Room 306 (Faraday Tower)
Phone 513380
Fax 295708
u.berger@swansea.ac.uk
http://www-compsci.swan.ac.uk/~csulrich/
The aims of this course

At the end of this course you should be able to

1. **understand the concepts of functional programming** including their historical and theoretical background and their relations to other programming styles;

2. **solve computational problems** by well-structured functional programs (we will use the language Haskell);

3. **prove the correctness of functional programs** by standard proof-techniques.
Recommended Books


• Kees Doets and Jan van Eijck. The Haskell Road to Logic, Maths and Programming, King’s College Publications, 2004.


Lab classes

- Monday 3-4 pm, Tuesday 1-2 pm, Thursday 9-10 am, room 217 (Linux lab)
- Start: Monday, 9th of October
- Get password via email
- Course web page
  http://www-compsci.swan.ac.uk/~csulrich/fp1.html
  Contains downloading information, web links to Haskell documentation, hints on how to use Haskell under Linux and Windows, lecture slides, coursework, e.t.c.
Coursework

- Coursework counts 20% for CS-221 and 30% for CS-M36 (part 1).
- CS-221 and CSM36 (part 1): 4 courseworks (5% each).
- One extra coursework for CS-M36 (part 1) (10%).
- Submission via email. See backside of coursework sheet or course web page for detailed instructions.
Overview (Part I)

1. Functional programming: Ideas, results, history, future
2. Types and functions
3. Case analysis, local definitions, recursion
4. Higher order functions and polymorphism
5. The $\lambda$-Calculus
6. Lists
7. User defined data types
8. Proofs
1 Functional Programming: Ideas, Results, History, Future
1.1 Ideas

• Programs as functions \( f : \text{Input} \rightarrow \text{Output} \)
  
  ○ No variables — no states — no side effects
  
  ○ All dependencies explicit

  ○ Output depends on inputs only, not on environment

Referential Transparancy
• Abstraction
  ○ Data abstraction
  ○ Function abstraction
  ○ Modularisation and Decomposition

• Specification and verification
  ○ Typing
  ○ Clear denotational and operational semantics
## Comparison with other programming paradigms

<table>
<thead>
<tr>
<th>Programming Paradigm</th>
<th>Concept 1</th>
<th>Concept 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative Programming</td>
<td>state</td>
<td>command</td>
</tr>
<tr>
<td>Logic Programming</td>
<td>specification</td>
<td>search</td>
</tr>
<tr>
<td>Functional Programming</td>
<td>expression</td>
<td>evaluation</td>
</tr>
</tbody>
</table>
# 1.2 Results

The main current functional languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Types</th>
<th>Evalution</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisp, Scheme</td>
<td>type free</td>
<td>eager</td>
<td>via side effects</td>
</tr>
<tr>
<td>ML, CAML</td>
<td>polymorphic</td>
<td>eager</td>
<td>via side effects</td>
</tr>
<tr>
<td>Haskell, Gofer</td>
<td>polymorphic</td>
<td>lazy</td>
<td>via monads</td>
</tr>
</tbody>
</table>
Haskell, the language we use in this course, is a purely functional language. Lisp and ML are not purely functional because they allow for programs with side effects.

Haskell is named after the American Mathematician Haskell B Curry (1900 – 1982).

Picture on next page copied from http://www-groups.dcs.st-and.ac.uk/~history
1.2 Results

MMISS:
Some areas where functional languages are applied

- Artificial Intelligence
- Scientific computation
- Theorem proving
- Program verification
- Safety critical systems
- Web programming
- Network toolkits and applications
1.2 Results

• XML parser

• Natural Language processing and speech recognition

• Data bases

• Telecommunication

• Graphic programming

• Games

http://homepages.inf.ed.ac.uk/wadler/realworld/
Productivity and security

- Functional programming is very often more productive and reliable than imperative programming.
- Ericsson measured an improvement factor of between 9 and 25 in experiments on telephony software.
- Because of their modularity, transparency and high level of abstraction, functional programs are particularly easy to maintain and adapt.
- See http://www.haskell.org/aboutHaskell.html
Example: Quicksort

To sort a list with head $x$ and tail $xs$, compute

- $low =$ the list of all elements in $xs$ that are smaller than $x$,
- $high =$ the list of all elements in $xs$ that are greater or equal than $x$.

Then, recursively sort $low$ and $high$ and append the results putting $x$ in the middle.
Quicksort in Haskell

\[ qsort \; [] = [] \]
\[ qsort \; (x:xs) = qsort \; low \; ++ \; [x] \; ++ \; qsort \; high \]

where

\[ low = [y \mid y \leftarrow xs, y < x] \]
\[ high = [y \mid y \leftarrow xs, y \geq x] \]
void qsort(int a[], int lo, int hi) {
{
    int h, l, p, t;

    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];
    }
}
do {
    while ((l < h) && (a[l] <= p))
        l = l+1;
    while ((h > l) && (a[h] >= p))
        h = h-1;
    if (l < h) {
        t = a[l];
        a[l] = a[h];
        a[h] = t;
    }
} while (l < h);
\begin{verbatim}
    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
\end{verbatim}
1.3 History

- **Foundations 1920/30**
  - Combinatory Logic and $\lambda$-calculus (Schönfinkel, Curry, Church)

- **First functional languages 1960**
  - LISP (McCarthy), ISWIM (Landin)

- **Further functional languages 1970–80**
  - FP (Backus); ML (Milner, Gordon), later SML and CAML; Hope (Burstall); Miranda (Turner)

- **1990: Haskell**
1.4 Future

- Functional programming more and more wide spread
- Functional and object oriented programming combined (Pizza, Generic Java)
- Extensions by dependent types (Chayenne)
- Big companies begin to adopt functional programming
- Microsoft initiative: F# = CAML into .net
2 Types and functions
Contents

• How to define a function
• How to run a function
• Some basic types and functions
• Pairs and pattern matching
• Infix operators
• Computation by reduction
2.1 How to define a function

• Example
  \[ \text{inc} :: \text{Int} \rightarrow \text{Int} \]
  \[ \text{inc} \ x = x + 1 \]

• Explanation
  ○ \text{inc} :: \text{Int} \rightarrow \text{Int} \text{ is the signature} declaring \text{inc} as a function expecting an integer as input and computing an integer as output.
  ○ \text{inc} \ x = x + 1 \text{ is the definition} saying that the function \text{inc} computes for any integer \( x \) the integer \( x + 1 \).
  ○ The symbol \( x \) is called a \text{formal parameter}, \text{Int} is the Haskell \text{type} of (small) integers.
• **Naming conventions:**

  ○ Functions and formal parameters begin with a *lower case* letter.

  ○ Types begin with an *upper case* letter.

• **The definition**

  \[
  \text{inc } x = x + 1
  \]

  should not be confused with the assignment

  \[
  x := x + 1
  \]

  in imperative languages.
• Example of a function expecting two arguments.

\[
d\text{Sum} :: \text{Int} \to \text{Int} \to \text{Int} \\
d\text{Sum} \ x \ y = 2 \times (x + y)
\]

• A combination of inc and dSum

\[
f :: \text{Int} \to \text{Int} \to \text{Int} \\
f \ x \ y = \text{inc} (d\text{Sum} \ x \ y)
\]
Why types?

- Early detection of errors at compile time
- Compiler can use type information to improve efficiency
- Type signatures facilitate program development
- and make programs more readable
- Types increase productivity and security
2.2 How to run a function

- **hugs** is a Haskell interpreter
  - Small, fast compilation (execution moderate)
  - Good environment for program development

- **How it works**
  - **hugs** reads definitions (programs, types, \ldots) from a file (script)
  - Command line mode: Evaluation of expressions
  - No definitions in command line
A hugs session

We assume that our example programs are written in a file `hugsdemo1.hs` (extension `.hs` required). After typing `hugs` in a command window in the same directory where our file is, we can run the following session (black = hugs, red = we)

MMISS: Running functions
Prelude> :l hugsdemo1.hs
Reading file "hugsdemo1.hs":
Hugs session for:
/usr/share/hugs/lib/Prelude.hs
hugsdemo1.hs
Main> dSum 2 3
10
Main> f 2 3
11
Main> f (f 2 3) 6
35
Main> :q
2.2 How to run a function

- At the beginning of the session hugs loads a file `Prelude.hs` which contains a bunch of definitions of Haskell types and functions. A copy of that file is available at our fp1 page. It can be quite useful as a reference.
- By typing `:?` (in hugs) one obtains a list of all available commands. Useful commands are:
  - `:load <filename>`
  - `:reload`
  - `:type <Haskell expression>`
  - `:quit`

  All commands may be abbreviated by their first letter.
-- That's how we write short comments

{-
Longer comments can be included like this
-}
Exercises

• Define a function `square` that computes the square of an integer (don’t forget the signature).

• Use `square` to define a function `p16` which raises an integer to its 16th power.

• Use `p16` to compute the 16th powers of some numbers between 1 and 10. What do you observe? Try to explain.
2.3 Some basic types and functions

We now discuss the basic Haskell types

- Boolean values
- Numeric types: Integers and Floating point numbers
- Characters and Strings
2.3 Some basic types and functions

**Boolean values:** `Bool`

- **Values** `True` and `False`

- **Predefined functions:**
  
  - `not :: Bool -> Bool`  
    
    negation
  
  - `(&&) :: Bool -> Bool -> Bool`  
    
    conjunction (infix)
  
  - `(||) :: Bool -> Bool -> Bool`  
    
    disjunction (infix)

```
True && False ⇝ False (⇝ means “evaluates to”)
```

```
True || False ⇝ True
```

```
True || True ⇝ True
```
• Example: exclusive disjunction:

```haskell
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && (not (x && y))
```

- `exOr True True`  \(\Rightarrow\)  False
- `exOr True False`  \(\Rightarrow\)  True
- `exOr False True`  \(\Rightarrow\)  True
- `exOr False False`  \(\Rightarrow\)  False
Basic numeric types

Computing with numbers

Limited precision ←→ arbitrary precision
constant cost increasing cost

Haskell offers:

- **Int** - integers as machine words
- **Integer** - arbitrarily large integers
- **Rational** - arbitrarily precise rational numbers
- **Float** - floating point numbers
- **Double** - double precision floating point numbers
Integers: Int and Integer

Some predefined functions (overloaded, also for Integer):

\(+\), \(*)\), (\(^\)\), (\(-\) :: Int \rightarrow Int \rightarrow Int

\(-\) :: Int \rightarrow Int \quad -- \text{unary minus}

abs :: Int \rightarrow Int \quad -- \text{absolute value}

div :: Int \rightarrow Int \rightarrow Int \quad -- \text{integer division}

mod :: Int \rightarrow Int \rightarrow Int \quad -- \text{remainder of int. div.}

show :: Int \rightarrow \text{String}
2.3 Some basic types and functions

\[
\begin{align*}
3 \ ^\ 4 & \rightsquigarrow 81 \\
4 \ ^\ 3 & \rightsquigarrow 64 \\
-9 \ + \ 4 & \rightsquigarrow -5 \\
-(9 \ + \ 4) & \rightsquigarrow -13 \\
2-(9 \ + \ 4) & \rightsquigarrow -11 \\
\text{abs} \ -3 & \rightsquigarrow \text{error} \\
\text{abs} \ (-3) & \rightsquigarrow 3 \\
\text{div} \ 9 \ 4 & \rightsquigarrow 2 \\
\text{mod} \ 9 \ 4 & \rightsquigarrow 1
\end{align*}
\]
Comparison operators:

\((==), (\neq), (\leq), (<), (\geq), (>)\) :: Int -> Int -> Bool

\(-9 == 4 \leadsto False\)

\(9 == 9 \leadsto True\)

\(4 \neq 9 \leadsto True\)

\(9 \geq 9 \leadsto True\)

\(9 > 9 \leadsto False\)
Floating point numbers: Float, Double

- Single and double precision Floating point numbers (IEEE 754 and 854)
- The arithmetic operations (+), (−), (×), (−) may also be used for Float and Double
- Float and Double support the same operations
2.3 Some basic types and functions

(/) :: Float -> Float -> Float
pi :: Float
exp, log, sqrt, logBase, sin, cos :: Float -> Float

3.4/2 \rightarrow 1.7
pi \rightarrow 3.14159265358979
exp 1 \rightarrow 2.71828182845905
log (exp 1) \rightarrow 1.0
logBase 2 1024 \rightarrow 10.0
cos pi \rightarrow -1.0
Conversion from and to integers:

\[
\text{fromIntegral} :: \text{Int} \to \text{Float} \\
\text{fromIntegral} :: \text{Integer} \to \text{Float} \\
\text{round} :: \text{Float} \to \text{Int} \quad -- \text{round to nearest int.} \\
\text{round} :: \text{Float} \to \text{Integer}
\]

Need signature to resolve overloading:

\[r = \text{round (fromIntegral 10)} :: \text{Int}\]
Example

half :: Int -> Float
half x = x / 2

Does not work because division (/) expects two floating point numbers as arguments, but x has type Int.

Solution:

half :: Int -> Float
half x = (fromIntegral x) / 2
Characters and strings: Char, String

Notation for characters: ’a’, . . .

Some predefined functions (one might have to load the module Char first: :l Char)

ord :: Char -> Int  -- ASCII code of a character
chr :: Int -> Char  -- inverse of ord

ord ’a’ ≈ 97
chr 97 ≈ ’a’
toLower :: Char -> Char       -- to lower case
toUpper :: Char -> Char       -- to upper case

isDigit :: Char -> Bool       -- 0,1,2,3,4,5,6,7,8,9
isAlpha :: Char -> Bool       -- is letter?

($) :: Char -> String -> String -- prefixing
(++) :: String -> String -> String -- concatenation

'H' : "ello W" ++ "orld!" ⟷ "Hello World!"
Example

rep :: String -> String
rep s = s ++ s

rep (rep "hello ") \implies "hello hello hello hello hello hello hello "
If \( a \) and \( b \) are types then \((a, b)\) denotes the cartesian product of \( a \) and \( b \) (in mathematics usually denoted \( a \times b \)).

The elements of \((a, b)\) are pairs \((x, y)\) where \( x \) is in \( a \) and \( y \) is in \( b \).
In the module *Prelude.hs* the projection functions are defined by pattern matching as follows:

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

snd :: (a,b) -> b
snd (_,y) = y
```

The underscore is a wild card or anonymous variable (like in Prolog).

The letters `a` and `b` are type variables, that is, place holders for arbitrary types (see section on polymorphism).
2.5 Infix operators

- **Operators**: Names from special symbols `!$%&/?+^` ...

- are written **infix**: `x && y`

- otherwise they are normal functions.
• Using other functions infix:

\[
\text{x 'ex0r' y} \\
\text{x 'dSum' y}
\]

Note the difference: ‘ex0r’ ‘a’

• Operators in prefix notation:

\[
\text{(&&)} \text{ x y} \\
\text{(+) 3 4}
\]
2.6 Computation by reduction

- Recall our examples.

\[
\begin{align*}
\text{inc} \; x &= x + 1 \\
\text{dSum} \; x \; y &= 2 \times (x + y) \\
f \; x \; y &= \text{inc} \; (\text{dSum} \; x \; y)
\end{align*}
\]

- Expressions are evaluated by reduction.

\[
\begin{align*}
\text{dSum} \; 6 \; 4 &\rightarrow 2 \times (6 + 4) \rightarrow 20
\end{align*}
\]
Evaluation strategy

• From outside to inside, from left to right.

\[
f (\text{inc } 3) 4 \\
\leadsto \text{inc (dSum (inc } 3) 4) \\
\leadsto (\text{dSum (inc } 3) 4) + 1 \\
\leadsto 2*(\text{inc } 3 + 4) + 1 \\
\leadsto 2*((3 + 1) + 4) + 1 \leadsto 17
\]

• call-by-need or lazy evaluation
  ○ Arguments are calculated only when they are needed.
  ○ Lazy evaluation is useful for computing with infinite data structures.
rep (rep "hello ")

\[ \rightarrow \text{rep "hello"} ++ \text{rep "hello"} \]

\[ \rightarrow ("hello " ++ "hello ") ++ ("hello " ++ "hello ") \]

\[ \rightarrow "hello hello hello hello hello hello" \]
Exercises

- Define a function `average` that computes the average of three integers.
• Consider the following definitions.

\[ f :: \text{Int} \to \text{Int} \]
\[ f \ x = f \ (x+1) \]

\[ g :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ g \ x \ y = y \]

\[ h :: \text{Int} \to \text{Int} \]
\[ h \ x = g \ (f \ x) \ x \]

What will be the result of evaluating \( h \ 0 \)?
3 Case analysis, local definitions, recursion
Content

• Forms of case analysis: if-then-else, guarded equations

• Local definitions: where, let

• Recursion

• Layout
3.1 Case analysis

- **If-then-else**
  
  $$\text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
  
  $$\text{max} \; x \; y = \text{if} \; x < y \; \text{then} \; y \; \text{else} \; x$$

- **Guarded equations**
  
  $$\text{signum} :: \text{Int} \rightarrow \text{Int}$$
  
  $$\text{signum} \; x$$
  
  $| \; x < 0 \; \quad = \; -1$
  
  $| \; x == 0 \; \quad = \; 0$
  
  $| \; \text{otherwise} \quad = \; 1$
3.2 Local definitions

- **let**

  \[
  g :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
g \ x \ y = (x^2 + y^2) / (x^2 + y^2 + 1)
  \]

  **better**

  \[
  g :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
g \ x \ y = \text{let } a = x^2 + y^2 \text{ in } a / (a + 1)
  \]
3.2 Local definitions

- **where**

```markdown
\[ g :: \text{Float} \to \text{Float} \to \text{Float} \]
\[ g \, x \, y = \frac{a}{a + 1} \text{ where } \]
\[ a = x^2 + y^2 \]

or

```markdown
\[ g :: \text{Float} \to \text{Float} \to \text{Float} \]
\[ g \, x \, y = \frac{a}{b} \text{ where } \]
\[ a = x^2 + y^2 \]
\[ b = a + 1 \]
```
The analogous definition with `let`:

\[
g :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}
g \ x \ y = \text{let } a = x^2 + y^2 \\
\quad \quad \quad \quad \quad \quad \quad b = a + 1 \\
\quad \quad \quad \quad \quad \quad \quad \text{in } a / b
\]
Local definition of functions

The sum of the areas of two circles with radii \( r, s \).

\[
\text{totalArea} :: \text{Float} \to \text{Float} \to \text{Float}
\]
\[
\text{totalArea} \ r \ s = \pi \cdot r^2 + \pi \cdot s^2
\]

Use auxiliary function to compute the area of one circle

\[
\text{totalArea} :: \text{Float} \to \text{Float} \to \text{Float}
\]
\[
\text{totalArea} \ r \ s = \begin{align*}
\text{let} & \quad \text{circleArea} \ x = \pi \cdot x^2 \\
\text{in} & \quad \text{circleArea} \ r + \text{circleArea} \ s
\end{align*}
\]
• Locally defined functions may also be written with a type signature:

\[
\text{totalArea} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}
\]
\[
\begin{align*}
\text{totalArea } r \text{ } s &= \\
&= \text{let } \text{circleArea} :: \text{Float} \rightarrow \text{Float} \\
&\quad \text{circleArea } x = \pi \times x^2 \\
&\quad \text{in } \text{circleArea } r + \text{circleArea } s
\end{align*}
\]

• Exercise: Use \texttt{where} instead.
3.3 Recursion

- **Recursion** = defining a function in terms of itself.

```haskell
fact :: Int -> Int
fact n = if n == 0 then 1 else n * fact (n - 1)
```

Does not terminate if `n` is negative. Therefore

```haskell
fact :: Int -> Int
fact n
  | n < 0    = error "negative argument to fact"
  | n == 0   = 1
  | n > 0    = n * fact (n - 1)
```
• The Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, ... 

```haskell
fib :: Integer -> Integer
fib n
  | n < 0           = error "negative argument"
  | n == 0 || n == 1 = 1
  | n > 0           = fib (n - 1) + fib (n - 2)
```

Due to two recursive calls this program has exponential run time.
• A linear Fibonacci program with a subroutine computing pairs of Fibonacci numbers:

```haskell
fib :: Integer -> Integer
fib n = fst (fibpair n) where
    fibpair n
    | n < 0     = error "negative argument to fib"
    | n == 0    = (1,1)
    | n > 0     = let (k,l) = fibpair (n - 1)
                 in (l,k+l)
```
3.4 Layout

Due to an elaborate layout Haskell programs do not need many brackets and are therefore well readable. The layout rules are rather intuitive:

- Definitions must start at the beginning of a line
- The body of a definition must be indented against the function defined,
- lists of equations and other special constructs must properly line up.