Functional Programming (CS4011)

Andrew Butterfield    Glenn Strong

Foundations & Methods Group,
Discipline of Software Systems
Trinity College, University of Dublin

{Andrew.Butterfield,Glenn.Strong}@cs.tcd.ie

September 28, 2010
What is a functional programming language?

- Basic notion of computation: the application of functions to arguments.
- Basic idea of program: writing function definitions
- Functional languages are declarative: *what* rather than *how*.
- This course covers the Haskell language
  Best known and most widely supported functional programming language.
What is not a functional language?

- **Imperative** programming languages
  - C, C++, Java, Perl, Python, Assembler, etc.
  - Basic notion of computation: issuing commands to change variable values
  - Basic idea of program: writing assignment \( a := b \) and control flow (while, for, if, switch) statements

- other **Declarative** languages
  - e.g. Prolog (multi-directional, logic-based)
Why functional?

- Concise. Shorter, clearer, more maintainable code.
- Understandable. Emphasis on what, not how results in programs that are easier to understand.
- Fewer errors, higher reliability ("No core dumps!").
- Powerful abstractions. There is a smaller "semantic gap" between the programmer and the language.

Disadvantages?

- Lose some ability to do detailed low-level tuning of algorithms (bit twiddling is much harder in Haskell).
- Slower (but often not by much).
What does it look like?

Here are some basic Haskell functions being applied:

```haskell
> 3+1
4
> "this" == "that"
False
> reverse "that"
"taht"
>
These examples were executed in GHCi, a Haskell interpreter. Symbols +, == and reverse are standard builtin functions in Haskell.
Defining Haskell values

- Function definitions are written as an equations
  - name on the left hand side
  - value or expression on the right
- phrase = "able was i ere i saw elba"
- This is a name definition, not an assignment statement.

On loading this definition, we can use it:

> phrase
"able was i ere i saw elba"
Defining Haskell functions

Functions are parameterised values, which will allow us to write rather more useful definitions.

double \ x = x + x

The function can be applied to a number to produce another number.

\[ \text{> double 2} \]
\[ 4 \]
\[ \text{> double (double 2)} \]
\[ 8 \]

Of course, functions can be defined in terms of other functions:

quadruple \ x = double (double \ x)
A Note on Haskell function notation

- Function Application is ubiquitous in Haskell
- So applying function `fun` to argument `arg` is written
  
  ```
  fun arg
  ```

  The normal mathematical/prog.-lang. convention would be
  ```
  fun(arg)
  ```

- Function with multiple arguments have them separated by spaces:
  ```
  fun arg1 arg2 arg3
  ```

  (instead of `fun(arg1,arg2,arg3)`)

- Function application in Haskell is so common it has the simplest syntax possible.
Functional Highlights

We are now going to do a whistle-stop tour of some of the forthcoming highlights of functional programming, including

- Pattern-Matching in Definitions
- Higher-Order Functions
- Types
  - Inference
  - Polymorphism
  - Classes
- Laziness
- Program Compactness
- etc ...
A key datastructure in Haskell is the *List*:  

- A list of integers:  
  
  [ -3, -2, -1, 0, 1, 2, 3 ]  

- A list of characters:  
  
  [ 'h', 'e', 'l', 'l', 'o' ]  

For character lists (e.g. *Strings*) we use *syntactic sugar*:  

"hello"  

- We can think of a list as being either:  
  
  - Empty— []  
  - An element stuck ("consed") onto the front of a list— x:xs  
  
  So the notation [1,2,3] is syntactic sugar for 1:(2:(3:[])).
A key datastructure in Haskell is the List:

- A list of integers:
  \[
  [-3, -2, -1, 0, 1, 2, 3]
  \]
- A list of characters:
  \[
  ['h', 'e', 'l', 'l', 'o']
  \]

For character lists (e.g. Strings) we use syntactic sugar: "hello"
A key datastructure in Haskell is the *List*:

- A list of integers:
  
  \[
  \left[ -3, -2, -1, 0, 1, 2, 3 \right]
  \]

- A list of characters:
  
  \[
  \left[ 'h', 'e', 'l', 'l', 'o' \right]
  \]

For character lists (e.g. Strings) we use *syntactic sugar*: "hello"

- We can think of a list as being either:
Lists

A key datastructure in Haskell is the *List*:

- A list of integers:
  
  \[ [-3, -2, -1, 0, 1, 2, 3] \]

- A list of characters:
  
  \[ ['h', 'e', 'l', 'l', 'o'] \]

For character lists (e.g. Strings) we use *syntactic sugar*: "hello"

- We can think of a list as being either:
  
  - Empty— []
A key datastructure in Haskell is the List:

- A list of integers:
  \[
  [ -3, -2, -1, 0, 1, 2, 3 ]
  \]

- A list of characters:
  \[
  [ 'h', 'e', 'l', 'l', 'o' ]
  \]

For character lists (e.g. Strings) we use syntactic sugar:
"hello"

- We can think of a list as being either:
  - Empty— []
  - An element stuck ("consed") onto the front of a list— \[ x : xs \]
A key datastructure in Haskell is the *List*:

- A list of integers:
  
  \[ [-3, -2, -1, 0, 1, 2, 3] \]

- A list of characters:
  
  \[ ['h', 'e', 'l', 'l', 'o'] \]

For character lists (e.g. Strings) we use *syntactic sugar*:

"hello"

- We can think of a list as being either:
  
  - Empty— \[
  \]
  - An element stuck ("consed") onto the front of a list— \(x:xs\)

- So the notation \([1,2,3]\) is syntactic sugar for \(1:(2:(3:[])))\).
We shall define a function to compute the length of a list.

- For an empty list, its length is 0:
  \[
  \text{length } [] = 0
  \]

- For a list with a head and a tail, its length is one greater than the length of the tail:
  \[
  \text{length } (x:xs) = 1 + \text{length } xs
  \]

Recursion is the natural way to describe repeated computation. Without pattern matching, we would have been forced to write something like:

\[
\text{length } xs = \begin{cases} 
0 & \text{if } \text{null } xs \\
1 + \text{length } (\text{tail } xs) & \text{otherwise}
\end{cases}
\]
Functions on Lists (I)

- We shall define a function to compute the length of a list
- For lists we can use *pattern-matching*
Functions on Lists (I)

- We shall define a function to compute the length of a list.
- For lists we can use *pattern-matching*.
- An empty list has length 0.
  
  \[
  \text{length} \ [\ ] = 0
  \]
We shall define a function to compute the length of a list. For lists we can use pattern-matching. An empty list has length 0:

\[
\text{length } [] = 0
\]

An empty list has length one greater than its “tail”:

\[
\text{length } (x:xs) = 1 + \text{length } xs
\]
We shall define a function to compute the length of a list.

For lists we can use *pattern-matching*

An empty list has length 0

\[
\text{length } \texttt{[]} = 0
\]

An empty list has length one greater than its “tail”

\[
\text{length } \texttt{(x:xs)} = 1 + \text{length } \texttt{xs}
\]

Recursion is the natural way to describe repeated computation.
We shall define a function to compute the length of a list.

For lists we can use *pattern-matching*.

An empty list has length 0:

\[
\text{length } \text{[]} = 0
\]

An empty list has length one greater than its “tail”:

\[
\text{length } (\text{x} : \text{xs}) = 1 + \text{length } \text{xs}
\]

Recursion is the natural way to describe repeated computation.

Without pattern matching we would have been forced to write something like:

\[
\text{length } \text{xs} = \text{if } \text{null } \text{xs} \text{ then } 0 \text{ else } 1 + \text{length } (\text{tail } \text{xs})
\]
Functions on Lists (II)

- Flushed with success . . .
Functions on Lists (II)

- Flushed with success . . .
- Define a function to compute the sum of a (numeric) list

```haskell
sum [] = 0
sum (n:ns) = n + sum ns
```

- Define a function to compute the product of a (numeric) list

```haskell
prod [] = 1
prod (n:ns) = n * prod ns
```
Functions on Lists (II)

- Flushed with success . . .
- Define a function to compute the sum of a (numeric) list

\[ \text{sum} \left[ \right] = 0 \]
\[ \text{sum} \left( n:ns \right) = n + \text{sum} \ ns \]
Flushed with success . . .

Define a function to compute the sum of a (numeric) list

\[
\begin{align*}
\text{sum} & \quad [ ] \quad = 0 \\
& \quad \text{sum} \ (n : ns) \quad = \ n \ + \ \text{sum} \ ns
\end{align*}
\]

Define a function to compute the product of a (numeric) list
Functions on Lists (II)

- Flushed with success . . .
- Define a function to compute the sum of a (numeric) list
  - \[
  \text{sum} \; [] = 0 \\
  \text{sum} \; (n:ns) = n + \text{sum} \; ns
  \]
- Define a function to compute the product of a (numeric) list
  - \[
  \text{sum} \; [] = 1 \\
  \text{sum} \; (n:ns) = n * \text{sum} \; ns
  \]
Functions on Lists (II)

- Flushed with success . . .
- Define a function to compute the sum of a (numeric) list
  \[
  \begin{align*}
  \text{sum } [ & ] = 0 \\
  \text{sum } (n:ns) & = n + \text{sum } ns
  \end{align*}
  \]
- Define a function to compute the product of a (numeric) list
  \[
  \begin{align*}
  \text{sum } [ & ] = 1 \\
  \text{sum } (n:ns) & = n * \text{sum } ns
  \end{align*}
  \]
- Notice a pattern ?
Higher Order Functions

- We can define functions that

fold $e \ op \ [] = e$

fold $e \ op \ (x:xs) = op x (\text{fold} \ e \ op \ xs)$

We have capture the recursion pattern common to all three functions.
Higher Order Functions

- We can define functions that
  - take other functions as arguments

Consider `length`, `sum` and `prod`:

- they had a specific value for empty list (call it `e`)
- they had a specific function to combine the "head" element with the recursive result (call it `op`)

Let us wrap this up as a special function that takes `e` and `op` as arguments:

```fold e op [] = e
fold e op (x:xs) = op x (fold e op xs)
```

We have capture the recursion pattern common to all three functions.
Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results
Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results

- Consider length, sum and prod:

  fold e op [] = e
  fold e op (x:xs) = op x (fold e op xs)

- We have capture the recursion pattern common to all three functions.
Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results
- Consider length, sum and prod:
  - they had a specific value for empty list (call it $e$)
Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results

- Consider length, sum and prod:
  - they had a specific value for empty list (call it e)
  - they had a specific function to combine the “head” element with the recursive result (call it op)
Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results
- Consider length, sum and prod:
  - they had a specific value for empty list (call it e)
  - they had a specific function to combine the “head” element with the recursive result (call it op)
- Let us wrap this up as a special function that takes e and op as arguments:

\[
\text{fold } e \ \text{op} \ [\ ] = e \\
\text{fold } e \ \text{op} \ (x:xs) = \text{op } x \ (\text{fold } e \ \text{op} \ xs)
\]
Higher Order Functions

- We can define functions that
  - take other functions as arguments
  - return functions as results
- Consider length, sum and prod:
  - they had a specific value for empty list (call it e)
  - they had a specific function to combine the “head” element with the recursive result (call it op)
- Let us wrap this up as a special function that takes e and op as arguments:

  \[
  \text{fold } e \text{ op } [] = e \\
  \text{fold } e \text{ op } (x:xs) = \text{op } x \text{ (fold } e \text{ op } xs)
  \]

- We have capture the recursion pattern common to all three functions.
Using fold

- For `sum`, we use 0 for the empty list, and add to combine values in the recursive case
  \[
  \text{sum} = \text{fold} \ 0 \ (+)
  \]

  Note we can pass the \(+\) function in as

- For `prod`, we use 1 for the empty list, and multiply to combine values in the recursive case
  \[
  \text{prod} = \text{fold} \ 1 \ (*)
  \]

- For `length`, we use 0 for the empty list, and add one to combine values in the recursive case
  \[
  \text{prod} = \text{fold} \ 1 \ \text{incsnd}
  \]
  \[
  \text{incsnd} \ x \ y = y + 1
  \]

  Here we need to define `incsnd` to ignore its first argument and increment the second
Types in Haskell

- Haskell is a strongly typed language
Types in Haskell

- Haskell is a strongly typed language
- Every value has a type:
Types in Haskell

- Haskell is a strongly typed language
- Every value has a type:
  - `Int` (machine-width integer)
  - `Integer` (big-int — grows in size)
  - `Char` (characters)
  - `[Int]` list of `Int`
  - `([Int] -> Int)` — function from `Int`-list to `Int`
Haskell is a strongly typed language

Every value has a type:
- Int (machine-width integer)
- Integer (big-int — grows in size)
Types in Haskell

- Haskell is a strongly typed language
- Every value has a type:
  - `Int` (machine-width integer)
  - `Integer` (big-int — grows in size)
  - `Char` (characters)

- Haskell can infer types itself (Type Inference)
  - It is rare that the programmer is required to specify these!
Haskell is a strongly typed language

Every value has a type:
- Int (machine-width integer)
- Integer (big-int — grows in size)
- Char (characters)
- [Int] list of Int
Types in Haskell

- Haskell is a strongly typed language
- Every value has a type:
  - `Int` (machine-width integer)
  - `Integer` (big-int — grows in size)
  - `Char` (characters)
  - `[Int]` list of `Int`
  - `[Int] -> Int` – function from Int-list to Int.
Types in Haskell

- Haskell is a strongly typed language
- Every value has a type:
  - Int (machine-width integer)
  - Integer (big-int — grows in size)
  - Char (characters)
  - [Int] list of Int
  - [Int] -> Int — function from Int-list to Int.
- But where are types declared?
Types in Haskell

- Haskell is a strongly typed language
- Every value has a type:
  - `Int` (machine-width integer)
  - `Integer` (big-int — grows in size)
  - `Char` (characters)
  - `[Int]` list of `Int`
  - `[Int] -> Int` — function from `Int-list` to `Int`.
- But where are types declared?
- Haskell can infer types itself (Type Inference)
  It is rare that the programmer is required to specify these!
Type Polymorphism

- What is the type of `length`?

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [[],[1,2],[3,2,1],[],[6,7,8]]
5
```

- `length` works for lists of elements of arbitrary type

```
length :: [a] -> Int
```

- Here 'a' denotes a type variable, so the above reads as "`length` takes a list of (arbitrary) type `a` and returns an `Int`".

- A similar notion to "generics" in O-O languages, but builtin without fuss.
Type Polymorphism

- What is the type of `length`?

```haskell
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [[],[1,2],[3,2,1],[],[6,7,8]]
5
```
What is the type of length?

```
> length [1,2,3]
3
> length ['a','b','c','d']
4
> length [[],[1,2],[3,2,1],[],[6,7,8]]
5
```

length works for lists of elements of arbitrary type:

```
length :: [a] -> Int
```
Type Polymorphism

- What is the type of length?
  - > length [1,2,3]
    3
  - > length ['a','b','c','d']
    4
  - > length [[],[1,2],[3,2,1],[],[6,7,8]]
    5

- length works for lists of elements of arbitrary type
  length :: [a] -> Int

- Here ‘a’ denotes a type variable, so the above reads as
  “length takes a list of (arbitrary) type a and returns an Int”.
Type Polymorphism

What is the type of length?

- \texttt{length [1,2,3]} 3
- \texttt{length ['a','b','c','d']} 4
- \texttt{length [[],[1,2],[3,2,1],[],[6,7,8]]} 5

\texttt{length} works for lists of elements of arbitrary type

\texttt{length :: [a] -> Int}

Here \texttt{"a"} denotes a type variable, so the above reads as "\texttt{length} takes a list of (arbitrary) type \texttt{a} and returns an \texttt{Int}".

A similar notion to "generics" in O-O languages, but builtin without fuss.
Laziness

What's wrong with the following (recursive) definition?

\[
\text{from } n = n : (\text{from } (n+1))
\]
Laziness

- What’s wrong with the following (recursive) definition?
  \[
  \text{from } n = n : (\text{from } (n+1))
  \]

- Nothing! (Provided we do not try to evaluate all of it (???)

Haskell is a lazy language, so values are evaluated only when needed. How? — later on in the course...
**Laziness**

- What’s wrong with the following (recursive) definition?
  
  \[
  \text{from } n = n : (\text{from } (n+1))
  \]

- Nothing! (Provided we do not try to evaluate all of it (???) )

- Builtin function `take` takes a number and list as arguments:
  
  \[
  \text{take } n \text{ list} — \text{return first } n \text{ elements of list.}
  \]
What’s wrong with the following (recursive) definition?
\[
\text{from } n = n : (\text{from } (n+1))
\]

Nothing! (Provided we do not try to evaluate all of it (???).)

Builtin function take takes a number and list as arguments:
\[
\text{take } n \text{ list} \quad \text{return first } n \text{ elements of list.}
\]

What is take 10 (from 1)?
Laziness

▶ What’s wrong with the following (recursive) definition?

\[ \text{from } n = n : (\text{from } (n+1)) \]

▶ Nothing! (Provided we do not try to evaluate all of it (???)�)

▶ Builtin function \texttt{take} takes a number and list as arguments:

\[ \text{take } n \text{ list} \quad \text{— return first } n \text{ elements of list.} \]

▶ What is \texttt{take 10 (from 1)}?

▶

\[ > \text{take 10 (from 1)} \]

\[ [1,2,3,4,5,6,7,8,9,10] \]
Laziness

- What’s wrong with the following (recursive) definition?
  
  ```haskell
  from n = n : (from (n+1))
  ```

- Nothing! (Provided we do not try to evaluate all of it (???)

- Builtin function `take` takes a number and list as arguments:
  
  `take n list` — return first `n` elements of `list`.

- What is `take 10 (from 1)`?
  
  ```haskell
  > take 10 (from 1)
  [1,2,3,4,5,6,7,8,9,10]
  ```

- Haskell is a lazy language, so values are evaluated only when needed.
What’s wrong with the following (recursive) definition?

\[
\text{from } n = n : (\text{from } (n+1))
\]

Nothing! (Provided we do not try to evaluate all of it (???)�)

Builtin function \text{take} takes a number and list as arguments:
\[
\text{take } n \text{ list} \quad \text{— return first } n \text{ elements of list.}
\]

What is \text{take 10 (from 1)}?

\[
> \text{take 10 (from 1)}
\]
\[
[1,2,3,4,5,6,7,8,9,10]
\]

Haskell is a \textit{lazy} language, so values are evaluated only when needed.

How? — later on in the course ...
Program Compactness

- A key advantage of Haskell is its compactness.
Program Compactness

- A key advantage of Haskell is its compactness.
- Sorting the empty list gives the empty list:
  \[ \text{qsort \ [\] = \[\]} \]
A key advantage of Haskell is its compactness.

Sorting the empty list gives the empty list:
\[
\text{qsort} \; [] = []
\]

Sorting a nonempty list uses the head as pivot, and partitions the rest into elements less than or greater than the pivot:
\[
\text{qs} \; (x:x:s) \\
= \text{qs} \; [y \mid y \leftarrow x:s, y < x] + [x] + \text{qs} \; [z \mid z \leftarrow x:s, x \leq z]
\]
A key advantage of Haskell is its compactness.

Sorting the empty list gives the empty list:

```haskell
qsort [] = []
```

Sorting a non empty list uses the head as pivot, and partitions the rest into elements less than or greater than the pivot:

```haskell
qs (x:xs)
  = qs [y | y <- xs, y < x] ++ [x] ++ qs [z | z <- xs, x <= z]
```

We have used Haskell list comprehensions

```haskell
[y | y <- xs, y < x]
```

“build list of ys, where y is drawn from xs, such that y < x”
A key advantage of Haskell is its compactness.

Sorting the empty list gives the empty list:
\[ \text{qsort } [] = [] \]

Sorting a non-empty list uses the head as pivot, and partitions the rest into elements less than or greater than the pivot:
\[
\text{qs } (x:xs) \\
= \text{qs } [y \mid y < x]++[x]++ \text{qs } [z \mid z < x, x <= z]
\]

We have used Haskell list comprehensions
\[
[y \mid y < x ]
\]
“build list of y's, where y is drawn from xs, such that y < x”

Try that in Java!
Whistle . . . Stop!

- Enough said!
- Haskell is powerful, and quite different to most mainstream languages
- It allows very powerful programs to be written in a concise manner
- It has many real-world features as well that we shall uncover:
  - File I/O
  - GUI design
  - optimizations
Course Structure

▶ Timetable:
  ▶ Tue 4pm LB 1.20 : Lecture
  ▶ Wed 3pm LB 0.1 : Lecture
  ▶ Wed 5pm LB 0.1 : Lecture/Tutorial
  ▶ Fri 12noon ICTLAB2 : Lab Class (2 hours)

▶ Assessment
  ▶ Exam : 75%
  ▶ Lab work : 10%
  ▶ Project : 15
Any Questions?