Lecture 1 — Functional Programming

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 1

- **From Imperative to Functional Programming:**
  - What is imperative programming?
  - What is functional programming?
- **Key Ideas in Functional Programming:**
  - **Types:** Provide the data for our programs
  - **Functions:** These are our programs!
- **Advantages:**
  - Haskell code is typically short
  - Haskell code is close to the algorithms used

What is Imperative Program — Adding up square numbers

- **Problem:** Add up the first `n` square numbers

$$\texttt{ssquares n} = 0^2 \; + \; 1^2 \; + \; \ldots \; + \; + \; \texttt{(n-1)}^2 \; + \; \texttt{n}^2$$

- **Program:** We could write the following in Java

```
public int ssquares(int n){
private int s,i;
s=0; i=0;
      while (i<n) {i:=i+1;s:=s+i*i;}
}
```

- **Execution:** We may visualize running the program as follows

(Stack) Memory				(Stack) Memory
`s = ??`	$\longrightarrow$	`ssquares 4`	$\longrightarrow$	`s = 30`
`i = ??`	$\longrightarrow$		$\longrightarrow$	`i = 4`

- **Key Idea:** Imperative programs transform the memory

The Two Aspects of Imperative Programs

- **Functional Content:** What the program achieves
  - Programs take some input values and return an output value
  - `ssquares` takes a number and returns the sum of the squares up to and including that number

- **Implementational Content:** How the program does it
  - Imperative programs transform the memory using variable declarations and assignment statements
  - `ssquares` uses variables `i` and `s` to represent locations in memory. The program transforms the memory until `s` contains the correct number.

What is Functional Programming?

- **Motivation:** Problems arise as programs contain two aspects:
  - High-level algorithms and low-level implementational features
  - Humans are good at the former but not the latter
- **Idea:** The idea of functional programming is to
  - Concentrate on the functional (I/O) behaviour of programs
  - Leave memory management to the language implementation
- **Summary:** Functional languages are more abstract and avoid low level detail.

A Functional Program — Summing squares in Haskell

- **Types:** First we give the type of summing-squares

```
hssquares ::  Int -> Int
```

- **Functions:** Our program is a function

```
hssquares 0 = 0
hssquares n = n*n + hssquares(n-1)
```

- **Evaluation:** Run the program by expanding definitions

```
hssquares 2  ⇒  2*2 + hssquares 1
             ⇒  4 + (1*1 + hssquares 0)
             ⇒  4 + (1 + 0)  ⇒  5
```

- **Comment:** No mention of memory in the code.

Key Ideas in Functional Programming I — Types

- **Motivation:** Recall from CO1003/4 that types model data.
- **Integers**: `Int` is the Haskell type $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- **String**: `String` is the Haskell type of lists of characters.
- **Complex Datatypes**: Can be made from the basic types, eg lists of integers.
- **Built in Operations ("Functions on types"):**
  - Arithmetic Operations: `+ * - div mod abs`
  - Ordering Operations: `> >= == /= <= <`

Key Ideas in Functional Programming II — Functions

- **Intuition:** Recall from CO1011, a function $f\colon a \to b$ between sets associates to <u>every</u> input-value a <u>unique</u> output-value

$$x \in a \longrightarrow \boxed{\text{Function } f} \xrightarrow{?} y \in b$$

- **Example:** The *square* and *cube* functions are written

```
square ::  Int -> Int      cube ::  Int -> Int
square x = x * x           cube x = x * square x
```

- **In General:** In Haskell, functions are defined as follows

```
          ⟨function-name⟩  ::  ⟨input type⟩->⟨output type⟩
⟨function-name⟩ ⟨variable⟩  =  ⟨expression⟩
```

Functions with Multiple Arguments

- **Intuition:** A function f with n inputs is written `f::a1->...-> an-> a`

$$\begin{array}{ccc} x_1 \in a_1 & \longrightarrow & \\ x_2 \in a_2 & \longrightarrow & \\ \vdots & \vdots & \boxed{\text{Function } f} \xrightarrow{?} y \in a \\ x_n \in a_n & \longrightarrow & \end{array}$$

- **Example:** The "distance" between two integers

```
diff ::  Int -> Int -> Int
diff x y = abs (x - y)
```

- **In General:**

$\langle$`function-name`$\rangle$ :: $\langle$`type 1`$\rangle$`->`...`->`$\langle$`type n`$\rangle$`->`$\langle$`output-type`$\rangle$

$\langle$`function-name`$\rangle$ $\langle$`variable 1`$\rangle$ ... $\langle$`variable n`$\rangle$ = $\langle$`expression`$\rangle$

Key Idea III — Expressions

- **Motivation:** Get the *result/output* of a function by *applying* it to an *argument/input*
  - Write the function name followed by the input
- **In General:** Application is governed by the typing rule
  - If `f` is a function of type `a->b`, and `e` is an expression of type `a`,
  - then `f e` is the result of applying `f` to `e` and has type `b`
- **Key Idea:** Expressions are fragments of code built by applying functions to arguments.

```
square 4            square (3 + 1)    square 3 + 1
cube (square 2)     diff 6 7          square 2.2
```

- **More Expressions:** Use quotes to turn functions into infix operations and brackets to turn infix operations into functions

```
5 * 4       (*) 5 4    mod 13 4      13 `mod` 4
5-(3*4)     (5-3)*4    7 >= (3*3)    5 * (-1)
```

- **Precedence:** Usual rules of precedence and bracketing apply

- **Example of Evaluation:**

```
cube(square3) ⇒ (square 3) * square (square 3)
              ⇒ (3*3) * ((square 3) * (square 3))
              ⇒ 9 * ((3*3) * (3*3))
              ⇒ (9 * (9*9)
              ⇒ 729
```

- The final outcome of an evalution is called a *value*

Summary — Comparing Functional and Imperative Programs

- **Difference 1:** Level of Abstraction
  - Imperative Programs include low level memory details
  - Functional Programs describe only high-level algorithms
- **Difference 2:** How execution works
  - Imperative Programming based upon memory transformation
  - Functional Programming based upon expression evaluation
- **Difference 3:** Type systems
  - Type systems play a key role in functional programming

Today You Should Have Learned ...

- **Types:** A type is a collection of data values

- **Functions:** Transform inputs to outputs
  - We build complex expressions by defining functions and applying them to other expressions
  - The simplest (evaluated) expressions are (data) values

- **Evaluation:** Calculates the result of applying a function to an input
  - Expressions can be evaluated by hand or by HUGS to values

- **Now:** Go and look at the first practical!

Lecture 2 — More Types and Functions

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 2

- **New Types:** Today we shall learn about the following types
  - The type of booleans: `Bool`
  - The type of characters: `Char`
  - The type of strings: `String`
  - The type of fractions: `Float`
- **New Functions and Expressions:** And also about the following functions
  - Conditional expressions and guarded functions
  - Error handling and local declarations

Booleans and Logical Operators

- **Values of** `Bool` **:** Contains two values — `True, False`

- **Logical Operations:** Various built in functions

```
&&  :: Bool -> Bool -> Bool
||  :: Bool -> Bool -> Bool
not :: Bool -> Bool
```

- **Example:** Define the exclusive-OR function which takes as input two booleans and returns `True` just in case they are different

```
exOr :: Bool -> Bool -> Bool
```

Conditionals — If statements

- **Example:** Maximum of two numbers

```
maxi ::  Int -> Int -> Int
maxi n m = if n>=m then n else m
```

- **Example:** Testing if an integer is `0`

```
isZero ::  Int -> Bool
isZero x = if (x == 0) then True else False
```

- **Conditionals:** A *conditional expression* has the form

```
if b then e1 else e2
```

where

- `b` is an expression of type `Bool`
- `e1` and `e2` are expressions with the <u>same</u> type

Guarded functions — An alternative to `if`*-statements*

- **Example:** `doubleMax` returns double the maximum of its inputs

```
doubleMax ::  Int -> Int -> Int
doubleMax x y
    | x >= y  =  2*x
    | x < y   =  2*y
```

- **Definition:** A guarded function is of the form

```
⟨function-name⟩ :: ⟨type 1⟩ ->⟨type n⟩ ->⟨output type⟩

⟨function-name⟩ ⟨var 1⟩...⟨var n⟩
        | ⟨guard 1⟩  =  ⟨expression 1⟩
        | ...        =  ...
        | ⟨guard m⟩  =  ⟨expression m⟩
```

where `⟨guard 1⟩, ..., ⟨guard m⟩ :: Bool`

The `Char` *type*

- **Elements of `Char` :** Letters, digits and special characters

- **Forming elements of `Char` :** Single quotes form characters:

  ```
  'd' ::  Char    '3' ::  Char
  ```

- **Functions:** Characters have codes and conversion functions

  ```
  chr ::  Int -> Char   ord ::  Char -> Int
  ```

- **Examples:** Try them out!

  ```
  offset ::  Int
  offset = ord 'A' - ord 'a'

  capitalize ::  Char -> Char
  capitalize ch = chr (ord ch + offset)

  isLower ::  Char -> Bool
  isLower x = ('a' <= x) && (x <= 'z')
  ```

The String type

- **Elements of `String`:** Lists of characters

- **Forming elements of `String`:** Double quotes form strings

  `‘‘Newcastle Utd’’   ‘‘1a’’`

- **Special Strings:** Newline and Tab characters

  `‘‘Super \n Alan’’  ‘‘1\t2\t3’’  putStr(‘‘Super \n Alan’’)`

- **Combining Strings:** Strings can be combined by `++`

  `‘‘Super ’’ ++ ‘‘Alan ’’ ++ ‘‘Shearer’’ = ‘‘Super Alan Shearer’’`

- **Example:** `duplicate` gives two copies of a string

The type of Fractions `Float`

- **Elements of `Float` :** Contains decimals, eg `-21.3`, `23.1e-2`

- **Built in Functions:** Arithmetic, Ordering, Trigonometric

- **Conversions:** Functions between `Int` and `String`

```
ceiling, floor, round  ::  Float -> Int
fromIntegral           ::  Int -> Float
show                   ::  Float -> String
read                   ::  String -> Float
```

- **Overloading:** Overloading is when values/functions belong to several types

```
2 ::  Int      show ::  Int -> String
2 ::  Float    show ::  Float -> String
```

Error-Handling

- **Motivation:** Informative error messages for run-time errors
- **Example:** Dividing by zero will cause a run-time error

```
myDiv ::  Float -> Float -> Float
myDiv x y = x/y
```

- **Solution:** Use an `error` message in a guarded definition

```
myDiv ::  Float -> Float -> Float
myDiv x y
    | y /= 0     = x/y
    | otherwise  = error ``Attempt to divide by 0''
```

- **Execution:** If we try to divide by `0` we get

```
Prelude> mydiv 5 0
Program error:  Attempt to divide by 0
```

- **Motivation:** Functions will often depend on other functions

- **Example :** Summing the squares of two numbers

```
sq ::  Int -> Int
sq x = x * x

sumSquares ::  Int -> Int -> Int
sumSquares x y = sq x + sq y
```

- **Problem:** Such definitions clutter the top-level environment

- **Answer:** Local definitions allow auxiliary functions

```
sumSquares2 ::  Int -> Int -> Int
sumSquares2 x y =  sq x + sq y
                   where sq z = z * z
```

- **Quadratic Equations:** The solutions of $ax^2 + bx + c = 0$ are

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- **Types:** Our program will have type

```
roots ::  Float -> Float -> Float -> String
```

- **Guards:** There are 3 cases to check so use a guarded definition

```
roots a b c
  | a == 0          =   ....
  | b*b-4*a*c == 0  =   ....
  | otherwise       =   ....
```

- **Code:** Now we can add in the answers

```
roots a b c
   | a == 0           = error ``Not a quadratic eqn''
   | b*b-4*a*c == 0   = ``One root:  ''  ++ show (-b/2*a)
   | otherwise        = ``Two roots:  ''  ++
                        show ((-b + sqrt (b*b-4*a*c))/2*a) ++
                         ``and'' ++
                        show ((-b - sqrt (b*b-4*a*c))/2*a)
```

- **Problem:** This program uses several expressions repeatedly
  - Being cluttered, the program is hard to read
  - Similarly the program is hard to understand
  - Repeated evaluation of the same expression is inefficient

- **Local decs:** Expressions used repeatedly are made local

```
roots a b c
   | a == 0                    = error ``Not a quadratic eqn''
   | disc == 0                 = ``One root:  ''  ++ show centre
   | otherwise                 = ``Two roots:  ''  ++
                                 show (centre + offset) ++
                                  ``and'' ++
                                 show (centre - offset)

     where
     disc = b*b-4*a*c
     offset = (sqrt disc) / 2*a
     centre = -b/2*a
```

Today You Should Have Learned

- **Types:** We have learned about Haskell's basic types. For each type we learned
  - Its basic values (elements)
  - Its built in functions
- **Expressions:** How to write expressions involving
  - Conditional expressions and Guarded functions
  - Error Handling and Local Declarations

Lecture 3 — New Types from Old

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 3

- **Building New Types:** Today we will learn about the following compound types
  - Pairs
  - Tuples
  - Type Synonyms
- **Describing Types:** As with basic types, for each type we want to know
  - What are the values of the type
  - What expressions can we write and how to evaluate them

From simple data values to complex data values

- **Motivation:** Data for programs modelled by values of a type
- **Problem:** Single values in basic types too simple for real data
- **Example:** A point on a plane can be specified by
  - A number for the x-coordinate and another for the y-coordinate
- **Example:** A person's complete name could be specified by
  - A string for the first name and another for the second name
- **Example:** The performance of a football team could be
  - A string for the team and a number for the points

New Types from Old I — Pair Types and Expressions

- **Examples:** For instance
  - The expression `(5,3)` has type `(Int, Int)`
  - The name `(‘‘Alan’’,‘‘Shearer’’)` has type `(String, String)`
  - The performance `(‘‘Newcastle’’, 22)` has type `(String,Int)`
- **Question:** What are the values of a pair type?
- **Answer:** A pair type contains pairs of values, ie
  - If `e1` has type `a` and `e2` has type `b`
  - Then `(e1,e2)` has type `(a,b)`

Functions using Pairs

- **Types:** Pair types can be used as input and/or output types

- **Examples:** The built in functions `fst` and `snd` are vital

```
fst ::  (a,b) -> a
fst (x,y) = x

winUpdate ::  (String,Int) -> (String,Int)
winUpdate (x,y) = (x,y+3)

movePoint ::  Int -> Int -> (Int,Int) -> (Int,Int)
movePoint m n (x,y) = (x+m,y+n)
```

- **Key Idea:** If input is a pair-type, use $(\langle \texttt{var1} \rangle, \langle \texttt{var2} \rangle)$ in definition

- **Key Idea:** If output is a pair-type, result is often $(\langle \texttt{exp1} \rangle, \langle \texttt{exp2} \rangle)$

New Types from Old II — Tuple Types and Expressions

- **Motivation:** Some data consists of more than two parts
- **Example:** Person on a mailing list
  - Specified by name, telephone number, and age
  - A person p on the list can have type `(String, Int, Int)`
- **Idea:** Generalise pairs of types to collections of types
- **Type Rule:** Given types `a1,...,an`, then `(a1,...,an)` is a type
- **Expression Formation:** Given expressions `e1::a1, ..., en::an`, then

  `(e1,...,en) :: (a1,...,an)`

Functions using Tuples

- **Example 1:** Write a function to test if a customer is an adult

```
isAdult ::  (String,Int,Int) -> Bool

isAdult (name, tel, age) = (age >= 18)
```

- **Example 2:** Write a function to update the telephone number

```
updateMove ::  (String,Int,Int) -> Int -> (String,Int,Int)
```

- **Example 3:** Write a function to update age after a birthday

```
updateAge ::  (String,Int,Int) -> (String,Int,Int)
```

General Definition of a Function: Patterns with Tuples

- **Definition:** Functions now have the form

```
<function-name> ::  <type 1> -> ...  -> <type n> -> <out-type>

<function-name> <pat 1> ...  <pat n> = <exp n>
```

- **Patterns:** Patterns are
  - Variables `x`: Use for any type
  - Constants `0`, `True`, `''cherry''`: Definition by cases
  - Tuples `(x,..,z)`: If the argument has a tuple-type
  - Wildcards `_`: If the output doesn't use the input
- **In general:** Use several lines and mix patterns.

More Examples

- **Example:** Using values and wildcards

```
isZero ::  Int -> Bool
isZero 0 = True
isZero _ = False
```

- **Example:** Using tuples and multiple arguments

```
expand ::  Int -> (Int,Int) -> (Int,Int,Int)
expand n (x,y) = (n, n*x, n*y)
```

- **Example:** Days in the month

```
days ::  String -> Int -> Int
 days ‘‘January’’   x   = 31
 days ‘‘February’’  x   = if isLeap x then 29 else 28
 days ‘‘March’’     x   = 31
  .....
```

New Types from Old III — Type Synonyms

- **Motivation:** More descriptive names for particular types.

- **Definition:** Type synonyms are declared with the keyword `type`.

```
type Team = String
type Goals = Int
type Match = ((Team,Goals), (Team,Goals))

numu ::  Match
numu = ((``Newcastle", 4), (``Manchester Utd'', 3))
```

- **Functions:** Types of functions are more descriptive, same code

```
homeTeam ::  Match -> Team
totalGoals ::  Match -> Goals
```

Today You Should Have Learned

- **Tuples:** Collections of data from other types
- **Pairs:** Pairs, triples etc are examples of tuples
- **Type synonyms:** Make programs easier to understand
- **Pattern Matching:** General description of functions covering definition by cases, tuples etc.
- **Pitfall!** What is the difference between

```
addPair ::  (Int,Int) -> Int
addPair (x,y) = x + y

addTwo ::  Int -> Int -> Int
addTwo x y = x + y
```

Lecture 4 — List Types

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 4 — List Types

- **Lists:** What are lists?
  - Forming list types
  - Forming elements of list types
- **Functions over lists:** Some old friends, some new friends
  - Functions from CO1003/4: `cons`, `append`, `head`, `tail`
  - Some new functions: `map`, `filter`
- **Clarity:** Unlike Java, Haskell treatment of lists is clear
  - No list iterators!

List Types and Expressions

- **Example 1:** `[3, 5, 14] :: [Int]` and `[3, 4+1, double 7] :: [Int]`
- **Example 3:** `['d','t','g'] :: [Char]`
- **Example 4:** `[['d'], ['d','t'], ['d','t','g']] :: [[Char]]`
- **Example 5:** `[double, square, cube] :: [Int -> Int]`
- **Empty List:** The empty list is `[]` and belongs to all list types
- **List Expressions:** Lists are written using square brackets `[...]`
  - If `e1,...,en` are expressions of type `a`
  - Then `[e1, ..., en]` is an expression of type `[a]`

- **Cons:** The cons function : adds an element to a list

```
:  ::  a -> [a] -> [a]

1       :  [2,3,4]     = [1,2,3,4]
addone  :  [square]    = [addone, square]
'a'     :  ['b', 'z']  = ['a', 'b', 'z']
```

- **Append:** Append joins two lists together

```
++ ::  [a] -> [a] -> [a]

[True, True] ++ [False]  = [True, True, False]
[1,2] ++ ([3] ++ [4,5])  = [1,2,3,4,5]
([1,2] ++ [3]) ++ [4,5]  = [1,2,3,4,5]
[] ++ [54.6, 67.5]       = [54.6, 67.5]
[6,5] ++ (4 :  [7,3])    = [6,5,4,7,3]
```

More Built In Functions

- **Head and Tail:** Head gives the first element of a list, tail the remainder

```
head [double, square] = double
head ([5,6]++[6,7])   = 5

tail [double, square] = [square]
tail ([5,6]++[6,7])   = [6,6,7]
```

- **Length and Sum:** The length of a list and the sum of a list of integers

```
length (tail [1,2,3]) = 2
sum [1+4,8,45] = 58
```

- **Sequences:** The list of integers from `1` to `n` is written

```
[1 .. n]
```

Two New Functions — Map And Filter

- **Map:** Map is a function which has two inputs.

  – The first input is a function eg `f`

  – The second is a list eg `[e1,e1,e3]`

  The output is the list obtained by applying the function to every element of the input list eg `[f e1, f e2, f e3]`

- **Filter:** Filter is a function which has two inputs.

  – The first is a *test*, ie a function returning a `Bool`.

  – The second is a list

  The output is the list of elements of the input list which the function maps to `True`, ie those elements which pass the test.

- **Even Numbers:** The even numbers less than or equal to `n`
  - `evens::Int->[Int]`

- **Solution 1** — Using `filter`.

```
evens2 ::  Int -> [Int]
evens2 n = filter isEven [1 ..  n]
           where isEven x = (x `mod` 2 == 0)
```

- **Solution 2** — Using `map`

Today You Should Have Learned

- **Types:** We have looked at list types
  - What list types and list expressions looks like
  - What built in functions are available
- **New Functions:**
  - Map: Apply a function to every member of a list
  - Filter: Delete those that don't satisfy a property or test
- **Algorithms:** Develop an algorithm by asking
  - Can I solve this problem by applying a function to every member of a list or by deleting certain elements.

Lecture 5 — List Comprehensions

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 5

- **Recall Map:** Map is a function which has two inputs.

```
map add2 [2, 5, 6] = [4, 7, 8]
```

- **Recall Filter:** Filter is a function which has two inputs.

```
filter isEven [2, 3, 4, 5, 6, 7] = [2, 4, 6]
```

- **List comprehension:** An alternative way of writing lists
  - Definition of list comprehension
  - Comparison with `map` and `filter`

List Comprehension — An alternative to `map` *and* `filter`

- **Example 1:** If `ex = [2,4,7]` then

  ```
  [ 2*e | e <- xs ] = [4,8,14]
  ```

- **Example 2:** If `isEven :: Int->Bool` tests for even-ness

  ```
  [ isEven e | e <- xs ] = [True,True,False]
  ```

- **In General:** (Simple) list comprehensions are of the form

  ```
  [ ⟨exp⟩ | ⟨variable⟩ <- ⟨list-exp⟩ ]
  ```

- **Evaluation:** The meaning of a list comprehension is
  - Take each element of `list-exp`, evaluate the expression `exp` for each element and return the results in a list.

Using List Comprehensions Instead of `map`

- **Example 1:** A function which doubles a list's elements

```
double ::  [Int] -> [Int]
```

- **Example 2:** A function which tags an integer with its evenness

```
isEvenList ::  [Int] -> [(Int,Bool)]
```

- **Example 3:** A function to add pairs of numbers

```
addpairs ::  [(Int,Int)] -> [Int]
```

- **In general:** `map f l = [f x | x <- l]`

Using List Comprehensions Instead of Filter

- **Intuition:** List Comprehension can also select elements from a list

- **Example:** We can select the even numbers in a list

```
[ e | e <- l, isEven e]
```

- **Example:** Selecting names beginning with `A`

```
names ::  [String] -> [String]
names l ::  [ e | e <- l , head e == 'A' ]
```

- **Example:** Combining selection and applying functions

```
doubleEven ::  [Int] -> [Int]
doubleEven l ::  [ 2*e | e <- l , isEven e ]
```

General Form of List Comprehension

- **In General:** These list comprehensions are of the form

```
[ ⟨exp⟩ | ⟨variable⟩ <- ⟨list-exp⟩ , ⟨test⟩ ]
```

- **Example:** Infact, we can use several tests — if `l = [2,5,8,10]`

```
[ 2*e | e <- l , isEven e , e>3 ] = [16,20]
```

- **Key Example:** Cartesian product of two lists is a list of all pairs, such that for each pair, the first component comes from the first list and the second component from the second list.

```
[ (x,y) | x<-[1,2,3], y<-['a','b','c'] ]
                           = [(1,'a'), (1,'b') ...  ]

league ::  [Team]
games = [ (t1,t2) | t1 <- league, t2 <- league, t1 /= t2]
```

- **Problem:** Given a list remove all duplicate entries

- **Algorithm:** Given a list,
  - Keep first element
  - Delete all occurrences of the first element
  - Repeat the process on the tail

- **Code:**

Today You Should Have Learned

- **List Types:** We have looked at list types
  - What list types and list expressions looks like
  - What built in functions are available
- **List comprehensions:** Like `filter` and `map`. They allow us to
  - Select elements of a list
  - Delete those that dont satisfy certain properties
  - Apply a function to each element of the remainder

Lecture 6 — Recursion over Natural Numbers

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 6

- **Recursion:** General features of recursion
  - What is a recursive function?
  - How do we write recursive functions?
  - How do we evaluate recursive functions?
- **Recursion over Natural Numbers:** Special features
  - How can we guarantee evaluation works?
  - Recursion using patterns.
  - Avoiding negative input.

What is recursion?

- **Example:** Adding up the first n squares

  $$\texttt{hssquares n} = 0^2 + 1^2 + \ldots + \texttt{(n-1)}^2 + \texttt{n}^2$$

- **Types:** First we give the type of summing-squares

  ```
  hssquares ::  Int -> Int
  ```

- **Definitions:** Our program is a function

  ```
  hssquares 0 = 0
  hssquares n = n*n + hssquares(n-1)
  ```

- **Key Idea:** `hssquares` is recursive as its definition contains `hssquares` in a right-hand side – the function name "recurs".

General Definitions

- **Definition:** A function is *recursive* if the name recurs in its definition.

- **Intuition:** You will have seen recursion in action before
  - Imperative procedures which call themselves
  - Divide-and-conquer algorithms

- **Why Recursion:** Recursive definitions tend to be
  - Shorter, more understandable and easier to prove correct
  - Compare with a non-recursive solution

```
nrssquares n = n * (n+0.5) * (n+1)/3
```

Examples of evaluation

- **Example 1:** Let's calculate `hssquares 4`

```
hssquares 4  ⇒  4*4 + hssquares 3
             ⇒  16 + (3*3 + hssquares 2)
                ...
             ⇒  16 + (9 + ..  (1 + hssquares 0))
             ⇒  16 + (9 + ...  (1 + 0))      ⇒ 30
```

- **Example 2:** Here is a non-terminating function

```
mydouble n   =  n + mydouble (n/2)
mydouble 4   ⇒  4 + mydouble 2
             ⇒  4 + 2 + mydouble 1
             ⇒  4 + 2 + 1 + mydouble 0.5 ⇒ ......
```

- **Question:** Will evaluation stop?

Problems with Recursion

- **Questions:** There are some outstanding problems

  1. Is `hssquares` defined for every number?
  2. Does an evaluation of a recursive function always terminate?
  3. What happens if `hssquares` is applied to a negative number?
  4. Are these recursive definitions sensible: `f n = f n`, `g n = g (n+1)`

- **Answers:** Here are the answers

  1. Yes: The variable pattern matches every input.
  2. Not always: See examples.
  3. Trouble: Evaluation doesn't terminate.
  4. No: Why not?

Primitive Recursion over Natural Numbers

- **Motivation:** Restrict definitions to get better behaviour

- **Idea:** Many functions defined by three cases

  – A non-recursive call selected by the pattern `0`

  – A recursive call selected by the pattern `n+1` (*matches numbers* ≥ 1)

  – The error case deals with negative input

- **Example** Our program now looks like

```
hssquares2 0      =  0
hssquares2 (n+1)  =  (n+1)*(n+1) + hssquares n
hssquares2 x      =  error ``Negative input''
```

Examples of recursive functions

- **Example 1:** `star` uses recursion over `Int` to return a string

```
star          :: Int -> String
star 0        =  []
star (n+1)    =  '*' :  star n
star n        =  error ``Negative input''
```

- **Example 2:** `power` is recursive in its second argument

```
power           ::  Float -> Int -> Float
power x 0       =   1
power x (n+1)   =   x * power x n
power x n       =   error ``Negative input''
```

- **In General:** Use the following style of definition

```
⟨function-name⟩ 0      =  ⟨exp 1⟩
⟨function-name⟩ (n+1)  =  ⟨exp 2⟩
⟨function-name⟩ x      =  error⟨message⟩
```

  where

⟨`exp 1`⟩	does not contain	⟨`function-name`⟩
⟨`exp 2`⟩	may contain	⟨`function-name`⟩ applied to `n`

- **Evaluation:** Termination guaranteed!
  - If the input evaluates to `0`, evaluate ⟨`exp 1`⟩
  - If not, if the input is greater than `0`, evaluate ⟨`exp 2`⟩
  - If neither, return the error message

Larger Example

- **Problem:** Produce a table for `perf :: Int -> (String, Int)` where `perf 1 = ("Arsenal",4)` etc.

- **Stage 1:** We need some headings and then the actual table

```
printTable :: Int -> IO()
```

```
printTable numberTeams = putStr(header ++ rows numberTeams)
                         where
                         header = "Team\t Points\n"
```

- **Stage 2:** Convert each "row" to a string, recursively.

```
rows         :: Int -> String
rows 0       =  .....
rows (n+1)   =  .....
rows _       =  .....
```

The Function `rows`

- **Base Case:** If we want no entries, then just return `[]`

```
rows 0 = []
```

- **Recursive Case:** Convert $(n+1)$-rows by
  - recursively converting the first `n`-rows, and
  - adding on the `(n+1)`-th row
- **Code:** Code for the recursive call

The Final Version

```
perf :: Int -> (String,Int)
perf 1 = ("Arsenal",4)
perf 2 = ("Notts",5)
perf 3 = ("Chelsea",7)
perf n = error "perf out of range"

rows :: Int -> String
rows 0     = []
rows (n+1) = rows n ++
             fst(perf(n+1)) ++ "\t\t " ++
             show(snd(perf(n+1))) ++ "\n"
rows _     = error"rows out of range"

printTable :: Int -> IO()
printTable numberTeams = putStr(header ++ rows numberTeams)
                         where
                         header = "Team\t\t Points\n"
```

Today You Should Have Learned

- **Recursion:** Allows new functions to be written.
  - Advantages: Clarity, brevity, tractability
  - Disadvantages: Evaluation may not stop
- **Primitive Recursion:** Avoids bad behaviour of some recursive functions
  - The value at 0 is non-recursive
  - Each recursive call uses a smaller input
  - An error-clause catches negative inputs
- **Algorithm:** Ask yourself, what needs to be done to the recursive call to get the answer.

Lecture 7 — Recursion over Lists

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 7

- **Lists:** Another look at lists
  - Lists are a recursive structure
  - Every list can be formed by `[]` and `:`
- **List Recursion:** Primitive recursion for Lists
  - How do we write primitive recursive functions
  - Examples — `++`, `length`, `head`, `tail`, `take`, `drop`, `zip`
- **Avoiding Recursion?:** List comprehensions revisited

Recursion over lists

- **Question:** This lecture is about the following question
  - We know what a recursive function over `Int` is
  - What is a recursive function over lists?

- **Answer:** In general, the answer is the same as before
  - A recursive function mentions itself in its definition
  - Evaluating the function may reintroduce the function
  - Hopefully this will stop at the answer

Another Look at Lists

- **Recall:** The two basic operations concerning lists
  - The empty list `[]`
  - The cons operator `(:)  ::  a -> [a] -> [a]`
- **Key Idea:** Every list is either empty, or of the form `x:xs`

  `[2,3,7] = 2:3:7:[]  [True, False] = True:False:[]`
- **Recursion:** Define recursive functions using the scheme
  - Non-recursive call: Define the function on the empty list `[]`
  - Recursive call: Define the function on `(x:xs)` by using the function only on `xs`

- **Example 1:** Doubling every element of an integer list

```
double ::  [Int] -> [Int]
double [] = []
double (x:xs) = (2*x) :  double xs
```

- **Example 2:** Selecting the even members of a list

```
onlyEvens ::  [Int] -> [Int]
 onlyEvens []       =  []
 onlyEvens (x:xs)   =  if isEven x then x:rest else rest
                       where rest = onlyEvens xs
```

- **Example 3:** Flattening some lists

```
flatten ::  [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x ++ flatten xs
```

- **Definition:** Primitive Recursive List Functions are given by

```
⟨function-name⟩ []     = ⟨expression 1⟩
⟨function-name⟩ (x:xs) = ⟨expression 2⟩
```

where

⟨expression 1⟩	does not contain	⟨function-name⟩
⟨expression 2⟩	may contain expressions	⟨function-name⟩ xs

- **Compare:** Very similar to recursion over `Int`

```
⟨function-name⟩ 0     = ⟨expression 1⟩
⟨function-name⟩ (n+1) = ⟨expression 2⟩
```

where

⟨expression 1⟩	does not contain	⟨function-name⟩
⟨expression 2⟩	may contain expressions	⟨function-name⟩ n

More Examples:

- **Example 4:** Append is defined recursively

```
append :: [a] -> [a] -> [a]
```

- **Example 5:** Testing if an integer is an element of a list

```
member :: Int -> [Int] -> Bool
```

- **Example 6:** Reversing a list

```
reverse :: [a] -> [a]
```

What can we do with a list?

- **Mapping:** Applying a function to every member of the list

```
double [2,3,72,1]  =  [2*2, 2*3, 2*72, 2*1]
isEven [2,3,72,1]  =  [True, False, True, False]
```

- **Filtering:** Selecting particular elements

```
onlyEvens [2,3,72,1] = [2,72]
```

- **Taking Lists Apart:** `head`, `tail`, `take`, `drop`

- **Combining Lists:** `zip`

- **Folding:** Combining the elements of the list

```
sumList [2,3,7,2,1]  =  2 + 3 + 7 + 2 + 1
```

- **Recall:** List comprehensions look like

  [⟨`exp`⟩ | ⟨`variable`⟩ `<-` ⟨`list-exp`⟩ , ⟨`test`⟩]

- **Intuition:** Roughly speaking this means
  - Take each element of the list ⟨`list-exp`⟩
  - Check they satisfy ⟨`test`⟩
  - Form a list by applying ⟨`exp`⟩ to those that do

- **Idea:** Equivalent to filtering and then mapping. As these are recursive, so are list comprehensions although the recursion is hidden

Today You Should Have Learned

- **List Recursion:** Lists are recursive data structures
  - Hence, functions over lists tend to be recursive
  - But, as before, general recursion is badly behaved
- **Primitive List Recursion:** Similar to natural numbers
  - A non-recursive call using the pattern `[]`
  - A recursive call using the pattern `(x:xs)`
- **List comprehension:** An alternative way of doing some recursion

Lecture 8 — More Complex Recursion

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 8

- **Problem:** Our restrictions on recursive functions are too severe
- **Solution:** New definitional formats which keep termination
  - Using new patterns
  - Generalising the recursion scheme
- **Examples:** Applications to integers and lists
- **Sorting Algorithms:** What is a sorting algorithm?
  - Insertion Sort, Quicksort and Mergesort

More general forms of primitive recursion

- **Recall:** Our primitive recursive functions follow the scheme
  - **Base Case:** Define the function non-recursively at `0`
  - **Inductive Case:** Define the function at `(n+1)` in terms of the function at `n`

```
⟨function-name⟩ 0      =  ⟨exp 1⟩
⟨function-name⟩ (n+1)  =  ⟨exp 2⟩
⟨function-name⟩ x      =  error⟨message⟩
```

where

`⟨expression 1⟩`	does not contain	`⟨function-name⟩`
`⟨expression 2⟩`	may contain	`⟨function-name⟩` applied to `n`

- **Motivation:** But some functions do not fit this scheme, requiring more complex patterns

Fibonacci Numbers – More Complex Patterns

- **Example:** The first Fibonacci numbers are `0,1`. For each subsequent Fibonacci number, add the previous two together

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

- **Problem:** The following does not terminate on input 1

```
fib 0 = 0
fib (n+1) = fib n + fib (n-1)
```

- **Solution:** The new *pattern* `(n+2)` matches inputs ≥ 2

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

- **In General:** There are patterns `(n+1)`, `(n+2)`, `(n+3)`

More general recursion on lists

- **Recall:** Our primitive recursive functions follow the pattern
  - **Base Case:** Defines the function at `[]` non-recursively
  - **Inductive Case:** Defines the function at `(x:xs)` in terms of the function at `xs`

```
⟨function-name⟩ []      = ⟨exp 1⟩
⟨function-name⟩ (x:xs)  = ⟨exp 2⟩
```

where

`⟨expression 1⟩`	does not contain	`⟨function-name⟩`
`⟨expression 2⟩`	may contain	`⟨function-name⟩` applied to `xs`

- **Motivation:** As with integers, some functions don't fit this shape

More General Patterns for Lists

- **Recall:** With integers, we used more general patterns.

- **Idea:** Use `(x:(y:xs))` pattern to access first two elements

- **Example:** We want a function to delete every second element

```
delete [2,3,5,7,9,5,7] = [2,5,9,7]
```

- **Solution:** Here is the code

```
delete ::  [a] -> [a]
delete [] = []
delete [x] = [x]
delete (x:(y:xs)) = x :  delete xs
```

- **Example:** To delete every third element use pattern `(x:(y:(z:xs)))`

Examples of Recursion and patterns — See how the typing helps

- **Example 1:** Summing pairs in a list of pairs

```
sumPairs ::  [(Int,Int)] -> Int
```

- **Example 2:** Unzipping lists `unZip ::  [(a,b)] -> ([a],[b])`

Sorting Algorithms 1: Insertsort

- **Problem:** A sorting algorithm rearranges a list in order

```
sort [2,7,13,5,0,4] = [0,2,4,5,7,13]
```

- **Recursion:** Such algorithms usually recursively sort a smaller list

- **Insertsort Alg:** To sort a list, sort the tail recursively, and then insert the head

- **Code:**

```
inssort ::  [Int] -> [Int]
inssort [] = []
inssort (x:xs) = insert x (inssort xs)
```

where `insert` puts the number `x` in the correct place

- **Patterns:** Insert takes two arguments, number and list
  - The recursion for `insert` doesn't depend on the number
  - The recursion for `insert` does depend on whether the list is empty or not — use the `[]` and `(x:xs)` patterns

- **Code:** Here is the final code

```
insert ::  Int -> [Int] -> [Int]
insert n [] = [n]
insert n (x:xs)
    | n <= x     = n:x:xs
    | otherwise  = x:(insert n xs)
```

Sorting Algorithms 2: Quicksort

- **Quicksort Alg:** Given a list `l` and a number `n` in the list

  sort `l` = sort those elements less than `n` ++
  number of occurrences of `n` ++
  sort those elements greater than `n`

- **Code:** The algorithm may be coded

```
qsort ::  [Int] -> [Int]
qsort [] = []
 qsort (x:xs) =  qsort (less x xs) ++
                 occs x (x:xs) ++
                 qsort (more x xs)
```

  where `less`, `occs`, `more` are auxiliary functions

Defining the Auxiliary Functions

- **Problem:** The auxiliary functions can be specified
  - `less` takes a number and a list and returns those elements of the list less than the number
  - `occs` takes a number and a list and returns the occurrences of the number in the list
  - `more` takes a number and a list and returns those elements of the list more than the number
- **Code:** Using list comprehensions gives short code

```
less, occs, more ::  Int -> [Int] -> [Int]
less n xs = [x | x <- xs, x < n]
occs n xs = [x | x <- xs, x == n]
more n xs = [x | x <- xs, x > n]
```

Sorting Algorithm 3: Mergesort

- **Mergesort Alg:** Split the list in half, recursively sort each half and merge the results

- **Code:** Overall function reflects the algorithm

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort ws)
           where (ys,ws) = (take l xs, drop l xs)
                 l = length xs `div` 2
```

where merge combines two sorted lists

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x<y then x :  merge xs (y:ys)
                             else y :  merge (x:xs) ys
```

- **Recursion Schemes:** We've generalised the recursion schemes to allow more functions to be written
  - More general patterns
  - Recursive calls to ANY smaller value
- **Examples:** Applied them to recursion over integers and lists
- **Sorting Algorithms:** We've put these ideas into practice by defining three sorting algorithms
  - Insertion Sort
  - QuickSort
  - MergeSort

Lecture 9 — Higher Order Functions

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 9

- **Motivation:** Why do we want higher order functions
- **Definition:** What is a higher order function
- **Examples:**
  - Mapping: Applying a function to every member of a list
  - Filtering: Selecting elements of a list satisfying a property
- **Application:** Higher order sorting algorithms

- **Example 1:** A function to double the elements of a list

```
doubleList ::  [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = (2*x) :  doubleList xs
```

- **Example 2:** A function to square the elements of a list

```
squareList ::  [Int] -> [Int]
squareList [] = []
squareList (x:xs) = (x*x) :  squareList xs
```

- **Example 3:** A function to increment the elements of a list

```
incList ::  [Int] -> [Int]
incList [] = []
incList (x:xs) = (x+1) :  incList xs
```

The Common Pattern

- **Problem:** Three separate definitions despite a clear pattern

- **Intuition:** Examples apply a function to each member of a list

```
function ::  Int -> Int

functionList ::  [Int] -> [Int]
functionList [] = []
functionList (x:xs) = (function x) :  functionList xs
```

  where in our previous examples `function` is

```
double   square   inc
```

- **Key Idea:** Make auxiliary function `function` an input

A Higher Order Function — `map`

- **The Idea Coded:**

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

- **Advantages:** There are several advantages

  – Shortens code as previous examples are given by

```
doubleList xs  =  map double xs
squareList xs  =  map square xs
   incList xs  =  map inc xs
```

  – Captures the algorithmic content and is easier to understand

  – Easier code-modification and code re-use

A Definition of Higher Order Functions

- **Question:** What is the type of `map`?
  - First argument is a function
  - Second argument is a list whose elements have the same type and the input of the function.
  - Result is a list whose elements are the output type of the function.
- **Answer:** So overall type is `map :: (a -> b) -> [a] -> [b]`
- **Definition:** A function is higher-order if an input is a function.
- **Another Example:** Type of `filter` is

```
filterInt :: (a -> Bool) -> [a] -> [a]
```

Quicksort Revisited

- **Idea:** Recall our implementation of *quicksort*

```
qsort ::  Ord a => [a] -> [a]
qsort [] = []
 qsort (x:xs)  =  qsort less ++ occs ++ qsort more
                  where
                  less = [e | e<-xs, e<x]
                  occs = x :  [e | e<-xs, e==x]
                  more = [e | e<-xs, e>x]
```

- **Polymorphism:** Quicksort requires an order on the elements:
  - The output list depends upon the order on the elements
  - This requirement is reflected in type class information `Ord a`
  - Don't worry about type classes as they are beyond this course

Limitations of Quicksort

- **Example:** Games tables might have type `[(Team,Points)]`
- **Problem:** How can we order the table?

```
Arsenal  16
AVilla   16
Derby    10
Birm.     4
...
```

- **Solution:** Write a new function for this problem

```
tSort [] = []
tSort (x:xs) = tSort less ++ [x] ++ tSort more
               where more = [e| e<-xs, snd e > snd x]
                     less = [e| e<-xs, snd e < snd x]
```

- What did we assume here?

Higher Order Sorting

- **Motivation:** But what if we want other orders, eg
  - Sort teams in order of names, not points
  - Sort on points, but if two teams have the same points, compare names

- **Key Idea:** Make the comparison a parameter of quicksort

```
qsortCp ::  (a -> a -> Bool) -> [a] -> [a]
qsortCp ord [] = []
qsortCp ord (x:xs) = qsortCp ord less ++ occs ++ qsortCp ord more
                 where less = [ e | e <- xs, ord e x]
                       occs = x :  [ e | e <- xs, e == x]
                       more = [ e | e <- xs, ord x e]
```

Examples

- **Key Idea:** To use a higher order sorting algorithm, use the required order to define the function to *sort by*

- **Example 1:** To sort by names

```
ord (t, p) (t', p') = t < t'
```

- **Example 2:** To sort by points and then names

```
ord (t, p) (t', p') = (p < p') || (p == p' && t < t')
```

- What should we assume about `ord`?

Today You Should Have Learned

- **Higher Order Functions:** Functions which takes functions as input
  - Facilitates code reuse and more abstract code
  - Many list functions are either `map`, `filter` or `fold`
- **HO Sorting:** An application of higher order functions to sorting
  - Produces more powerful sorting
  - Order of resulting list determined by a function
  - Lexicographic order allows us to try one order and then another

Lecture 10 — (Parametric) Polymorphism

Roy Crole

Department of Computer Science
University of Leicester

October 6, 2005

Overview of Lecture 10

- **Motivation:** Some examples leading to polymorphism

- **Definition:** What is *parametric* polymorphism?
  - What is a polymorphic type?
  - What is a polymorphic function?
  - Polymorphism and higher order functions
  - Applying polymorphic functions to polymorphic expressions

Monomorphic `length`

- **Example:** Let us define the length of a list of integers

```
mylength ::  [Int] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

- **Problem:** We want to evaluate the length of a list of characters

```
Prelude> mylength ['a', 'g']
ERROR: Type error in application
*** expression :  mylength ['a','g']
*** term :  ['a','g']
*** type :  [Char]
*** does not match :  [Int]
```

- **Solution:** Define a new length function for lists of characters . . . but this is not very efficient!

- **Better Solution:** The algorithm's input depends on the list type, but not on the type of integers.

- **Idea:** An alternative approach to typing `mylength`
  - There is one input and one output: `mylength :: a -> b`
  - The output is an integer: `mylength :: a -> Int`
  - The input is a list: `mylength :: [c] -> Int`
  - There is nothing more to infer from the code of `mylength` so

```
mylength :: [c] -> Int
```

  This is an efficient function - works at all list types!

- **Types**: Now we will deal with the following types:
  - Basic, built in types: `Int`, `Char`, `Bool`, `String`, `Float`
  - Type variables representing any type: `a`, `b`, `c`, `...`
  - Types built with type construc tors: `[]`, `->`, `(,)`

    `[Int]  a->a  a->b  a->Bool  (String,a->a)  [a->Bool]`

  - Type synonyms: `type <type-name> = <type-expression>`

```
type Point = (Int,Int)
type Line = (Point,Point)
type Test = a->Bool
```

Some Definitions

- **Polymorphism** is the ability to appear in different forms
- **Definition:** A type is *parametric polymorphic* iff it contains type variables (that is, type parameters).
- **Definition:** A function is *parametric polymorphic* iff it can be called on different types of input, and it is implemented by (code for) a single algorithm
- **Definition:** A function is *overloaded* iff it can be called on different types of input, and for each type of input, the function is implemented by (code for) a particular algorithm.
- **Examples:** Of overloading are the arithmetic operators: integer and floating-point addition.

Polymorphic Expressions

- **Key Idea:** Expressions have many types
  - Amongst these is a *principle* type
- **Example:** What is the type of `id x = x`
  - `id` sends an integer to an integer. So `id ::  Int -> Int`
  - `id` sends a list of type `a` to a list of type `a`. So `id::[a]->[a]`
  - `id` sends an expression of type b to an expression of type b. So `id::b->b`
- **Principle Type:** The last type includes the previous two – why?
  - In fact the principal type of `id` is `id::b->b` – why?

Examples

- **Example 1:** What is the type of `map`

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

- **Example 2:** What is the type of `filter`

```
filter f [] = []
filter f (x:xs) = if f x then x:filter f xs else filter f xs
```

- **Example 3:** What is the type of `iterate`

```
iterate f 0 x = x
iterate f (n+1) x = f (iterate f n x)
```

Applying Polymorphic Expressions to Polymorphic Functions

- **Previously:** The typing of applications of expressions:
  - If `exp1` is an expression with type `a -> b`
  - And `exp2` is an expression with type `a`
  - Then `exp1 exp2` has type `b`
- **Problem:** How does this apply to polymorphic functions?

```
length          ::  [c] -> Int
[2,4,5]         ::  [Int]
length [2,4,5]  ::  Int
```

- **Key Idea:** Argument type can be an *instance* of input type

When is a Type an Instance *of Another Type*

- **Recall:** Two facts about expressions containing variables
  - Variables stand for arbitrary elements of a particular type
  - *Instances* of the expression are obtained by substituting expressions for variables
- **Key Idea:** (Parametric) polymorphic types are defined in the same way:
  - Type-expressions may contain type-variables
  - *Instances* of type-expressions are obtained by substituting types for type-variables
- **Example:** `[Int]` is an instance of `[c]` – substitute `Int` for `c`

- **Monomorphic:** Can a function be applied to an argument?
  - If the function's input type is the same type as its argument

$$\frac{\texttt{f::a->b x::a}}{\texttt{f x :: b}}$$

- **Polymorphically:** Can a function be applied to an argument?
  - If the function's input type is *unifiable* with argument's type

$$\frac{\texttt{f::a->b}\quad\texttt{x::c}\quad \theta \text{ unifies } \texttt{a,c}}{\texttt{f x :}\theta\texttt{b}}$$

  where θ maps type variables to types

- **Example:** In the `length` example, set θ`c=Int`

Example

- **Past Paper:** Assume `f` is a function with principle type

  ```
  f::([a],[b])->Int->[(b,a)]
  ```

  Do the following expressions type check? State **Yes** or **No** with a brief reason and, if **Yes**, what is the principal type of the expression?

  1. `f (3,3) 2`
  2. `f ([],[]) 5`
  3. `f ([tail,head], []) 3`
  4. `f ([True,False], ['x'])`

Today You Should Have Learned

- **Polymorphism:**
  - Saves on code — one function (algorithm) has many types
  - This implements our algorithmic intuition

- **Type Checking:** Expressions and functions have many types including a principle one
  - Polymorphic functions are applied to expressions whose type is an instance of the type of the input of the function