INTRODUCTION TO FUNCTIONAL PROGRAMMING

Graham Hutton
University of Nottingham

0

What is Functional Programming?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- A functional language is one that supports and encourages the functional style.

1

Example

Summing the integers 1 to 10 in Java:

```
total = 0;
for (i = 1; i ≤ 10; ++i)
   total = total+i;
```

The computation method is variable assignment.

2

Example

Summing the integers 1 to 10 in Haskell:

```
sum [1..10]
```

The computation method is function application.

3

Why is it Useful?

Again, there are many possible answers to this question, but generally speaking:

- The abstract nature of functional programming leads to considerably simpler programs;
- It also supports a number of powerful new ways to structure and reason about programs.

4

This Course

A series of mini-lectures reviewing a number of basic concepts, using Haskell:

- The Hugs system;
- Types and classes;
- Defining functions;
- List comprehensions;
- Recursive functions;
- Higher-order functions;
- Interactive programs;
- Defining types.

5

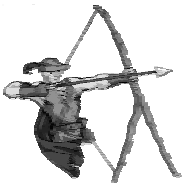
These concepts will be tied together at the end by developing a Haskell program to solve the numbers game from Countdown, a popular quiz show.

Note:

- Some prior exposure to functional programming is assumed, but not specifically Haskell;
- Please ask questions during the lectures!

6

LECTURE 1
THE HUGS SYSTEM



Graham Hutton
University of Nottingham

7

What is Hugs?

- An interpreter for Haskell, and the most widely used implementation of the language;
- An interactive system, which is well-suited for teaching and prototyping purposes;
- Hugs is freely available from:

```
www.haskell.org/hugs
```

8

The Standard Prelude

When Hugs is started it first loads the library file Prelude.hs, and then repeatedly prompts the user for an expression to be evaluated.

For example:

```
> 2+3*4
14

> (2+3)*4
20
```

9

The standard prelude also provides many useful functions that operate on lists. For example:

```
> length [1,2,3,4]
4

> product [1,2,3,4]
24

> take 3 [1,2,3,4,5]
[1,2,3]
```

10

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

```
f(a,b) + c d
```

Apply the function f to a and b, and add the result to the product of c and d.

11

In Haskell, function application is denoted using space, and multiplication is denoted using *.

```
f a b + c*d
```

As previously, but in Haskell syntax.

12

Moreover, function application is assumed to have higher priority than all other operators.

```
f a + b
```

Means (f a) + b, rather than f (a + b).

13

Examples

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x)g(y)	f x * g y

14

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running Hugs.

Start an editor, type in the following two function definitions, and save the script as test.hs:

```
double x    = x + x

quadruple x = double (double x)
```

15

Leaving the editor open, in another window start up Hugs with the new script:

```
% hugs test.hs
```

Now both Prelude.hs and test.hs are loaded, and functions from both scripts can be used:

```
> quadruple 10
40

> take (double 2) [1..6]
[1,2,3,4]
```

16

Leaving Hugs open, return to the editor, add the following two definitions, and resave:

```
factorial n = product [1..n]

average ns  = sum ns `div` length ns
```

Note:

- div is enclosed in back quotes, not forward;
- x `f` y is just syntactic sugar for f x y.

17

Hugs does not automatically reload scripts when they are changed, so a reload command must be executed before the new definitions can be used:

```
> :reload
Reading file "test.hs"

> factorial 10
3628800

> average [1..5]
3
```

18

Exercises

(1) Try out some of the other functions from the standard prelude using Hugs.

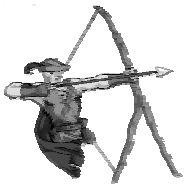
(2) Work through "My First Script" using Hugs.

(3) Show how the functions last and init from the standard prelude could be re-defined using other functions from the prelude.

Note: there are many possible answers!

19

LECTURE 2
TYPES AND CLASSES (I)



Graham Hutton
University of Nottingham

20

What is a Type?

A type is a collection of related values.

`Bool` — The logical values False and True.

`Bool → Bool` — All functions that map a logical value to a logical value.

21

Types in Haskell

We use the notation e :: T to mean that evaluating the expression e will produce a value of type T.

```
False           :: Bool

not             :: Bool → Bool

not False       :: Bool

False && True   :: Bool
```

22

Note:

- Every expression must have a valid type, which is calculated prior to evaluating the expression by a process called type inference;
- Haskell programs are type safe, because type errors can never occur during evaluation;
- Type inference detects a very large class of programming errors, and is one of the most powerful and useful features of Haskell.

23

Basic Types

Haskell has a number of basic types, including:

- `Bool` - Logical values
- `Char` - Single characters
- `String` - Strings of characters
- `Int` - Fixed-precision integers
- `Integer` - Arbitrary-precision integers

24

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

In general:

[T] is the type of lists with elements of type T.

25

Note:

- The type of a list says nothing about its length:

```
[False,True]       :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

26

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(T1,T2,...,Tn) is the type of n-tuples whose ith components have type Ti for any i in 1...n.

27

Note:

- The type of a tuple encodes its arity:

```
(False,True)       :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b'])  :: (Bool,[Char])
```

28

Function Types

A function is a mapping from values of one type to values of another type:

```
not     :: Bool → Bool

isDigit :: Char → Bool
```

In general:

T1 → T2 is the type of functions that map arguments of type T1 to results of type T2.

29

Note:

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add       :: (Int,Int) → Int
add (x,y)  = x+y

zeroto    :: Int → [Int]
zeroto n   = [0..n]
```

30

Exercises

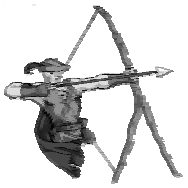
(1) What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

[isDigit,isLower,isUpper]
```

(2) Check your answers using Hugs.

31

LECTURE 3
TYPES AND CLASSES (II)



Graham Hutton
University of Nottingham

32

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add'    :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function. In turn, this function takes an integer y and returns the result x+y.

33

Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called curried functions.

34

- Functions with more than two arguments can be curried by returning nested functions:

```
mult      :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function, which in turn takes an integer y and returns a function, which finally takes an integer z and returns the result x*y*z.

35

Curry Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow → associates to the right.

```
Int → Int → Int → Int
```

Means Int → (Int → (Int → Int)).

36

- As a consequence, it is then natural for function application to associate to the left.

```
mult x y z
```

Means ((mult x) y) z.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

37

Polymorphic Types

The function length calculates the length of any list, irrespective of the type of its elements.

```
> length [1,3,5,7]
4

> length ["Yes","No"]
2

> length [isDigit,isLower,isUpper]
3
```

38

This idea is made precise in the type for length by the inclusion of a type variable:

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

A type with variables is called polymorphic.

39

Note:

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]
```

40

Overloaded Types

The arithmetic operator + calculates the sum of any two numbers of the same numeric type.

For example:

```
> 1+2
3

> 1.1 + 2.2
3.3
```

41

This idea is made precise in the type for + by the inclusion of a class constraint:

```
(+) :: Num a ⇒ a → a → a
```

For any type a in the class Num of numeric types, + takes two values of type a and returns another.

A type with constraints is called overloaded.

42

Classes in Haskell

A class is a collection of types that support certain operations, called the methods of the class.

`Eq`

Types whose values can be compared for equality and difference using

(==) :: a → a → Bool
(/=) :: a → a → Bool

43

Haskell has a number of basic classes, including:

- `Eq` - Equality types
- `Ord` - Ordered types
- `Show` - Showable types
- `Read` - Readable types
- `Num` - Numeric types

44

Example methods:

```
(==) :: Eq a   ⇒ a → a → Bool

(<)  :: Ord a  ⇒ a → a → Bool

show :: Show a ⇒ a → String

read :: Read a ⇒ String → a

(*)  :: Num a  ⇒ a → a → a
```

45

Exercises

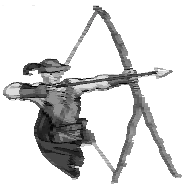
(1) What are the types of the following functions?

```
second xs        = head (tail xs)
swap (x,y)       = (y,x)
pair x y         = (x,y)
double x         = x*2
palindrome xs    = reverse xs == xs
twice f x        = f (f x)
```

(2) Check your answers using Hugs.

46

LECTURE 4
DEFINING FUNCTIONS



Graham Hutton
University of Nottingham

47

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs  :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

48

Conditional expressions can be nested:

```
signum  :: Int → Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

Note:

- In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

49

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

As previously, but using guarded equations.

50

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

Note:

- The catch all condition otherwise is defined in the prelude by otherwise = True.

51

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not      :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

52

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)           :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

53

However, the following definition is more efficient, as it avoids evaluating the second argument if the first argument is False:

```
False && _ = False
True  && b = b
```

Note:

- The underscore symbol _ is the wildcard pattern that matches any argument value.

54

List Patterns

In Haskell, every non-empty list is constructed by repeated use of an operator : called "cons" that adds a new element to the start of a list.

```
[1,2,3]
```

Means 1:(2:(3:[])).

55

The cons operator can also be used in patterns, in which case it destructs a non-empty list.

```
head       :: [a] → a
head (x:_)  = x

tail       :: [a] → [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

56

Lambda Expressions

A function can be constructed without giving it a name by using a lambda expression.

```
λx → x+1
```

The nameless function that takes a number x and returns the result x+1.

57

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

```
add x y = x+y
```

means

```
add = λx → (λy → x+y)
```

58

Lambda expressions are also useful when defining functions that return functions as results.

For example,

```
compose f g x = f (g x)
```

is more naturally defined by

```
compose f g = λx → f (g x)
```

59

Exercises

(1) Assuming that else branches were optional in conditional expressions, give an example of a nested conditional with ambiguous meaning.

(2) Give three possible definitions for the logical or operator || using pattern matching.

60

(3) Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

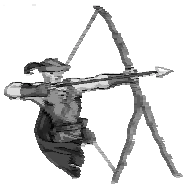
(i) a conditional expression;
(ii) guarded equations;
(iii) pattern matching.

Hint:

The prelude function null :: [a] → Bool can be used to test if a list is empty.

61

LECTURE 5
LIST COMPREHENSIONS



Graham Hutton
University of Nottingham

62

Set Comprehensions

In mathematics, the comprehension notation can be used to construct new sets from old sets.

$\{x^2 \mid x \in \{1..5\}\}$

The set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1..5}.

63

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

```
[x^2 | x ← [1..5]]
```

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

64

Note:

- The expression x ← [1..5] is called a generator, as it states how to generate values for x.
- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x ← [1..3], y ← [1..2]]

[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
```

65

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [1..2], x ← [1..3]]

[(1,1),(2,1),(3,1),(1,2),(2,2),(3,2)]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

66

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and x ≤ y.

67

Using a dependant generator we can define the library function that concatenates a list of lists:

```
concat    :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

68

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x ← [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

69

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors  :: Int → [Int]
factors n = [x | x ← [1..n]
               , n `mod` x == 0]
```

For example:

```
> factors 15

[1,3,5,15]
```

70

A positive integer is prime if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime  :: Int → Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

71

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes  :: Int → [Int]
primes n = [x | x ← [1..n], prime x]
```

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```


Exercises

(1) A pythagorean triad is triple (x,y,z) of positive integers such that $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
triads :: Int → [(Int,Int,Int)]
```

that maps a number n to the list of all triads with components in the range [1..n].

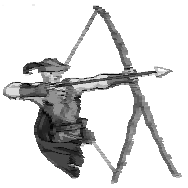
(2) A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
> perfects 500

[6,28,496]
```


LECTURE 6
RECURSIVE FUNCTIONS



Graham Hutton
University of Nottingham

Introduction

As we have seen, many functions can naturally be defined in terms of other functions.

```
factorial  :: Int → Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n.

Expressions are evaluated by a stepwise process of applying functions to their arguments.

For example:

```
  factorial 3
=
  product [1..3]
=
  product [1,2,3]
=
  1*2*3
=
  6
```

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other integer to the product of itself with the factorial of its predecessor.

78

For example:

```
  factorial 3
= 3 * factorial 2
= 3 * (2 * factorial 1)
= 3 * (2 * (1 * factorial 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

79

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions;
- In practice, however, most functions can naturally be defined in terms of themselves;
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

80

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product       :: [Int] → Int
product []     = 1
product (x:xs) = x * product xs
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

81

For example:

```
  product [1,2,3]
= product (1:(2:(3:[])))
= 1 * product (2:(3:[]))
= 1 * (2 * product (3:[]))
= 1 * (2 * (3 * product []))
= 1 * (2 * (3 * 1))
= 6
```

82

Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

83

Using recursion, this specification can be translated directly into an implementation:

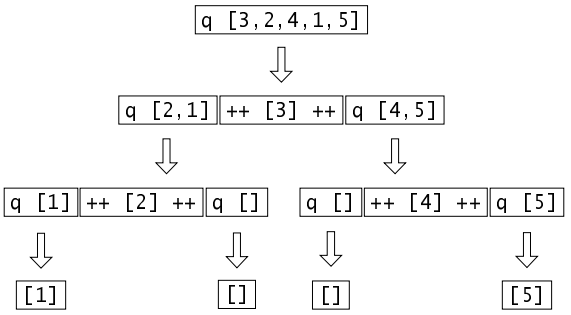
```
qsort       :: [Int] → [Int]
qsort []     = []
qsort (x:xs) = qsort [a | a ← xs, a ≤ x]
               ++ [x] ++
               qsort [b | b ← xs, b > x]
```

Note:

- This is probably the simplest implementation of quicksort in any programming language!

84

For example (abbreviating qsort as q):



85

Exercises

(1) Define a recursive function

```
insert :: Int → [Int] → [Int]
```

that inserts an integer into the correct position in a sorted list of integers. For example:

```
> insert 3 [1,2,4,5]

[1,2,3,4,5]
```

86

(2) Define a recursive function

```
isort :: [Int] → [Int]
```

that implements insertion sort, which can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail and inserting the head into the result.

87

(3) Define a recursive function

```
merge :: [Int] → [Int] → [Int]
```

that merges two sorted lists of integers to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

88

(4) Define a recursive function

```
msort :: [Int] → [Int]
```

that implements merge sort, which can be specified by the following two rules:

- Lists of length ≤ 1 are already sorted;
- Other lists can be sorted by sorting the two halves and merging the resulting lists.

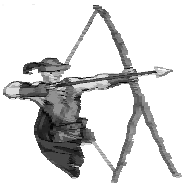
89

(5) Test both sorting functions using Hugs to see how they compare. For example:

```
> :set +s

> isort (reverse [1..500])

> msort (reverse [1..500])
```

The command :set +s tells Hugs to give some useful statistics after each evaluation.

LECTURE 7
HIGHER-ORDER FUNCTIONS



Graham Hutton
University of Nottingham

Introduction

A function is called higher-order if it takes a function as an argument or returns a function as a result.

```
twice    :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

Why Are They Useful?

- Common programming idioms, such as applying a function twice, can naturally be encapsulated as general purpose higher-order functions;
- Special purpose languages can be defined within Haskell using higher-order functions, such as for list processing, interaction, or parsing;
- Algebraic properties of higher-order functions can be used to reason about programs.

The Map Function

The higher-order library function called map applies a function to every element of a list.

```
map :: (a → b) → [a] → [b]
```

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```


The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

The Filter Function

The higher-order library function filter selects every element from a list that satisfies a predicate.

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

96

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p []     = []
filter p (x:xs)
   | p x        = x : filter p xs
   | otherwise  = filter p xs
```

97

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to a value v, and any non-empty list to a function ⊕ applied to its head and f of its tail.

98

For example:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

v = 0
⊕ = +

```
product []     = 1
product (x:xs) = x * product xs
```

v = 1
⊕ = *

```
and []     = True
and (x:xs) = x && and xs
```

v = True
⊕ = &&

99

The higher-order library function foldr ("fold right") encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

```
sum     = foldr (+) 0

product = foldr (*) 1

and     = foldr (&&) True
```

100

Foldr itself can be defined using recursion:

```
foldr (⊕) v []     = v

foldr (⊕) v (x:xs) =

   x ⊕ foldr (⊕) v xs
```

In practice, however, it is better to think of foldr non-recursively, as simultaneously replacing each cons in a list by a function, and [] by a value.

101

For example:

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

Replace each cons by + and [] by 0.

102

Why Is Foldr Useful?

- Some recursive functions on lists, such as sum, are simpler to define using foldr;
- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule;
- Advanced program optimisations can be simpler if foldr is used in place of explicit recursion.

103

Exercises

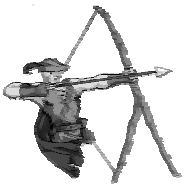
(1) What are higher-order functions that return functions as results better known as?

(2) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

(3) Show how the functions length, reverse, map f and filter p could be re-defined using foldr.

104

LECTURE 8
INTERACTIVE PROGRAMS

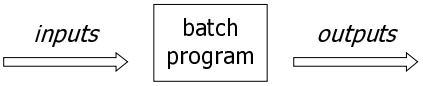


Graham Hutton
University of Nottingham

105

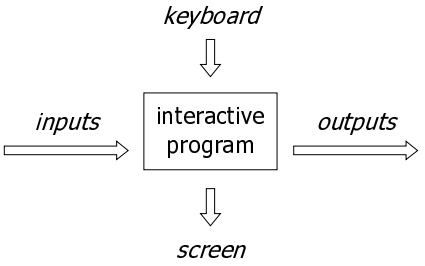
Introduction

To date, we have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end.



106

However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.



107

The Problem

Haskell programs are pure mathematical functions:

- Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

- Interactive programs have side effects.

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

```
IO a
```

The type of actions that return a value of type a.

For example:

```
IO Char
```

The type of actions that return a character.

```
IO ()
```

The type of purely side effecting actions that return no result value.

Note:

- () is the type of tuples with no components.

Primitive Actions

The standard library provides a number of actions, including the following three primitives:

- The action getChar reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

```
getChar :: IO Char
```


- The action putChar c writes the character c to the screen, and returns no result value:

```
putChar :: Char → IO ()
```

- The action return v simply returns the value v, without performing any interaction:

```
return :: a → IO a
```


Sequencing Actions

A sequence of actions can be combined as a single composite action using the keyword do.

For example:

```
getTwo :: IO (Char,Char)
getTwo  = do x ← getChar
             y ← getChar
             return (x,y)
```

Note - in a sequence of actions:

- Each action must begin in precisely the same column. That is, the layout rule applies;
- The values returned by intermediate actions are discarded by default, but if required can be named using the ← operator;
- The value returned by the last action is the value returned by the sequence as a whole.

114

Other Library Actions

- Reading a string from the keyboard:

```
getLine :: IO String
getLine  = do x ← getChar
              if x == '\n' then
                 return []
              else
                 do xs ← getLine
                    return (x:xs)
```

115

- Writing a string to the screen:

```
putStr       :: String → IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

- Writing a string and moving to a new line:

```
putStrLn   :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

116

Example

We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen  = do putStr "Enter a string: "
             xs ← getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

117

For example:

```
> strlen

Enter a string: hello there
The string has 11 characters
```

Note:

- Evaluating an action executes its side effects, with the final result value being discarded.

118

Exercise

Implement the game of nim in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

119

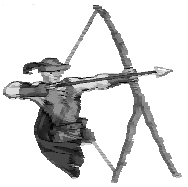
- Two players take it turn about to remove one or more stars from the end of a single row.
- The winner is the player who removes the last star or stars from the board.

Hint:

Represent the board as a list of five integers that give the number of stars remaining on each row. For example, the initial board is [5,4,3,2,1].

120

LECTURE 9
DEFINING TYPES



Graham Hutton
University of Nottingham

121

Data Declarations

A new type can be defined by specifying its set of values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

122

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers     :: [Answer]
answers      = [Yes,No,Unknown]

flip        :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

123

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square          :: Float → Shape
square n         = Rect n n

area            :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

124

Similarly, data declarations themselves can also have parameters. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
return       :: a → Maybe a
return x      = Just x

(>>=) :: Maybe a → (a → Maybe b) → Maybe b
Nothing >>= _ = Nothing
Just x  >>= f = f x
```

125

Recursive Types

In Haskell, new types can be defined in terms of themselves. That is, types can be recursive.

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

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

126

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

```
Zero
```

```
Succ Zero
```

```
Succ (Succ Zero)
```

⋮

127

- We can think of values of type Nat as natural numbers, where Zero represents 0, and Succ represents the successor function (1 +).
- For example, the value

```
Succ (Succ (Succ Zero))
```

represents the natural number

```
1 + (1 + (1 + 0))
```

=

```
3
```

128

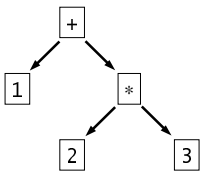
Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int          :: Nat → Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n

int2nat          :: Int → Nat
int2nat 0         = Zero
int2nat (n+1)     = Succ (int2nat n)
```

129

Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.



130

Using recursion, a suitable new type to represent such expressions can be defined by:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

131

Using recursion, it is now easy to define functions that process expressions. For example:

```
size          :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval          :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

132

Exercises

(1) Show how add can be defined using recursion rather than conversion functions.

(2) Define a suitable function fold for expressions, and show how size and eval can be re-defined more compactly using this function.

133

LECTURE 10
THE COUNTDOWN PROBLEM

Graham Hutton
University of Nottingham

134

What Is Countdown?

- A popular quiz programme on British television that has been running for almost 20 years;
- Based upon an original French version called "Des Chiffres et Des Lettres";
- Includes a numbers game that we shall refer to as the countdown problem.

135

Example

Using the numbers

1 3 7 10 25 50

and the arithmetic operators

+ - * ÷

construct an expression whose value is 765

136

Rules

- All the numbers, including intermediate results, must be integers greater than zero;
- Each of the source numbers can be used at most once when constructing the expression;
- We abstract from other rules that are adopted on television for pragmatic reasons.

137

For our example, one possible solution is

```
(25-10) * (50+1)  =  765
```

Notes:

- There are 780 solutions for this example;
- Changing the target number to `831` gives an example that has no solutions.

138

Evaluating Expressions

Operators:

```
data Op = Add | Sub | Mul | Div
```

Apply an operator:

```
apply          :: Op → Int → Int → Int
apply Add x y   = x + y
apply Sub x y   = x - y
apply Mul x y   = x * y
apply Div x y   = x `div` y
```

139

Decide if the result of applying an operator to two integers greater than zero is another such:

```
valid          :: Op → Int → Int → Bool
valid Add _ _   = True
valid Sub x y   = x > y
valid Mul _ _   = True
valid Div x y   = x `mod` y == 0
```

Expressions:

```
data Expr = Val Int | App Op Expr Expr
```

140

Return the overall value of an expression, provided that it is an integer greater than zero:

```
eval             :: Expr → [Int]
eval (Val n)      = [n | n > 0]
eval (App o l r) = [apply o x y | x ← eval l
                                , y ← eval r
                                , valid o x y]
```

Either succeeds and returns a singleton list, or fails and returns the empty list.

141

Specifying The Problem

Return a list of all possible ways of selecting zero or more elements from a list:

```
subbags :: [a] → [[a]]
```

For example:

```
> subbags [1,2]

[[],[1],[2],[3],[1,2],[2,1]]
```

142

Return a list of all the values in an expression:

```
values             :: Expr → [Int]
values (Val n)      = [n]
values (App _ l r) = values l ++ values r
```

Decide if an expression is a solution for a given list of source numbers and a target number:

```
solution       :: Expr → [Int] → Int → Bool
solution e ns n = elem (values e) (subbags ns)
                  && eval e == [n]
```

143

Brute Force Implementation

Return a list of all possible ways of splitting a list into two non-empty parts:

```
nesplit :: [a] → [([a],[a])]
```

For example:

```
> nesplit [1,2,3,4]

[([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]
```

144

Return a list of all possible expressions whose values are precisely a given list of numbers:

```
exprs    :: [Int] → [Expr]
exprs []  = []
exprs [n] = [Val n]
exprs ns  = [e | (ls,rs) ← nesplit ns
               , l       ← exprs ls
               , r       ← exprs rs
               , e       ← combine l r]
```

The key function in this lecture.

145

Combine two expressions using each operator:

```
combine    :: Expr → Expr → [Expr]
combine l r =
   [App o l r | o ← [Add,Sub,Mul,Div]]
```

Return a list of all possible expressions that solve an instance of the countdown problem:

```
solutions     :: [Int] → Int → [Expr]
solutions ns n = [e | ns' ← subbags ns
                    , e   ← exprs ns'
                    , eval e == [n]]
```

146

Correctness Theorem

Our implementation returns all the expressions that satisfy our specification of the problem:

```
elem e (solutions ns n)
```

```
solution e ns n
```

147

How Fast Is It?

System:	1GHz Pentium-III laptop
Compiler:	GHC version 5.00.2
Example:	`solutions [1,3,7,10,25,50] 765`
One solution:	0.89 seconds
All solutions:	113.74 seconds

148

Can We Do Better?

- Many of the expressions that are considered will typically be invalid - fail to evaluate;
- For our example, only around 5 million of the 33 million possible expressions are valid;
- Combining generation with evaluation would allow earlier rejection of invalid expressions.

149

Applying Program Fusion

Valid expressions and their values:

```
type Result = (Expr,Int)
```

Specification of a function that fuses together the generation and evaluation of expressions:

```
results   :: [Int] → [Result]
results ns = [(e,n) | e ← exprs ns
                    , n ← eval e]
```

150

We can now calculate an implementation:

```
results []  = []
results [n] = [(Val n,n) | n > 0]
results ns  =
   [res | (ls,rs) ← nesplit ns
        , lx      ← results ls
        , ry      ← results rs
        , res     ← combine' lx ry]
```

where

```
combine' :: Result → Result → [Result]
```

151

Return a list of all possible expressions that solve an instance of the countdown problem:

```
solutions'     :: [Int] → Int → [Expr]
solutions' ns n =
   [e | ns'   ← subbags ns
      , (e,m) ← results ns'
      , m == n]
```

Correctness Theorem:

```
solutions'  =  solutions
```

152

How Fast Is It Now?

Example: `solutions' [1,3,7,10,25,50] 765`

One solution: 0.08 seconds — Around 10 times faster.

All solutions: 5.76 seconds — Around 20 times faster.

153

Can We Do Better?

- Many expressions will be essentially the same using simple arithmetic properties, such as:

```
x * y  =  y * x
```

```
x * 1  =  x
```

- Exploiting such properties would considerably reduce the search and solution spaces.

154

Exploiting Properties

Strengthening the valid predicate to take account of commutativity and identity properties:

```
valid         :: Op → Int → Int → Bool
valid Add x y  = x ≤ y
valid Sub x y  = x > y
valid Mul x y  = x ≤ y && x ≠ 1 && y ≠ 1
valid Div x y  = x `mod` y == 0 && y ≠ 1
```

155

Using this new predicate gives a new version of our specification of the countdown problem.

Notes:

- The new specification is sound with respect to our original specification;
- It is also complete up to equivalence of expressions under the exploited properties.

156

Using the new predicate also gives a new version of our implementation, written as solutions''.

Notes:

- The new implementation requires no new proof of correctness, because none of our previous proofs depend upon the definition of valid;
- Hence our previous correctness results still hold under changes to this predicate.

157

How Fast Is It Now?

Example: `solutions'' [1,3,7,10,25,50] 765`

Valid: 250,000 expressions — Around 20 times less.

Solutions: 49 expressions — Around 16 times less.

158

One solution: 0.04 seconds — Around 2 times faster.

All solutions: 0.86 seconds — Around 7 times faster.

More generally, our program usually produces a solution to problems from the television show in an instant, and all solutions in under a second.

159

Further Work

- Using memoisation or tabulation techniques to avoiding repeated computations;
- Further reducing the search and solution spaces by exploiting extra arithmetic properties;
- Constructing an algorithm in which expressions are generated from the bottom-up.

160