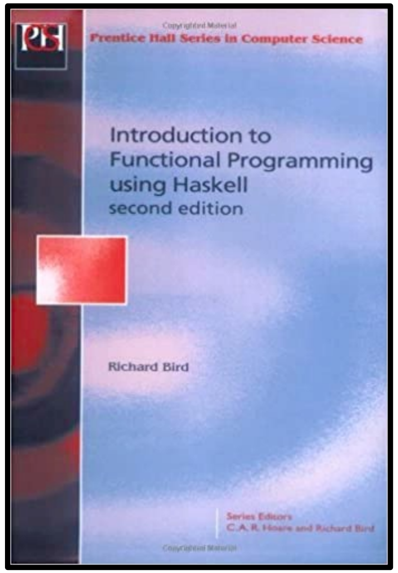
Recall from section **3.2** that, for the datatype Nat of natural numbers, structural induction is based on three cases: every element of Nat is either $\bot$, or $Zero$, or else has the form $Succ\ n$ for some element n of Nat. Similarly, structural induction on lists is also based on on three cases: every list is either the undefined list $\bot$, the empty list [ ], or else has the form $(x\!:xs)$ for some x and list xs.
To show by induction that a proposition $P(xs)$ holds for all lists xs it suffices therefore to establish three cases:

Case ($\bot$). That $P(\bot)$ holds.

Case ([ ]). That $P([\,])$ holds.

Case $(x\!:xs)$. That if $P(xs)$ holds, then $P(x\!:xs)$ also holds for every x.

If we prove only the second two cases, then we can conclude only that $P(xs)$ holds for every finite list; if we prove only the first and third cases. Then we can conclude only that $P(xs)$ holds for every partial list. If P takes the form of an equation, as all of our laws do, then proving the first and third cases is sufficient to show that $P(xs)$ holds for every infinite list. Partial lists and infinite lists are described in the following section. Examples of induction proofs are given throughout the remainder of the chapter.



Richard Bird

@philip_schwarz

Richard Bird provides other examples of **recursive functions** over **lists**. Let's see some of them: list concatenation, flattening of lists of lists, list reversal and length of a list.

When looking at the first one, i.e. concatenation, let's also see an example of **proof by structural induction** on **lists**.

4.2.1 Concatenation

Two lists can be **concatenated** to form one longer list. This function is denoted by the binary operator $+\!\!+$ (pronounced 'concatenate'). As two simple examples, we have

$?[1,2,3] +\!\!+ [4,5]$
$[1,2,3,4,5]$

$?[1,2] +\!\!+ [\,] +\!\!+ [\,1]$
$[1,2,1]$

The formal definition of $+\!\!+$ is

$$
\begin{array}{lcl}
(+\!\!+) & :: & [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
[\,] +\!\!+ ys & = & ys \\
(x{:}\,xs) +\!\!+ ys & = & x : (xs +\!\!+ ys)
\end{array}
$$

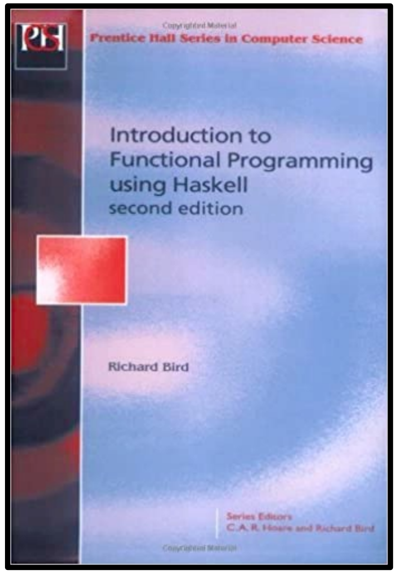
Concatenation takes two lists, both of the same type, and produces a third list, again of the same type. Hence the type assignment. The definition of $+\!\!+$ is by pattern matching on the left-hand argument; the two patterns are disjoint and cover all cases, apart from the undefined list $\bot$. It follows by case exhaustion that $\bot +\!\!+ ys = \bot$.

However, it is not the case that $ys +\!\!+ \bot = \bot$. For example,

$?[1,2,3] +\!\!+ undefined$
$[1,2,3\{Interrupted!\}]$

The list $[1,2,3] +\!\!+ \bot$ is a **partial** list; In full form it is the list $1{:}\,2{:}\,3{:}\,\bot$. The evaluator can compute the first three elements, but thereafter it goes into a **nonterminating** computation, so we interrupt it.

The second equation for $+\!\!+$ is very succinct and requires some thought. Once one has come to grips with the definition of $+\!\!+$, one



Richard Bird

has understood a good deal about how lists work in functional programming. Note that the number of steps required to compute $xs +\!\!+ ys$ is proportional to the number of elements in xs.

$$[1,\ 2] +\!\!+ [3,\ 4,\ 5]$$
= { notation}
$$(1 : (2 : [\,])) +\!\!+ (3 : (4 : (5 : [\,])))$$
= { second equation for $+\!\!+$, i.e. $(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)$ }
$$1 : ((2 : [\,]) +\!\!+ (3 : (4 : (5 : [\,]))))$$
= { second equation for $+\!\!+$ }
$$1 : (2 : ([\,] +\!\!+ (3 : (4 : (5 : [\,])))))$$
= { first equation for $+\!\!+$ i.e. , $[\,] +\!\!+ ys = ys$ }
$$1 : (2 : (3 : (4 : (5 : [\,]))))$$
= { notation}
$$[1,\ 2,\ 3,\ 4,\ 5]$$

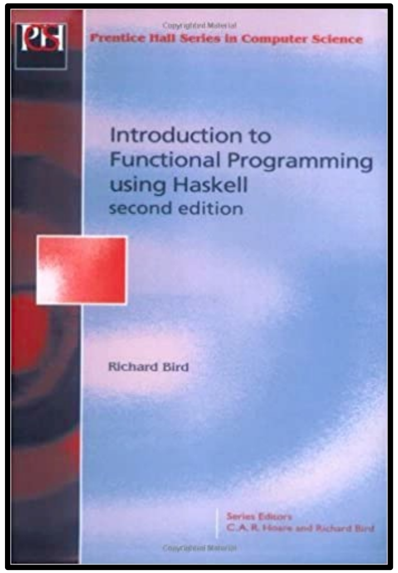
Concatenation is an **associative operation** with **unit** $[\,]$:

$$(xs +\!\!+ ys) +\!\!+ zs \;=\; xs +\!\!+ (ys +\!\!+ zs)$$
$$xs +\!\!+ [\,] \;=\; [\,] +\!\!+ xs \;=\; xs$$

Let us now prove by induction that $+\!\!+$ is **associative**.

Proof. The proof is by **induction** on xs.
Case ($\perp$). For the left-hand side, we reason

$$\perp +\!\!+ (ys +\!\!+ zs)$$
= { case exhaustion}
$$\perp +\!\!+ zs$$
= { case exhaustion}
$$\perp$$




Richard Bird

The right-hand side simplifies to $\bot$ as well, establishing the case.

Case ([ ]). For the left hand side, we reason

$$
\begin{array}{ll}
 & [\,] +\!\!+ (ys +\!\!+ zs) \\
= & \{\text{ first equation for } +\!\!+ \text{ i.e. , } [\,] +\!\!+ ys = ys \;\} \\
 & (ys +\!\!+ zs)
\end{array}
$$

The right-hand side simplifies to$(ys +\!\!+ zs)$ as well, establishing the case.

Case $(x : xs)$. For the left hand side, we reason

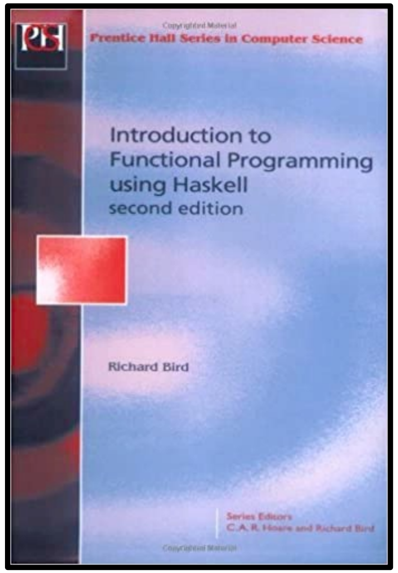
$$
\begin{array}{ll}
 & ((x : xs) +\!\!+ ys) +\!\!+ zs \\
= & \{\text{ second equation for } +\!\!+\text{, i.e. } (x{:}\, xs) +\!\!+ ys = x : (xs +\!\!+ ys) \;\} \\
 & (\, x : (xs +\!\!+ ys)) +\!\!+ zs \\
= & \{\text{ second equation for } +\!\!+ \;\} \\
 & x : ((xs +\!\!+ ys) +\!\!+ zs) \\
= & \{\text{ induction hypothesis }\} \\
 & x : (xs +\!\!+ (ys +\!\!+ zs))
\end{array}
$$

For the right-hand side we reason

$$
\begin{array}{ll}
 & (x : xs) +\!\!+ (ys +\!\!+ zs) \\
= & \{\text{ second equation for } +\!\!+\text{, i.e. } (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \;\} \\
 & x : (xs +\!\!+ (ys +\!\!+ zs))
\end{array}
$$

The two sides are equal, establishing the case.

…Note that associativity is proved for *all* lists, **finite**, **partial** or **infinite**. Hence we can assert that $+\!\!+$ is **associative** without qualification….



Richard Bird

4.2.2 Concat

Concatenation performs much the same function for lists as the **union operator** ∪ does for sets. A companion function is *concat*, which concatenates a list of lists into one long list. This function, which roughly corresponds to the **big-union operator** ⋃ for sets of sets, is defined by

$$
\begin{aligned}
&concat && :: && [[\alpha]] \rightarrow [\alpha] \\
&concat\,[\,] && = && [\,] \\
&concat\,(xs : xss) && = && xs \mathbin{+\!\!+} concat\ xss
\end{aligned}
$$

For example,

?*concat* [[1, 2], [ ], [3, 2,1]]
[1,2,3,2,1]

4.2.3 Reverse

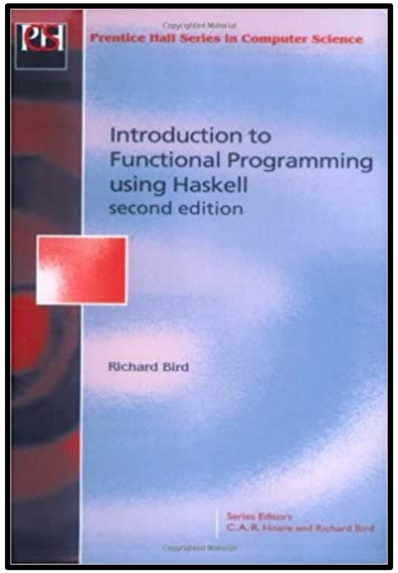
Another basic function on lists is *reverse*, the function that reverses the order of elements in a finite list. For example:

? *reverse* “Madam, I’m Adam.”
“.MadA m’I ,madaM”

The definition is

$$
\begin{aligned}
&reverse && :: && [\alpha] \rightarrow [\alpha] \\
&reverse\,[\,] && = && [\,] \\
&reverse\,(x : xs) && = && reverse\ xs \mathbin{+\!\!+} [x]
\end{aligned}
$$

In words, to reverse a list $(x : xs)$, one reverses xs, and then adds x to the end. As a program, the above definition is not very



Richard Bird

efficient: on a list of length n, it will need a number of reduction steps proportional to n^2 to deliver the reversed list. The first element will be appended to the end of a list of length $(n-1)$, which will take about $(n-1)$ steps, the second element will be appended to a list of length $(n-2)$, taking $(n-2)$ steps, and so on. The total time is therefore about

$$(n-1) + (n-2) + \cdots 1 = n(n-1)/2 \text{ steps}$$

A more precise analysis is given in chapter 7, and a more efficient program for *reverse* is given in section 4.5.

4.2.2 Length

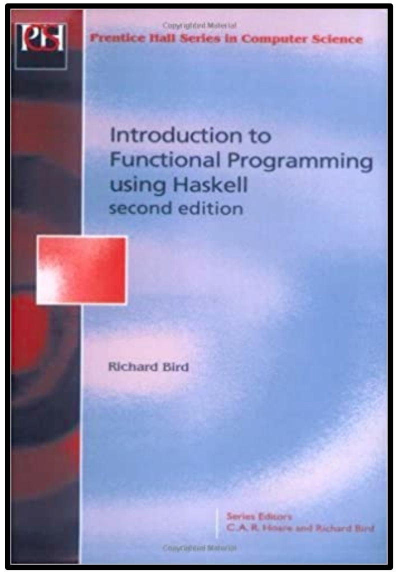
The length of a list is the number of elements it contains:

$$\begin{array}{lll} length & :: & [\alpha] \rightarrow \mathbf{Int} \\ length\ [\,] & = & 0 \\ length\ (x : xs) & = & 1 + length\ xs \end{array}$$

The nature of the list element is irrelevant when computing the length of a list, whence the type assignment. For example,

? $length\ [undefined, undefined]$
2

However, not every list has a well-defined length. In particular, the **partial** lists $\bot$, $x : \bot$, $x : y : \bot$, and so on, have an undefined length. Only **finite** lists have well-defined lengths. The list $[\bot, \bot]$ is a **finite** list, *not* a **partial** list, because it is the list $\bot : \bot : [\,]$, which ends in $[\,]$, not $\bot$. The computer cannot produce the elements, but it can produce the length of the list.
…




Richard Bird

4.3 Map and filter

Two useful functions on lists are *map* and *filter*. The function *map* applies a function to each element of a list. For example

? $map\ square\ [9, 3]$
$[81, 9]$

? $map\ (<3)\ [1, 2, 3]$
$[True, True, False]$

? $map\ nextLetter$ “HAL”
“IBM”

The definition is

$$
\begin{aligned}
&map && :: && (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&map\ f\ [\,] && = && [\,] \\
&map\ f\ (x : xs) && = && f\ x : map\ f\ xs
\end{aligned}
$$

…

4.3 filter

The second function, *filter*, takes a Boolean function p and a list xs and returns that sublist of xs whose elements satisfy p. For example,

? $filter\ even\ [1,2,4,5,32]$
$[2,4,32]$

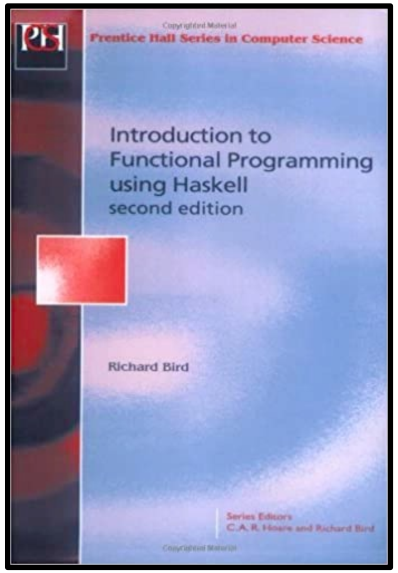
? $(sum \cdot map\ square \cdot filter\ even)\ [1..10]$
220

The last example asks for the sum of the squares of the even integers in the range 1..10.

The definition of filter is

$$
\begin{aligned}
&filter && :: && (\alpha \to Bool) \to [\alpha] \to [\alpha] \\
&filter\ p\ [\,] && = && [\,] \\
&filter\ p\ (x : xs) && = && \mathbf{if}\ p\ x\ \mathbf{then}\ x : filter\ p\ xs\ \mathbf{else}\ filter\ p\ xs
\end{aligned}
$$

…



Richard Bird


Now let's look at **fold** functions over **lists**.

4.5 The fold functions

We have seen in the case of the datatype ***Nat*** that many **recursive definitions** can be expressed very succinctly using a suitable *fold* operator. Exactly the same is true of lists. Consider the following definition of a function h :

$$h\ [\,] \quad = \quad e$$
$$h\ (x : xs) \quad = \quad x \oplus h\ xs$$

The function h works by taking a list, replacing $[\,]$ by e and $(:)$ by $\oplus$, and evaluating the result. For example, h converts the list

$$x_1 : (x_2 : (x_3 : (x_4 : [\,])))$$

to the value

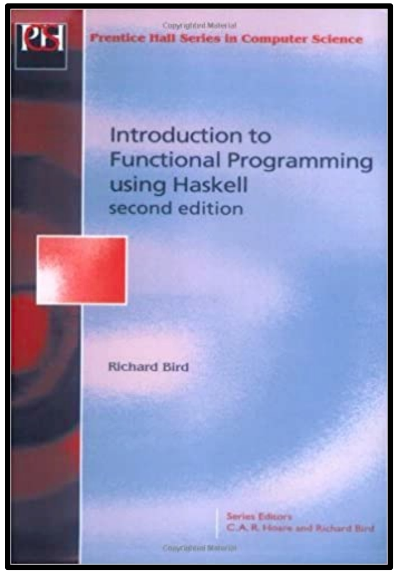
$$x_1 \oplus (x_2 \oplus (x_3 \oplus (x_4 \oplus e)))$$

Since $(:)$ **associates to the right**, there is no need to put in parentheses in the first expression. However, we do need to put in parentheses in the second expression because we do not assume that $\oplus$ **associates to the right**.

The pattern of definition given by h is captured in a function *foldr* (prounced 'fold right') defined as follows:

$$foldr \quad :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$
$$foldr\ f\ e\ [\,] \quad = e$$
$$foldr\ f\ e\ (x : xs) \quad = f\ x\ (foldr\ f\ e\ xs)$$

We can now write $\mathrm{h} = foldr\ (\oplus)\ e$. The first argument of *foldr* is a binary operator that takes an α-value on its left and an a β–value on its right, and delivers a β–value. The second argument of *foldr* is a β-value. The third argument is of type $[\alpha]$, and the result is of type β. In many cases, α and β will be instantiated to the same type, for instance when $\oplus$ denotes an **associative operation**.



Richard Bird

In the next slide we look at how some of the recursively defined functions on lists that we have recently seen can be redefined in terms of $foldr$.

To aid comprehension, I have added the original function definitions next to the new definitions in terms of $foldr$. For reference, I also added the definition of $foldr$.

The single function *foldr* can be used to define almost every function on lists that we have met so far. Here are just some examples:

$$\begin{aligned} &concat && :: && [[\alpha]] \rightarrow [\alpha] \\ &concat && = && foldr\ (+\!\!+)\ [\,] \end{aligned}$$

$$\begin{aligned} &reverse && :: && [\alpha] \rightarrow [\alpha] \\ &reverse && = && foldr\ snoc\ [\,] \\ & && && \mathbf{where}\ snoc\ x\ xs = xs +\!\!+ [x] \end{aligned}$$

$$\begin{aligned} &length && :: && [\alpha] \rightarrow \mathbf{Int} \\ &length && = && foldr\ oneplus\ 0 \\ & && && \mathbf{where}\ oneplus\ x\ n = 1 + n \end{aligned}$$

…

$$\begin{aligned} &sum && :: && [\mathbf{Int}] \rightarrow \mathbf{Int} \\ &sum && = && foldr\ (+)\ 0 \end{aligned}$$

$$\begin{aligned} &map && :: && (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ &map\ f && = && foldr\ (cons \cdot f)\ [\,] \\ & && && \mathbf{where}\ cons\ x\ xs = x : xs \end{aligned}$$

…

$$\begin{aligned} &concat && :: && [[\alpha]] \rightarrow [\alpha] \\ &concat\ [\,] && = && [\,] \\ &concat\ (xs : xss) && = && xs +\!\!+ concat\ xss \end{aligned}$$

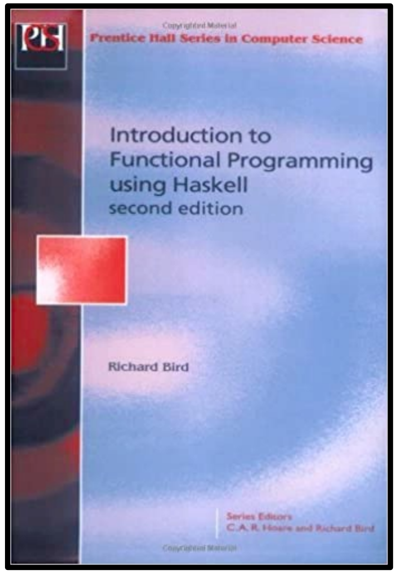
$$\begin{aligned} &reverse && :: && [\alpha] \rightarrow [\alpha] \\ &reverse\ [\,] && = && [\,] \\ &reverse\ (x : xs) && = && reverse\ xs +\!\!+ [x] \end{aligned}$$

$$\begin{aligned} &length && :: && [\alpha] \rightarrow \mathbf{Int} \\ &length\ [\,] && = && 0 \\ &length\ (x : xs) && = && 1 + length\ xs \end{aligned}$$

$$\begin{aligned} &sum && :: && [\mathbf{Int}] \rightarrow \mathbf{Int} \\ &sum\ [\,] && = && 0 \\ &sum\ (x : xs) && = && x + (sum\ xs) \end{aligned}$$

$$\begin{aligned} &map && :: && (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ &map\ f\ [\,] && = && [\,] \\ &map\ f\ (x : xs) && = && f\ x : map\ f\ xs \end{aligned}$$

$$\begin{aligned} &foldr && :: && (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta \\ &foldr\ f\ e\ [\,] && = && e \\ &foldr\ f\ e\ (x : xs) && = && f\ x\ (foldr\ f\ e\ xs) \end{aligned}$$



Richard Bird

@philip_schwarz

On the next slide, the same code translated into **Scala**

$$
\begin{aligned}
&foldr && :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta\\
&foldr\ f\ e\ [\,] && = e\\
&foldr\ f\ e\ (x{:}xs) && = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&(+\!\!+) && :: [\alpha] \to [\alpha] \to [\alpha]\\
&[\,] +\!\!+\ ys && = ys\\
&(x{:}xs) +\!\!+\ ys && = x : (xs +\!\!+\ ys)
\end{aligned}
$$

$$
\begin{aligned}
&concat && :: [[\alpha]] \to [\alpha]\\
&concat && = foldr\ (+\!\!+)\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&reverse && :: [\alpha] \to [\alpha]\\
&reverse && = foldr\ snoc\ [\,]\\
& && \quad \mathbf{where}\ snoc\ x\ xs = xs +\!\!+ [x]
\end{aligned}
$$

$$
\begin{aligned}
&length && :: [\alpha] \to \mathbf{Int}\\
&length && = foldr\ oneplus\ 0\\
& && \quad \mathbf{where}\ oneplus\ x\ n = 1 + n
\end{aligned}
$$

$$
\begin{aligned}
&sum && :: [\mathbf{Int}] \to \mathbf{Int}\\
&sum && = foldr\ (+)\ 0
\end{aligned}
$$

$$
\begin{aligned}
&map && :: (\alpha \to \beta) \to [\alpha] \to [\beta]\\
&map\ f && = foldr\ (cons \cdot f)\ [\,]\\
& && \quad \mathbf{where}\ cons\ x\ xs = x : xs
\end{aligned}
$$

```scala
def foldr[A,B](f: A => B => B)(e: B)(xs: List[A]): B =
  xs match {
    case Nil    => e
    case x::xs => f(x)(foldr(f)(e)(xs))
  }

def concatenate[A]: List[A] => List[A] => List[A] =
  xs => ys => xs match {
    case Nil => ys
    case x :: xs => x :: concatenate(xs)(ys)
  }

def concat[A]: List[List[A]] => List[A] =
  foldr(concatenate[A])(Nil)

def reverse[A]: List[A] => List[A] = {
  def snoc[A]: A => List[A] => List[A] =
    x => xs => concatenate(xs)(List(x))
  foldr(snoc[A])(Nil)
}

def length[A]: List[A] => Int = {
  def oneplus[A]: A => Int => Int = x => n => 1 + n
  foldr(oneplus)(0)
}

val sum: List[Int] => Int = {
  val plus: Int => Int => Int = a => b => a + b
  foldr(plus)(0)
}

def map[A,B]: (A => B) => List[A] => List[B] = {
  def cons: B => List[B] => List[B] = x => xs => x :: xs
  f => foldr(cons compose f)(Nil)
}
```

Here a some sample tests for the **Scala** functions on the previous slide.

```
assert( concatenate(List(1,2,3))(List(4,5)) == List(1,2,3,4,5) )

assert( concat(List(List(1,2), List(3), List(4,5))) == List(1,2,3,4,5) )

assert( reverse(List(1,2,3,4,5)) == List(5,4,3,2,1) )

assert( length(List(0,1,2,3,4,5)) == 6 )

assert( sum(List(2,3,4)) == 9 )

val mult: Int => Int => Int = a => b => a * b
assert( map(mult(10))(List(1,2,3)) == List(10,20,30))
```

@philip_schwarz

It turns out that if it is possible to define a function on lists both using a **recursive definition** and using a definition in terms of $foldr$, then **there is a technique that can be used to go from the recursive definition to the definition using** $foldr$.

I came across the technique in the following paper by the author of **Programming in Haskell**:

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

University of Nottingham, Nottingham, UK
http://www.cs.nott.ac.uk/~gmh

The tutorial (which I shall be referring to as **TUEF**), shows how to apply the technique to the **sum** function and the **map** function, which is the subject of the next five slides. **Note**: in the paper, the $foldr$ function is referred to as $fold$.

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

Graham Hutton
@haskellhutt

3 The universal property of fold

As with the **fold** operator itself, the **universal property of $fold$** also has its origins in recursion theory. The first systematic use of the **universal property** in functional programming was by Malcolm (1990a), in his generalisation of Bird and Meerten's theory of lists (Bird, 1989; Meertens, 1983) to arbitrary regular datatypes. **For finite lists**, **the universal property of $fold$ can be stated as the following equivalence between two definitions for a function g that processes lists**:

$$\begin{aligned} g\ [\,] &= v \\ g(x : xs) &= f\ x\ (g\ xs) \end{aligned} \quad \Leftrightarrow \quad g = fold\ f\ v$$

In the right-to-left direction, substituting $g = fold\ f\ v$ into the two equations for g gives the **recursive definition for $fold$**. Conversely, in the left-to-right direction the two equations for g are precisely the assumptions required to show that $g = fold\ f\ v$ using a simple **proof by induction** on **finite** lists (Bird, 1998). Taken as a whole, the **universal property** states that for **finite** lists the function $fold\ f\ v$ is not just a solution to its defining equations, but in fact the unique solution.... The **universal property of $fold$** can be generalised to handle **partial** and **infinite** lists (Bird, 1998), but for simplicity we only consider **finite** lists in this article.

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

Graham Hutton
@haskellhutt

3.3 Universality as a definition principle

As well as being used as a proof principle, the **universal property of $fold$** **can also be used as a** **definition principle** **that guides the transformation of** **recursive functions** **into definitions using** $fold$. As a simple first example, consider the recursively defined function sum that calculates the sum of a list of numbers:

$$\begin{aligned} sum &:: [Int] \rightarrow Int \\ sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

Suppose now that **we want to redefine** sum **using** $fold$. That is, we want to **solve the equation** $sum\ =\ fold\ f\ v$ **for a function** f **and a value** v. We begin by observing that **the equation matches the right-hand side of the universal property**, from which we conclude that **the equation is equivalent to the following two equations**:

$$\begin{aligned} sum\ [\,] &= v \\ sum\ (x : xs) &= f\ x\ (sum\ xs) \end{aligned}$$

$$\begin{aligned} g\ [\,] &= v \\ g\ (x : xs) &= f\ x\ (g\ xs) \end{aligned} \quad \Leftrightarrow \quad g\ =\ fold\ f\ v$$

universal property of $fold$

From the first equation and the definition of sum, it is immediate that $v = 0$.

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

Graham Hutton
@haskellhutt

From the second equation, we calculate a definition for f as follows:

$$sum\ [\,]\quad = v$$
$$sum\ (x : xs)\ = f\ x\ (sum\ xs)$$

$$
\begin{array}{ll}
 & sum\ (x : xs)\ = f\ x\ (sum\ xs) \\
\Leftrightarrow & \quad \{\ \text{Definition of } sum\ \} \\
 & x + sum\ xs\ = f\ x\ (sum\ xs) \\
\Leftarrow & \quad \{\ \dagger\ \text{Generalising } (sum\ xs) \text{ to } y\ \} \\
 & x + y = f\ x\ y \\
\Leftrightarrow & \quad \{\ \text{Functions}\ \} \\
 & f\ = (+)
\end{array}
$$

$$sum\ :: [Int] \rightarrow Int$$
$$sum\ [\,]\quad = 0$$
$$sum\ (x : xs)\ = x + sum\ xs$$

That is, **using the universal property we have calculated that**:

$$sum = fold\ (+)\ 0$$

Note that the key step (†) above in calculating a definition for f is the generalisation of the expression $sum\ xs$ to a fresh variable y. In fact, such a generalisation step is not specific to the sum function, but will be a key step in the transformation of any **recursive function** into a definition using $fold$ in this manner.

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

Graham Hutton
@haskellhutt

Of course, the *sum* example above is rather artificial, because the definition of *sum* using *fold* is immediate. However, there are many examples of functions whose definition using *fold* is not so immediate. For example, consider the recursively defined function $map\ f$ that applies a function f to each element of a list:

$$
\begin{aligned}
map &:: (\alpha \to \beta) \to ([\alpha] \to [\beta]) \\
map\ f\ [\,] &= [\,] \\
map\ f\ (x : xs) &= f\ x : map\ f\ xs
\end{aligned}
$$

To redefine $map\ f$ using *fold* we must solve the equation $map\ f = fold\ v\ g$ for a function g and a value v. By appealing to the **universal property**, we conclude that this equation is equivalent to the following two equations:

$$
\begin{aligned}
map\ f\ [\,] &= v \\
map\ f\ (x : xs) &= g\ x\ (map\ f\ xs)
\end{aligned}
$$

$$
\begin{aligned}
g\ [\,] &= v \\
g\ (x : xs) &= f\ x\ (g\ xs)
\end{aligned}
\quad \Leftrightarrow \quad g = fold\ f\ v
$$

universal property of *fold*

substitute $map\ f$ for g and g for f

From the first equation and the definition of *map* it is immediate that $v = [\,]$.

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

Graham Hutton
@haskellhutt

$$
\begin{aligned}
&map &&:: (\alpha \to \beta) \to ([\alpha] \to [\beta])\\
&map\ f\ \ [\,] &&= v\\
&map\ f\ (x : xs) &&= g\ x\ (map\ f\ xs)
\end{aligned}
$$

From the second equation, we calculate a definition for g as follows:

$$
\begin{aligned}
& map\ f\ (x : xs) = g\ x\ (map\ f\ xs)\\
\Leftrightarrow\quad & \{\text{Definition of } map\}\\
& f\ x : map\ f\ xs = g\ x\ (map\ f\ xs)\\
\Leftarrow\quad & \{\text{Generalising } (map\ f\ xs) \text{ to } ys\}\\
& f\ x : ys = g\ x\ ys\\
\Leftrightarrow\quad & \{\text{Functions}\}\\
& g = \lambda x\ ys \to f\ x : ys
\end{aligned}
$$

That is, using the **universal property** we have calculated that

$$map\ f = fold\ (\lambda x\ ys \to f\ x : ys)\ [\,]$$

In general, any function on lists that can be expressed using the $fold$ **operator can be transformed into such a definition using the universal property of** $fold$.

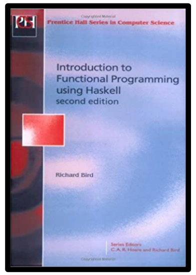
There are several other interesting things in **TUEF** that we'll be looking at.

I like its description of $foldr$ (see right), because it reiterates a key point (see left) made by **Richard Bird** about **recursive functions** on lists.

Consider the following definition of a function h :

$$h\,[\,] \quad = \quad e$$
$$h\,(x{:}\,xs) \quad = \quad x \oplus h\,xs$$

The function h works by **taking a list**, **replacing** $[\,]$ **by** e **and** $(:)$ **by** $\oplus$, **and evaluating the result**. For example, h converts the list



$$x_1 : (x_2 : (x_3 : (x_4 : [\,])))$$

to the value

$$x_1 \oplus (x_2 \oplus (x_3 \oplus (x_4 \oplus e)))$$

Since $(:)$ associates to the right, there is no need to put in parentheses in the first expression. However, we do need to put in parentheses in the second expression because we do not assume that $\oplus$ associates to the right.

The pattern of definition given by h is captured in a function $foldr$ (pronounced 'fold right') defined as follows:

$$foldr \quad :: \ (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$
$$foldr\ f\ e\ [\,] \quad = e$$
$$foldr\ f\ e\ (x{:}\,xs) \ = f\ x\ (foldr\ f\ e\ xs)$$

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

2 The fold operator

The **fold** operator has its origins in recursion theory (Kleene, 1952), while the use of **fold** as a central concept in a programming language dates back to the reduction operator of APL (Iverson, 1962), and later to the insertion operator of FP (Backus, 1978). In **Haskell**, the **fold** operator for lists can be defined as follows:

$$fold \quad :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow ([\alpha] \rightarrow \beta)$$
$$fold\ f\ v\ [\,] \quad = v$$
$$fold\ f\ v\ (x : xs) \ = f\ x\ (fold\ f\ v\ xs)$$

That is, given a function f of type $\alpha \rightarrow \beta \rightarrow \beta$ and a value v of type β, **the function** $fold\ f\ v$ **processes a list of type** $[\alpha]$ **to give a value of type** β **by replacing the nil constructor** $[\,]$ **at the end of the list by the value** v, **and each cons constructor** $(:)$ **within the list by the function** f. **In this manner, the** $fold$ **operator encapsulates a simple pattern of recursion for processing lists, in which the two constructors for lists are simply replaced by other values and functions**.

Remember the **list concatenation** function we saw earlier?

$$
\begin{aligned}
(+\!\!+) \quad &:: \quad [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
[\,] +\!\!+ ys \quad &= \quad ys \\
(x : xs) +\!\!+ ys \quad &= \quad x : (xs +\!\!+ ys)
\end{aligned}
$$

Concatenation takes two lists, both of the same type, and produces a third list, again of the same type.

```scala
def concatenate[A]: List[A] => List[A] => List[A] =
  xs => ys => xs match {
    case Nil => ys
    case x :: xs => x :: concatenate(xs)(ys)
  }
```

```scala
assert( concatenate(List(1,2,3))(List(4,5)) == List(1,2,3,4,5) )
```

In **TUEF** we find a definition of **concatenation** in terms of $foldr$ (which it calls $fold$)

$$
\begin{aligned}
(+\!\!+) \quad &:: \quad [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
(+\!\!+\ ys) \quad &= \quad fold\ (:)\ ys
\end{aligned}
$$

```scala
def concatenate[A]: List[A] => List[A] => List[A] = {
  def cons: A => List[A] => List[A] =
    x => xs => x :: xs
  xs => ys => foldr(cons)(ys)(xs)
}
```

Remember the *filter* function we saw earlier?

$$
\begin{aligned}
&\mathit{filter} &&:: && (\alpha \rightarrow \mathit{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \\
&\mathit{filter}\ p\ [\,] &&= && [\,] \\
&\mathit{filter}\ p\ (x : xs) &&= && \textbf{if}\ p\ x\ \textbf{then}\ x : \mathit{filter}\ p\ xs\ \textbf{else}\ \mathit{filter}\ p\ xs
\end{aligned}
$$

```scala
def filter[A]: (A => Boolean) => List[A] => List[A] = p => {
  case Nil      => Nil
  case x :: xs => if (p(x)) x :: filter(p)(xs) else filter(p)(xs)
}
```

```scala
val gt: Int => Int => Boolean = x => y => y > x
assert(filter(gt(5))(List(10,2,8,5,3,6)) == List(10,8,6))
```

In **TUEF** we find a definition of *filter* in terms of *foldr* (which as we saw, it calls *fold*)

$$
\begin{aligned}
&\mathit{filter} &&:: && (\alpha \rightarrow \mathit{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \\
&\mathit{filter}\ p &&= && \mathit{fold}\ (\lambda x\ xs\ \rightarrow\ \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)\ \ [\,]
\end{aligned}
$$

```scala
def filter[A]: (A => Boolean) => List[A] => List[A] = p =>
  foldr((x:A) => (xs:List[A]) => if (p(x)) (x::xs) else xs)(Nil)
```

Not every function on lists can be defined as an instance of $foldr$. For example, zip cannot be so defined. Even for those that can, an alternative definition may be more efficient. To illustrate, suppose we want a function *decimal* that takes a list of digits and returns the corresponding decimal number; thus

$$decimal\ [x_0, x_1, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{(n-k)}$$

It is assumed that the most significant digit comes first in the list. One way to compute *decimal* efficiently is by a process of multiplying each digit by ten and adding in the following digit. For example

$$decimal\ [x_0, x_1, x_2] = 10 \times (10 \times (10 \times 0 + x_0) + x_1) + x_2$$

This decomposition of a sum of powers is known as ***Horner's* rule**.

Suppose we define $\oplus$ by $n \oplus x = 10 \times n + x$. Then we can rephrase the above equation as

$$decimal\ [x_0, x_1, x_2] = ((0 \oplus x_0) \oplus x_1) \oplus x_2$$

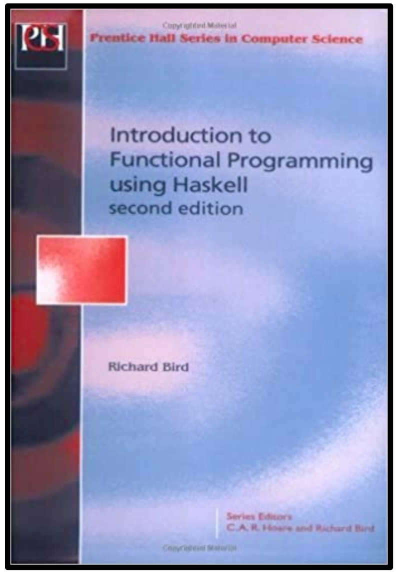
This is almost like an instance of $foldr$, except that the grouping is the other way round, and the starting value appears on the left, not on the right. In fact the computation is dual: instead of processing from right to left, the computation processes from left to right.

This example motivates the introduction of a second fold operator called $foldl$ (pronounced 'fold left'). Informally:

$$foldl\ (\oplus)\ e\ [x_0, x_1, \ldots, xn\ \_\ 1] = (\ldots((e \oplus x_0) \oplus x_1) \ldots) \oplus x_{n\ \_\ 1}$$

The parentheses group from the left, which is the reason for the name. The full definition of $foldl$ is

$$
\begin{aligned}
&foldl &&:: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldl\ f\ e\ [\,] &&= e \\
&foldl\ f\ e\ (x{:}xs) &&= foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$



Richard Bird

For example

$$
\begin{aligned}
& foldl\ (\oplus)\ \ e\ \ [x_0, x_1, x_2] \\
= \ & foldl\ (\oplus)\ (e \oplus x_0)\ [x_1, x_2] \\
= \ & foldl\ (\oplus)\ ((e \oplus x_0) \oplus x_1)\ [x_2] \\
= \ & foldl\ (\oplus)\ (((e \oplus x_0) \oplus x_1) \oplus x_2)\ \ [\,] \\
= \ & ((e \oplus x_0) \oplus x_1) \oplus x_2
\end{aligned}
$$

If $\oplus$ is **associative** with **unit** e, then $foldr\ (\oplus)\ e$ and $foldl\ (\oplus)\ e$ define the same function on **finite** lists, as we will see in the following section.

As another example of the use of $foldl$, consider the following definition:

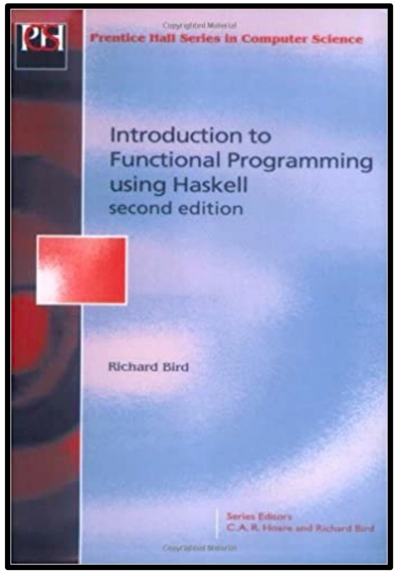
$$
\begin{aligned}
& reverse' \ :: \ [\alpha] \rightarrow [\alpha] \\
& reverse' \ = \ foldl\ cons\ [\,] \\
& \qquad\qquad \mathbf{where}\ cons\ xs\ x = x : xs
\end{aligned}
$$

$$
\begin{aligned}
& reverse \ :: \ [\alpha] \rightarrow [\alpha] \\
& reverse \ = \ foldr\ snoc\ [\,] \\
& \qquad\qquad \mathbf{where}\ snoc\ x\ xs = xs \mathbin{+\!\!+} [x]
\end{aligned}
$$

Note the order of the arguments to cons; we have $cons = flip\ (:)$, where the standard function $flip$ is defined by $flip f\ x\ y = f\ y\ x$. The function $reverse'$, reverses a finite list. For example:

$$
\begin{aligned}
& reverse'\ [x_0, x_1, x_2] \\
= \ & cons(cons(cons\ [\,]\ x_0)\ x_1)\ x_2 \\
= \ & cons(cons\ [x_1]\ x_0)\ x_2 \\
= \ & cons\ [x_1, x_0]\ x_2 \\
= \ & [x_2, x_1, x_0]
\end{aligned}
$$

One can prove that $reverse' = reverse$ **by induction**, or as an instance of a more general result in the following section. Of greater importance than the mere fact that $reverse$ can be defined in a different way, is that $reverse'$ gives a much more efficient program: $reverse'$ takes time proportional to n on a list of length n, while $reverse$ takes time proportional to n^2.



Richard Bird

@philip_schwarz

Here we can see the **Scala** version of *reverse'*, and how it compares with *reverse*

$$
\begin{aligned}
(+\!\!+) &:: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
(+\!\!+\ ys) &= \mathit{fold}\ (:)\ ys
\end{aligned}
$$

```scala
def concatenate[A]: List[A] => List[A] => List[A] = {
  def cons: A => List[A] => List[A] =
    x => xs => x :: xs
  xs => ys => foldr(cons)(ys)(xs)
}
```

$$
\begin{aligned}
\mathit{reverse} &:: [\alpha] \rightarrow [\alpha] \\
\mathit{reverse} &= \mathit{foldr}\ \mathit{snoc}\ [\,] \\
&\quad \textbf{where}\ \mathit{snoc}\ x\ xs = xs +\!\!+ [x]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reverse}' &:: [\alpha] \rightarrow [\alpha] \\
\mathit{reverse}' &= \mathit{foldl}\ \mathit{cons}\ [\,] \\
&\quad \textbf{where}\ \mathit{cons}\ xs\ x = x : xs
\end{aligned}
$$

```scala
def reverse[A]: List[A] => List[A] = {
  def snoc[A]: A => List[A] => List[A] =
    x => xs => concatenate(xs)(List(x))
  foldr(snoc[A])(Nil)
}
```

```scala
def reverse'[A]: List[A] => List[A] = {
  def cons: List[A] => A => List[A] =
    xs => x => x :: xs
  foldl(cons)(Nil)
}
```

```scala
assert( reverse(List(1,2,3,4,5)) == List(5,4,3,2,1) )
```

```scala
assert( reverse'(List(1,2,3,4,5)) == List(5,4,3,2,1) )
```

That’s it for part 1. I hope you enjoyed that.

There is still a lot to cover of course, so I’ll see you in part 2.