Functional Programming I

***

Functional Programming and Interactive Theorem Proving

Ulrich Berger
Michaelmas Term 2007
***

Room 306 (Faraday Tower)
Phone 513380
Fax 295708
u.berger@swansea.ac.uk
http://www.cs.swan.ac.uk/~csulrich/

***
The aims of this course

• To understand the main concepts of functional programming;

• To solve computational problems using a functional programming language (we will use Haskell);

• To prove the correctness of functional programs.
Recommended Books


Information on the web

• The course web page (contains more links to useful web pages)
  http://www.cs.swan.ac.uk/~csulrich/fp1.html

• The Haskell home page
  http://www.haskell.org

• Wikipedia
  http://en.wikipedia.org
Coursework

- Lab classes: Linux lab (room 2007), Monday 12-1, Tuesday 1-2. Start: 8/10. Computer accounts will be sent by email.
- Coursework counts 20% for CS-221 and 30% for CS-M36 (part 1).
- CS-221: Courseworks 1,2.
- CSM36 (Part 1): Courseworks 1,2,3.
- Submission via coursework box on the 2nd floor.
- Deadlines:
  CW1: 17/10 - 1/11,  CW2: 14/11 - 29/11,  CW3: 21/11 - 6/12.
No lecture on Thursday, 11th of October!
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 Transparency
• 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>State</th>
<th>Command</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>Evaluation</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 | y <- xs, y < x]
        high = [y | y <- xs, y >= 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))
            
            MMISS:
        } 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);

t = a[l];
a[l] = a[hi];
a[hi] = t;
qsort( a, lo, l-1 );
qsort( a, l+1, hi );
}
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 widespread
- 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
- Type synonyms
2.1 How to define a function

• Example

```haskell
inc :: Int -> Int
inc x = x + 1
```

• Explanation

  ○ 1. line: **Signature** declaring \( \texttt{inc} \) as a function expecting an integer as input and computing an integer as output.

  ○ 2. line: **Definition** saying that the function \( \texttt{inc} \) computes for any integer \( x \) the integer \( x + 1 \).

  ○ The symbol \( x \) is called a **formal parameter**,
Naming conventions (required)

- Example

```
inc :: Int -> Int
inc x = x + 1
```

- Functions and formal parameters begin with a lower case letter.

- Types begin with an upper case letter.
Definition vs. assignment

The **definition**

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

must not be confused with the **assignment**

\[ x := x + 1 \]

in imperative languages.
Functions with more than one argument

Example of a function expecting two arguments.

dSum :: Int -> Int -> Int
dSum x y = 2 * (x + y)

A combination of inc and dSum

f :: Int -> Int -> Int
f x y = inc (dSum 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, ...) from a file (**script**)
  - Command line mode: Evaluation of **expressions**
  - No definitions in command line

- **ghci** (Glasgow Haskell Compiler Interactive) works similarly.
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)
2.2 How to run a function

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.
2.2 How to run a function

-- 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

- Boolean values
- Numeric types: Integers and Floating point numbers
- Characters and Strings
Boolean values: \texttt{Bool}

- **Values** \texttt{True} and \texttt{False}

- **Predefined functions:**
  \begin{align*}
    \texttt{not} & : \texttt{Bool} \rightarrow \texttt{Bool} & \text{negation} \\
    (\&\&) & : \texttt{Bool} \rightarrow \texttt{Bool} \rightarrow \texttt{Bool} & \text{conjunction (infix)} \\
    (\|\|) & : \texttt{Bool} \rightarrow \texttt{Bool} \rightarrow \texttt{Bool} & \text{disjunction (infix)}
  \end{align*}

\[
\begin{align*}
\text{True} \&\& \text{False} & \leadsto \text{False} & (\leadsto \text{means “evaluates to”}) \\
\text{True} \|\| \text{False} & \leadsto \text{True} \\
\text{True} \|\| \text{True} & \leadsto \text{True}
\end{align*}
\]
• Example: exclusive disjunction:

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

- exOr True True → False
- exOr True False → True
- exOr False True → True
- exOr False False → 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 -> Int -> Int
- :: Int -> Int -- unary minus
abs :: Int -> Int -- absolute value
div :: Int -> Int -> Int -- integer division
mod :: Int -> Int -> Int -- remainder of int. div.

show :: Int -> String
2.3 Some basic types and functions

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

\[
(==), (/=), (\leq), (<), (\geq), (>), (>) :: \text{Int} \to \text{Int} \to \text{Bool}
\]

\[
-9 \; == \; 4 \; \mapsto \; \text{False}
\]

\[
9 \; == \; 9 \; \mapsto \; \text{True}
\]

\[
4 \; /= \; 9 \; \mapsto \; \text{True}
\]

\[
9 \; \geq \; 9 \; \mapsto \; \text{True}
\]

\[
9 \; > \; 9 \; \mapsto \; \text{False}
\]
But also

\(\text{true} = \text{false} \implies \text{false}\)

\(\text{true} = \text{true} \implies \text{true}\)

\(\text{true} \neq \text{false} \implies \text{true}\)

\(\text{true} \neq \text{true} \implies \text{false}\)

\(\text{true} > \text{false} \implies \text{true}\)

\(\text{false} > \text{true} \implies \text{false}\)
Exercise

Recall the definition

\[
\text{exOr} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{exOr} \ x \ y = (x || y) \&\& (\text{not} \ (x \&\& y))
\]

Give a shorter definition of \text{exOr} using a comparison operator.
Floating point numbers: Float, Double

- Single and double precision Floating point numbers (IEEE 754 and 854)

- The arithmetic operations (+), (-), (\times), (-) may also be used for Float and Double

- Float and Double support the same operations
2.3 Some basic types and functions

\[
(\div) :: \text{Float} \to \text{Float} \to \text{Float} \\
p_i :: \text{Float} \\
\text{exp, log, sqrt, logBase, sin, cos} :: \text{Float} \to \text{Float}
\]

\[
\begin{align*}
3.4/2 & \quad \mapsto \quad 1.7 \\
p_i & \quad \mapsto \quad 3.14159265358979 \\
\text{exp 1} & \quad \mapsto \quad 2.71828182845905 \\
\log (\text{exp 1}) & \quad \mapsto \quad 1.0 \\
\logBase 2 1024 & \quad \mapsto \quad 10.0 \\
\cos p_i & \quad \mapsto \quad -1.0
\end{align*}
\]
Conversion from and to integers:

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

Use signature to resolve overloading:

\[\text{round 10.7 :: Int}\]
Example

```haskell
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:

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

Notation for characters: ’a’
Notation for characters: ’’hello’’

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

’H’ : "ello W" ++ "orld!" ~→ "Hello World!"

: binds stronger than ++

’H’ : "ello W" ++ "orld!" is the same as
(’H’ : "ello W") ++ "orld!"
Example

\[ \text{rep} :: \text{String} \rightarrow \text{String} \]

\[ \text{rep } s = s ++ s \]

\[ \text{rep (rep "hello ") } \rightarrow \text{"hello hello hello hello hello "} \]
2.4 Pairs and pattern matching

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:

\[
\begin{align*}
\text{fst} &: (a, b) \rightarrow a \\
\text{fst} (x, _) &= x \\
\text{snd} &: (a, b) \rightarrow b \\
\text{snd} (_, y) &= y
\end{align*}
\]

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).
Booleans and numbers can be used for pattern matching as well:

```plaintext
myif :: Bool -> Int -> Int -> Int
myif True  n  m = n
myif False n  m = m

The order of arguments corresponds to the order of types.

myif' :: Bool -> (Int,Int) -> Int
myif' True  (n,m) = n
myif' False (n,m) = m

myif expects 3 arguments while myif' expects only 2 arguments.
```
Pattern matching with numbers and a wildcard:

```haskell
myfun :: Int -> String
myfun 0 = "Hello"
myfun 1 = "world"
myfun 2 = "!"
myfun _ = ""
```

The definitions are tried from top to bottom. Therefore, the wildcard _ applies if neither of the numbers 0, 1, 2, match.
2.5 Infix operators

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

- are written **infix**: \( x \ \&\& \ y \)

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

\[ x \ 'exOr' \ y \]
\[ x \ 'dSum' \ y \]

Note the difference: ‘exOr’ ‘a’

• Operators in prefix notation:

\[ (&&) \ x \ y \]
\[ (+) \ 3 \ 4 \]
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.

\[
\text{dSum} \; 6 \; 4 \implies 2 \times (6 + 4) \implies 20
\]
Evaluation strategy

- From **outside** to **inside**, from **left** to **right**.

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

- **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 rep "hello" ++ rep "hello"
\rightarrow ("hello " ++ "hello ") ++ ("hello " ++ "hello ")
\rightarrow "hello hello hello hello hello"
2.7 Type synonyms

Types can be given names to improve readability:

type Ct = Float  -- Coordinate

type Pt = (Ct,Ct)  -- Point in the plane

swap :: Pt -> Pt
swap (x,y) = (y,x)

Note that Haskell identifies the types Pt and (Float,Float).
2.8 Exercises

• Define a function `average` that computes the average of three integers.

• Define a function `distance` that computes the distance of two points in the plane.

• Define a variant of the distance function where the coordinates are integers.

• Define `ex0r` using pattern matching.
• Add the missing signatures to the following definitions:

\[
\text{cubroot } x = \sqrt{\sqrt{x}}
\]

\[
\text{energy } x \ y = x \ \ast \ y \ \ast \ 9.81
\]

\[
\text{three } x \ y \ z = x \ ++ \ y \ ++ \ z
\]
• Consider the following definitions.

\[
\begin{align*}
f &:: \text{Int} \rightarrow \text{Int} \\
f \ x &= f \ (x+1) \\
g &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
g \ x \ y &= y \\
h &:: \text{Int} \rightarrow \text{Int} \\
h \ x &= g \ (f \ x) \ x
\end{align*}
\]

What are the results of evaluating \( f \ 0 \) and \( h \ 0 \)?
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**

  \[
  \text{signum} :: \text{Int} \rightarrow \text{Int} \\
  \text{signum } x \\
  \quad | x < 0 \quad = -1 \\
  \quad | x == 0 \quad = 0 \\
  \quad | \text{otherwise} \quad = 1
  \]
3.2 Local definitions

• let

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

better

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

- **where**

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

or

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

\begin{verbatim}
g :: Float -> Float -> Float

\texttt{g x y = let a = x^2 + y^2
b = a + 1
in a / b}
\end{verbatim}
• **Local definition of functions**

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

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

Use auxiliary function to compute the area of one circle

\[
\text{totalArea} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{totalArea} \ r \ s = \\
\text{let} \ \text{circleArea} \ x = \pi \cdot x^2 \\
\text{in} \ \text{circleArea} \ r + \text{circleArea} \ s
\]
3.2 Local definitions

- Locally defined functions may also be written with a type signature:

```haskell
totalArea :: Float -> Float -> Float
totalArea r s =
  let circleArea :: Float -> Float
      circleArea x = pi * x^2
  in circleArea r + circleArea s
```

- Exercise: Use `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)
```
3.3 Recursion

- 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 :: Integer -> (Integer, Integer)
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.
3.5 Exercises

• Define recursively a function `euler` such that for every nonnegative integer `n` and every real number `x` (given as a floating point number)

\[ \text{euler } n \ x = \sum_{k=0}^{n} \frac{x^k}{k!} \]

For negative `n` the result shall be an error.

• Analyse the behaviour of `euler n x` when `n` increases and `x` is fixed (for example `x = 1`).
4 Higher order functions and Polymorphism
Contents

• Function types
• Higher order functions
• Polymorphic functions
• Type classes and overloading
• Lambda abstraction
4.1 Function types

Consider the function

\[ \text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ \text{add } x \ y = x + y \]

The signature of \text{add} is shorthand for

\[ \text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]
We may, for example, define

\[
\text{addFive} :: \text{Int} \to \text{Int}
\]
\[
\text{addFive} = \text{add } 5
\]
\[
\text{add} :: \text{Int} \to (\text{Int} \to \text{Int})
\]
\[
\text{addFive} = \text{add } 5 :: \text{Int} \to \text{Int}
\]
\[
\text{addFive } 7 :: \text{Int}
\]

Therefore

\[
\text{addFive } 7
\]
\[
\leadsto (\text{add } 5) \ 7 = \text{add } 5 \ 7
\]
\[
\leadsto 5 + 7
\]
\[
\leadsto 12
\]
• **Int -> Int** is a function type.

• Function types are ordinary types,

• they may occur as result types (like in **Int -> (Int -> Int)**),

• but also as argument types:

\[
twice :: (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int}
\]

\[
twice \ f \ x = f \ (f \ x)
\]

\[
twice \ \text{addFive} \ 7 \ \leadsto \ \text{addFive} \ (\text{addFive} \ 7) \ \leadsto \ \ldots \ 17
\]
4.1 Function types

- \( \text{type}_1 \rightarrow \text{type}_2 \rightarrow \ldots \rightarrow \text{type}_n \rightarrow \text{type} \)

  is shorthand for

  \( \text{type}_1 \rightarrow (\text{type}_2 \rightarrow \ldots \rightarrow (\text{type}_n \rightarrow \text{type})\ldots) \)

- \( f \ x_1 \ x_2 \ldots \ x_n \)

  is shorthand for

  \( (\ldots((f \ x_1) \ x_2) \ldots) \ x_n \)
Sections

If we apply a function of type, say
\[ f :: \text{type1} \to \text{type2} \to \text{type3} \to \text{type4} \to \text{type5} \]
to arguments, say, \( x_1 \) and \( x_2 \), then
\( x_1 \) must have type \( \text{type1} \), \( x_2 \) must have type \( \text{type2} \).

and we get
\[ f \ x_1 \ x_2 :: \text{type3} \to \text{type4} \to \text{type5} \]

The expression \( f \ x_1 \ x_2 \) is called a section of \( f \).

For example \( \text{add} \ 5 \) is a section of \( \text{add} \) because
\[ \text{add} :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ \text{add} \ 5 :: \text{Int} \to \text{Int} \]
• Applying a function $n$-times:

\[
\text{iter} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}
\]
\[
\text{iter } n \ f \ x
\]
\[
| \ n == 0 \quad = x \\
| \ n > 0 \quad = f \ (\text{iter} \ (n-1) \ f \ x) \\
| \ \text{otherwise} \quad = \text{error} \ "\text{negative argument to iter}" \\
\]

Informally, \( \text{iter } n \ f \ x = f \ (\ldots \ (f \ x)\ldots) \) with \( n \) applications of \( f \).
iter 3 addFive 7  ⇝ 
addFive (iter 2 addFive 7)  ⇝ 
(iter 2 addFive 7) + 5  ⇝ 
addFive (iter 1 addFive 7) + 5  ⇝ 
(iter 1 addFive 7) + 5 + 5  ⇝ 
addFive (iter 0 addFive 7) + 5 + 5  ⇝ 
(iter 0 addFive 7) + 5 + 5 + 5  ⇝ 
7 + 5 + 5 + 5  ⇝  22
• If a function has a function type as argument type, then it is called a **higher-order function**, 

• otherwise it is called a **first-order function**.

• `twice` and `iter` are higher-order functions whereas `add` and the functions we discussed in previous chapters were first-order functions.
4.2 Polymorphism

Another important difference between the functions

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

and

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

is that while in the signature of \text{add} the type \text{Int} was special (we could not have replaced it by, say, \text{Char}), in the signature of \text{twice} the type \text{Int} was irrelevant.
• In Haskell we can express the latter fact by assigning \texttt{twice} a polymorphic type, that is, a type that contains type variables:

\begin{align*}
\texttt{twice} :: & (a \to a) \to (a \to a) \\
\texttt{twice } f \ x &= f (f \ x)
\end{align*}

\texttt{twice} then becomes a polymorphic function.
• Type variable begin with a lower case letter.

• Polymorphic functions can be used in any context where the type variables can be substituted by suitable types such that the whole expression is well typed.
Exercise

Check the types of the following expressions and compute their values:

twice square 2

twice twice square 2
Type check:

\[
\begin{align*}
twice & :: (a \to a) \to (a \to a) \\
twice & :: (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \\
square & :: \text{Int} \to \text{Int} \\
2 & :: \text{Int} \\
\text{therefore} & \\
twice\ square\ 2 & :: \text{Int}
\end{align*}
\]
Computation:
(twice square) 2 $\rightsquigarrow$ square (square 2) $\rightsquigarrow$ (square 2)$^2$
$\rightsquigarrow$ $(2^2)^2$ $\rightsquigarrow$ 16
Type check:

\[
\begin{align*}
twice & :: (\ a \to \ a \ ) \to (\ a \to \ a \ ) \\
twice & :: ((\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})) \to ((\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})) \\
twice & :: (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \\
square & :: \text{Int} \to \text{Int} \\
2 & :: \text{Int}
\end{align*}
\]

therefore

\[
\begin{align*}
twice \ twice \ square \ 2 & :: \text{Int}
\end{align*}
\]
Computation:

\[
((\text{twice twice}) \text{ square}) \ 2 \\
\Rightarrow (\text{twice (twice square)}) \ 2 \\
\Rightarrow (\text{twice square}) ((\text{twice square}) \ 2) \\
\Rightarrow \text{square (square ((\text{twice square}) \ 2))} \\
\Rightarrow \text{(square ((\text{twice square}) \ 2))^2} \\
\Rightarrow (((\text{twice square}) \ 2)^2)^2 \\
\Rightarrow \ldots ((2^2)^2)^2 \\
\Rightarrow \ldots 2^{16} = 65536
\]
Some predefined polymorphic functions

- **Identity** (more useful than one might think)
  
  ```haskell
  id :: a -> a
  id x = x
  ```

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

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

- Composition

\[(.) :: (b \to c) \to (a \to b) \to (a \to c)\]
\[(f \cdot g) x = f (g x)\]

- Currying and Uncurrying

\[\text{curry} :: ((a,b) \to c) \to (a \to b \to c)\]
\[\text{curry } f \ x \ y = f (x,y)\]

\[\text{uncurry} :: (a \to b \to c) \to ((a,b) \to c)\]
\[\text{uncurry } f (x,y) = f x y\]
Exercises

In Haskell’s Prelude.hs the function uncurry is defined slightly differently:

\[
\text{uncurry} :: (a \to b \to c) \to ((a, b) \to c)
\]

\[
\text{uncurry } f \; p = f \; (\text{fst } p) \; (\text{snd } p)
\]

instead of

\[
\text{uncurry } f \; (x,y) = f \; x \; y
\]

Can you explain why?
Prove that the functions \texttt{curry} and \texttt{uncurry} are inverse to each other, that is

\[
\text{uncurry (curry } f) = f \\
\text{curry (uncurry } g) = g
\]

for all \( f :: (a,b) \rightarrow c \) and \( g :: a \rightarrow b \rightarrow c \).

We verify the equations for all possible arguments:

\[
\text{uncurry (curry } f) (x,y) = \text{curry } f \ x \ y = f (x,y) \\
\text{curry (uncurry } g) x \ y = \text{uncurry } g (x,y) = g x \ y
\]
Extensionality

Two functions are equal when they are equal at all arguments, that is,

\[ f = f' \text{ if and only if } f \ z = f' \ z \text{ for all } z. \]
• Until: Repeatedly applying an operation \( f \) to an initial value \( x \) until a property \( p \ x \) holds (predefined).

\[
\text{until} :: (a \to \text{Bool}) \to (a \to a) \to a \to a \\
\text{until } p \ f \ x = \\
\quad \text{if } p \ x \text{ then } x \text{ else until } p \ f \ (f \ x)
\]
Example: Given \( f :: \text{Int} \to \text{Int} \), find the least \( n \geq 0 \) such that \( f \ n \leq f \ (n+1) \).

\[
\text{nondecreasing} :: (\text{Int} \to \text{Int}) \to \text{Int} \\
\text{nondecreasing} \ f = \text{until \ nondec \ inc \ 0} \quad \text{where} \\
\quad \text{nondec} \ n = f \ n \leq f \ (n+1) \\
\quad \text{inc} \ n = n+1
\]

What is the value of \( \text{nondecreasing} \ f \) for

\[
f :: \text{Int} \to \text{Int} \\
f \ x = x^2 - 6x
\]
Exercise

Compute for any function \( f : \text{Int} \rightarrow \text{Int} \) and \( m \geq 0 \) the maximum of the values \( f \ 0, \ldots, f \ m \).

\[
\text{maxfun} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{maxfun} \ f \ m \\
| m < 0 \quad = \text{error "negative argument to maxfun"} \\
| m == 0 \quad = f \ 0 \\
| m > 0 \quad = \text{max} (\text{maxfun} \ f \ (m-1)) \ (f \ m)
\]

Here we used the predefined function \( \text{max} \).
4.3 Type Classes and Overloading

In Hugs one can find out the type of an expression by typing, for example:

Hugs> :t True
True :: Bool

Hugs> :t (&&)
(&&) :: Bool -> Bool -> Bool

(:t is shorthand for :type)
Also polymorphic functions may be queried for their types:

Hugs> :t fst
fst :: (a,b) -> a

Hugs> :t curry
curry :: ((a,b) -> c) -> a -> b -> c
Asking for the type of numeric expressions we get the following:

Hugs> :t 0
0 :: Num a => a

Hugs> :t (+)
(+) :: Num a => a -> a -> a

What does Num a => mean?
Type classes and bounded polymorphism

The fact that the type of (+) is

\[(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a\]

means that (+) is a function of two arguments that can be used with any type \(a\) which is member of the type class \textit{Num}.

Hence (+) is polymorphic, but its polymorphism is \textit{bounded} by the type class \textit{Num}.

All overloading in Haskell is implemented via type classes.
More about type classes

We can find out more about the type class `Num` by typing:

```
Hugs> :i Num
    -- type class
    infixl 6 +
    infixl 6 -
    infixl 7 *
    class (Eq a, Show a) => Num a where
        (+) :: a -> a -> a
        (-) :: a -> a -> a
        (*) :: a -> a -> a
        negate :: a -> a
```
4.3 Type Classes and Overloading

abs :: a -> a
signum :: a -> a
fromIntegral :: (Integral a, Num b) => a -> b

-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double
instance Integral a => Num (Ratio a)
The section

-- type class

... says which operations each member of the type class Num has.

The section

-- instances:

... says which types are member of the type class Num.
The line

```haskell
class (Eq a, Show a) => Num a where
```
says that each `Num` inherits all operations from the type classes `Eq` and `Show`.

The line

```haskell
fromIntegral :: (Integral a, Num b) => a -> b
```
says that for any type `a` in the type class `Integral` and any type `b` in the type class `Num` there is a function

```haskell
fromIntegral :: a -> b
```
Later in the course you will learn how to

- put a type into a type class;

- create your own type class.
Exercise

Find out more about the following type classes:

- Eq
- Show
- Ord
- Integral
4.4 λ-Abstraction

Using λ-abstraction we can create anonymous functions, that is, functions without name.

For example, instead of defining at top level the test function

\[ f :: \text{Int} \rightarrow \text{Int} \]
\[ f \ x = x^2 - 6 \times x \]

(which is not of general interest) and then running

Main> nondecreasing f

we may simply run

Main> nondecreasing (\x -> x^2 - 6*x)
In general an expression
\( \lambda x \rightarrow \ldots \)  

is equivalent to

\[
\text{let } f \ x = \ldots \text{ in } f
\]

\( \lambda \)-abstraction is not needed, strictly speaking, but often very handy.
Examples of $\lambda$-abstraction

- $\lambda$-abstraction in definitions:

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

  (instead of `square x = x*x`)

- Pattern matching in $\lambda$-abstraction:

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

  (instead of `swap (x,y) = (y,x)`)
Avoiding local definitions by \( \lambda \)-abstraction (not always recommended):

```haskell
nondecreasing :: (Int -> Int) -> Int
nondecreasing f = until (\n -> f n <= f (n+1))
                   (\n -> n+1)
                   0
```

instead of

```haskell
nondecreasing f = until nondec inc 0 where
    nondec n = f n <= f (n+1)
    inc n = n+1
```
Higher-order \( \lambda \)-abstraction (that is, the abstracted parameter has a higher-order type):

\[
\text{quad :: Int} \rightarrow \text{Int} \\
\text{quad = twice \ square}
\]

can be defined alternatively by

\[
\text{quad = (} \lambda f \rightarrow \lambda x \rightarrow f (f \ x) \text{)} \ (\lambda x \rightarrow x^*x)
\]
The type of a $\lambda$-abstraction

If we have an expression of type

$$<\ldots> :: \text{type2}$$

and the parameter $x$ has type $\text{type1}$, then the lambda abstraction has type

$$(\lambda x \rightarrow <\ldots>) :: \text{type1} \rightarrow \text{type2}$$
5 The $\lambda$-Calculus
Contents

• Historical background
• Syntax
• Operational Semantics
• Some fundamental questions
• Normalisation
• Uniqueness of normal forms
• Reduction strategies
• Universality of the λ-calculus
• Typed λ-calculi
5.1 Historical background

The $\lambda$-calculus is a formal system capturing the ideas of function abstraction and application. It was introduced by Alonzo Church and Haskell B Curry in the 1930s.

It has deep connections with logic and proofs and plays an important role in the development of Philosophy, Logic, and Computer Science.
In the following, the fundamentals of the $\lambda$-calculus will be outlined and the main results highlighted.

The study of the $\lambda$-calculus will give us an appreciation of the historical and philosophical context of functional programming and will help us in better understanding the power and elegance of functional concepts.
5.2 Syntax

\( \lambda \)-terms are defined by the grammar

\[
M ::= x \mid (\lambda x \, M) \mid (M \, N)
\]

Conventions:

- We omit outermost brackets.

- \( M \, N \, K \) means \( ((M \, N) \, K) \) e.t.c.

- \( \lambda x \, . \, M \, N \, K \) means \( \lambda x \, ((M \, N) \, K) \) e.t.c.
**Lambda-calculus vs. Haskell**

We may view $\lambda$-terms as Haskell expressions, if we read

\[
\lambda x \; M \quad \text{as} \quad \text{"}\; \text{x \rightarrow M}\quad \text{".}
\]

In fact, Haskell is an applied form of the $\lambda$-calculus.
Free variables

The set of free variables of a term $M$, denoted $\text{FV}(M)$, is the set of variables $x$ in a term that are not in the scope of a $\lambda$-abstraction, $\lambda x$.

Example: $\text{FV}((\lambda x . y x)(\lambda y . z y)) = \{y, z\}$

Note that in the example the variable $y$ has a free and a bound occurrence.
Substitution

Substitution: $M[N/x]$ is the result of replacing in $M$ every free occurrence of $x$ by $N$, renaming, if necessary, bound variables in $M$ in order to avoid “variable clashes”.

Instead of $M[N/x]$ some authors write $M[x/N]$, or $M[x := N]$.

Example:

$(((\lambda x . y x)(\lambda y . z y))[x z / y] = (\lambda x'.(x z)x')(\lambda y . z y)$

The following literal replacement would be **wrong**:

$(((\lambda x . y x)(\lambda y . z y))[x z / y] = (\lambda x.(x z)x)(\lambda y . z y)$
5.3 Operational Semantics

Conversions

- (α) \( \lambda x . M \rightarrow \lambda y . (M[y/x]) \) provided \( y \not\in \text{FV}(M) \).
- (β) \( (\lambda x . M) N \rightarrow M[N/x] \).
- (η) \( \lambda x . M x \rightarrow M \) provided \( x \not\in \text{FV}(M) \).

\( \alpha \)-conversion expresses bound renaming

\( \beta \)-conversion expresses computation

\( \eta \)-conversion expresses extensionality

We will concentrate on \( \beta \)-conversion.

We will identify terms that can be transformed into each other by \( \alpha \)-conversions (\( \alpha \)-variants).
Reduction

\(\beta\)-reduction means carrying out a \(\beta\)-conversion inside a term, that is

\[ C[(\lambda x \ M) \ N] \rightarrow C[M[N/x]] \]

where \(C[.]\) denotes a ‘term context’.

In the following ‘reduction’ means ‘\(\beta\)-reduction’ and ‘\(M \rightarrow N\)’ means that \(M\ \beta\)-reduces to \(N\).
Normal forms

A term is in $\beta$-normal form (or normal form, for short), if it cannot be reduced.

Examples: $\lambda x . x x$ is in normal form. $(\lambda x . x) x$ is not in normal form.

A term $N$ is the normal form of another term $M$ if $N$ is in normal form and there is a reduction chain from $M$ to $N$, that is,

$$M \rightarrow M_1 \rightarrow \ldots \rightarrow M_k \rightarrow N$$

for some terms $M_1, \ldots, M_k$. 
Example: Set $M := \lambda f \lambda x . f (f x)$ (= program twice).

\[
M \ M = (\lambda f \lambda x . f (f x))(\lambda f \lambda x . f (f x)) \\
\rightarrow \lambda x . (\lambda f \lambda x . f (f x))(\lambda f \lambda x . f (f x)) x \\
\rightarrow \lambda x . (\lambda f \lambda x . f (f x))(\lambda y . x (x y)) \\
\rightarrow \lambda x . \lambda z . (\lambda y . x (x y))((\lambda y . x (x y)) z) \\
\rightarrow \lambda x . \lambda z . (\lambda y . x (x y))(x (x z)) \\
\rightarrow \lambda x . \lambda z . x (x (x (x z))) \text{ (this is in normal form)}
\]

Hence $M \ M$ has normal form $\lambda x . \lambda z . x (x (x (x z)))$. 
5.4 Some fundamental questions

- Does every term have a normal form?
- Is the normal form unique?
- Which reduction strategies are good, which are bad?
- Which kinds of computations can be done with the \( \lambda \)-calculus?
5.5 Normalisation

Not every term has a normal form:

Set $\omega := \lambda x . x x$ and $\Omega := \omega \omega$. Then

$\Omega = (\lambda x . x x) (\lambda x . x x) \rightarrow (\lambda x . x x) (\lambda x . x x) \rightarrow \ldots$

Hence, $\Omega$ has no normal form.
• A term is **weakly normalising** if it has a normal form.

• A term is **strongly normalising** if every reduction sequence terminates (with a normal form).

• Exercise: Find terms that are
  
  ○ not weakly normalising;
  
  ○ weakly, but not strongly normalising;
  
  ○ strongly normalising.
5.6 Uniqueness of normal forms

- **Theorem (Church/Rosser)** $\beta$-reduction is **confluent** (or has the Church-Rosser property). This means: If $M \rightarrow^* M_1$ and $M \rightarrow^* M_2$, then there is a term $M_3$ such that $M_1 \rightarrow^* M_3$ and $M_2 \rightarrow^* M_3$.

- **Corollary**: Every term has at most one normal form.
5.7 Reduction strategies

Rightmost-innermost or eager reduction (Lisp, Scheme, ML)

- Advantages: Avoids repeated computations; computation cost easy to analyze.

- Disadvantages: May perform unnecessary computations; if divergence (non-termination) possible, then eager reduction does diverge.
Leftmost-outermost or lazy reduction (Haskell, Gofer)

- Advantages: No unnecessary computations; if termination possible, then lazy reduction terminates (safe strategy).

- Disadvantages: May perform repeated computations (if implemented naively); computation cost difficult to analyze.
Example

In a previous reduction chain we used the ‘rightmost-innermost’, or ‘eager’ reduction strategy.

Now let us reduce the same term $M M$ using ‘leftmost-outermost’, or ‘lazy’ reduction which is the strategy Haskell uses.
$$M M = (\lambda f \lambda x . f (f x))(\lambda f \lambda x . f (f x))$$

$$\rightarrow \lambda x . (\lambda f \lambda x . f (f x))(\lambda f \lambda x . f (f x)) x$$

$$\rightarrow \lambda x . \lambda y . ((\lambda f \lambda x . f (f x)) x)(((\lambda f \lambda x . f (f x)) x) y)$$

$$\rightarrow^2 \lambda x . \lambda y . (\lambda z . x (x z))((\lambda z . x (x z)) y)$$

$$\rightarrow \lambda x . \lambda y . x (x ((\lambda z . x (x z)) y))$$

$$\rightarrow \lambda x . \lambda y . x (x (x (x y)))$$
5.8 Universality of the \( \lambda \)-calculus

- **Theorem (Kleene)** The \( \lambda \)-calculus is a universal model of computation.

- This means:
  
  - Data can be encoded by \( \lambda \)-terms, for example \( n \in \mathbb{N} \) is encoded by the **Church Numeral**
    
    \[
    c_n := \lambda f. \lambda x. f(f \ldots (fx) \ldots) \quad (n \text{ times } 'f').
    \]
  
  - Given a computable function \( f : \mathbb{N} \to \mathbb{N} \), there is a \( \lambda \)-term \( M \) such that for all \( n \)
    
    \[
    M(c_n) \to^* c_f(n)
    \]
Key idea of the proof: Find a $\lambda$-term that performs recursion:

$$Y := \lambda f \cdot (\lambda x . f (x x)) (\lambda x . f (x x))$$

$Y$ is a fixed point combinator: For every term $M$

$$Y M =_{\beta} M (Y M)$$

where $N =_{\beta} K$ means that $N$ and $K$ can be reduced to a common term, that is $N \rightarrow^* L$ and $K \rightarrow^* L$ for some term $L$. 
Indeed if \( Y := \lambda f . (\lambda x \cdot (x \, x)) \, (\lambda x \cdot (x \, x)) \) then

\[
Y \, M \quad \rightarrow \quad (\lambda x \cdot M(x \, x)) \, (\lambda x \cdot M(x \, x)) \\
\rightarrow \quad M \, ((\lambda x \cdot M(x \, x)) \, (\lambda x \cdot M(x \, x)))
\]

But also

\[
M \, (Y \, M) \quad \rightarrow \quad M \, ((\lambda x \cdot M(x \, x)) \, (\lambda x \cdot M(x \, x)))
\]
Using the fixed point combinator $Y$ it is possible to encode any Turing machine.

The surprising computational power of the $\lambda$-calculus triggered the development of functional programming languages.

Modern implementations do not use $\beta$-reduction, literally, but are based on related, but more efficient models like, for example, Krivine’s Abstract Machine, SECD-machines, or Graph Reduction (see, for example, Davie’s book).
5.9 Typed λ-calculi

Typing is a way of assigning meaning to a term.

A typing judgement is of the form

\[ x_1 : a_1, \ldots, x_n : a_n \vdash M : b \]

It means (intuitively) if \( x_1 \) has type \( a_1 \) e.t.c., then \( M \) has type \( b \).

The sequence \( x_1 : a_1, \ldots, x_n : a_n \) is called a context.

It is common to use the letter \( \Gamma \) as a name for a context.
5.9 Typed $\lambda$-caluli

Basic typing rules

The three basic typing rules are:

\[
\begin{align*}
\Gamma, x : a & \vdash x : a \\
\Gamma, x : a & \vdash M : b \\
\Gamma & \vdash \lambda x \, M : a \to b \\
\Gamma & \vdash M : a \to b \quad \Gamma & \vdash N : a \\
\Gamma & \vdash MN : b
\end{align*}
\]

These rules define the simply typed $\lambda$-calculus.
For example

\[ M : a \rightarrow b \]

corresponds to Haskell’s

\[ M :: a \rightarrow b \]
Strong normalisation via typing

Theorem (Turing, Girard, Tait, Troelstra. . . ) Every typeable term is strongly normalizing.

This theorem extends to

- terms containing recursively defined functions which perform recursive calls at smaller arguments only.

- a variety of type systems, including large fragments of Haskell and ML.
Type inference

Haskell and ML use elaborate type inference systems (finding a context $\Gamma$ and type $a$ such that $\Gamma \vdash M : a$). Type inference was introduced by Roger Hindley.

Theorem (Hindley) It is decidable whether a term is typeable. If a term is typeable, its principal type (that is, most general type) can be computed.

Exercise: What is the most general type of $\lambda f. \lambda g. \lambda x. g(f x)$?
Books on the $\lambda$-calculus

- Comprehensive and very readable expositions of the Lambda-calculus can be found in the books

- The cited textbooks on Functional Programming by Bird, Davie and Thompson contain chapters about the $\lambda$-calculus and further pointers to the literature.
6 Lists
Contents

• Lists in Haskell

• Recursion on lists

• Higher-order functions on lists

• List comprehension

• Examples
6.1 Lists in Haskell

Haskell has a predefined data type of polymorphic lists, [a].

[] :: [a] -- the empty list

(,:) :: a -> [a] -> [a] -- adding an element

Hence the elements of [a] are either [], or of the form x:xs where x :: a and xs :: [a].
Lists: Haskell versus Prolog

Haskell and Prolog have the same notation, \([\ ]\), for the empty list

Haskell’s

\[x:xs\]

corresponds to Prolog’s

\([X|Xs]\).
Notations for lists

Haskell allows the usual syntax for lists

\[x_1, x_2, \ldots, x_n\] = 

\[x_1 : x_2 : \ldots : x_n : []\] = 

\[x_1 : (x_2 : \ldots (x_n : []) \ldots )\]
Some lists and their types

\[[1, 3, 5+4]\] :: [Int]

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

\[\begin{array}{c}
[ \begin{array}{c}
\text{True}, 3<2 \end{array}, [] \end{array}\] :: [[Bool]]
\end{array}
 [(198845,"Cox"),(203187,"Wu") ] :: [(Int,String)]

 [inc,\x \rightarrow x ^ 10] :: [Int \rightarrow Int]

 [iter 1, iter 2, iter 3] :: [(Int \rightarrow Int) \rightarrow Int \rightarrow Int]

 Recall

 inc :: Int \rightarrow Int
 inc x = x + 1

 iter :: Int \rightarrow (Int \rightarrow Int) \rightarrow Int \rightarrow Int
 iter n f x
 | n == 0      = x
 | n > 0       = f (iter (n-1) f x)
Generating lists conveniently

Haskell has a special syntax for lists of equidistant numbers:

\[1..5] \leadsto [1,2,3,4,5]\]

\[1,3..11] \leadsto [1,3,5,7,9,11]\]

\[10,9..5] \leadsto [10,9,8,7,6,5]\]

\[3.5,3.6..4] \leadsto [3.5,3.6,3.7,3.8,3.9,4.0]\]
We may also define functions using this notation:

- All integers between (and including) \( n \) and \( m \).

  ```haskell
type interval :: Int -> Int -> [Int]
interval n m = [n..m]
```

- The first \( n \) even numbers, starting with 0.

  ```haskell
type evens :: Int -> [Int]
evens n = [0,2..2*n]
```
Strings are lists of characters

From Prelude.hs:

\[ \text{type String} = [\text{Char}] \]

"Hello" == ['H', 'e', 'l', 'l', 'o'] ⇝ True
"
" == [] ⇝ True
'H' : "ello" ⇝ "Hello"
"H" == 'H' ⇝ Type error ...
6.2 Recursion on lists

General form: Define a function on lists by pattern matching and recursion:

\[ f :: [a] \rightarrow b \]
\[ f [] = \ldots \]
\[ f(x:xs) = \ldots x \ldots xs \ldots f xs \ldots \]

- Example: The sum of a list of integers

\[ \text{sumList} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sumList} [] = 0 \]
\[ \text{sumList} (x : xs) = x + \text{sumList} xs \]

\[ \text{sumList} [2,3,4] \leadsto 9 \]
• Squaring all members of a list of integers

\[
\text{squareAll} :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
\text{squareAll} \; [] = []
\]
\[
\text{squareAll} \; (x : xs) = x^2 : \text{squareAll} \; xs
\]

\[
\text{squareAll} \; [2,3,4] \quad \rightarrow \quad [4,9,16]
\]

Exercise: Make \text{squareAll} polymorphic using a suitable type constraint.
The sum of squares of a list of integers

```
sumSquares :: [Int] -> Int
sumSquares xs = sumList (squareAll xs)
```

Alternatively

```
sumSquares = sumList . squareAll
```

```
sumSquares [2,3,4] ~> 29
```
• Length of a list (predefined):

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```
6.2 Recursion on lists

- Concatenating (or appending) 2 lists

\[
(++) :: [a] \to [a] \to [a]
\]

\[
[] \quad ++ \quad ys \quad = \quad ys
\]

\[
(x:xs) \quad ++ \quad ys \quad = \quad x \quad : \quad (xs \quad ++ \quad ys)
\]
6.2 Recursion on lists

- Flattening nested lists (predefined):

\[
\text{concat} :: \text{[[a]]} \rightarrow \text{[a]}
\]
\[
\text{concat} \ [\] = [\]
\]
\[
\text{concat} \ (x:xs) = x ++ \ (\text{concat} \ xs)
\]

\[
\text{concat} \ [[1,2],[7],[8,9]] \rightarrow [1,2,7,8,9]
\]
\[
\text{concat} \ ["hello ", "world", "] \rightarrow "hello world!"
\]
6.2 Recursion on lists

- Replacing a character by another one

```haskell
replace :: Char -> Char -> String -> String
replace old new "" = ""
replace old new (x : s) = y : replace old new s
  where
    y = if x == old then new else x
```

```haskell
replace 'a' 'o' "Australia" ~> "Austrolio"
```

**Exercise:** Make replace polymorphic using a suitable type constraint.
head and tail

Head and tail of a nonempty list (predefined):

\[
\begin{align*}
\text{head} &:: [a] \rightarrow a \\
\text{head} (x:xs) &= x \\
\text{tail} &:: [a] \rightarrow [a] \\
\text{tail} (x:xs) &= xs
\end{align*}
\]

Undefined for the empty list.
take

Taking the first \( n \) elements of a list (predefined):

\[
\text{take} :: \text{Int} \to [a] \to [a]
\]

\[
\text{take } n \_ | n \leq 0 = []
\]

\[
\text{take } \_ \[\] = []
\]

\[
\text{take } n \ (x:xs) = x : \text{take } (n-1) \ xs
\]

\[
\text{take } 5 \ [1..100] \Rightarrow [1,2,3,4,5]
\]
6.2 Recursion on lists

reverse

The function reverse is predefined. Here is naive program for reversing lists:

\[
\begin{align*}
\text{reverse} & \quad : \quad [a] \to [a] \\
\text{reverse} \; [] & \quad = \quad [] \\
\text{reverse} \; (x:xs) & \quad = \quad \text{reverse} \; xs \; ++ \; [x]
\end{align*}
\]

The runtime of this program is \(O(n^2)\) because (++) is defined by recursion on its first argument.

We want a linear program.
A linear program for reverse

Idea for an improved program: compute

\[
\text{revConc } xs \ ys := \text{reverse } xs \ ++ \ ys
\]

\[
\text{revConc } [] \ ys = \text{reverse } [] \ ++ \ ys = [] \ ++ \ ys = ys
\]

\[
\text{revConc } (x:xs) \ ys = \text{reverse } (x:xs) \ ++ \ ys
\]
\[
= (\text{reverse } xs \ ++ \ [x]) \ ++ \ ys
\]
\[
= \text{reverse } xs \ ++ \ ([x] \ ++ \ ys)
\]
\[
= \text{reverse } xs \ ++ \ (x:ys)
\]
\[
= \text{revConc } xs \ (x:ys)
\]

The calculations above lead us immediately to a linear program:
Calculating a linear program for `reverse`

```haskell
reverse :: [a] -> [a]
reverse xs = revConc xs []
where

revConc :: [a] -> [a] -> [a]
revConc [] ys = ys
revConc (x:xs) ys = revConc xs (x:ys)
```
6.2 Recursion on lists

zip

Transforming two lists into one list of pairs.

Informally:

\[
\text{zip } [x_1, \ldots, x_n] [y_1, \ldots, y_n] = [(x_1, y_1), \ldots, (x_n, y_n)]
\]

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []

zip [1..5] "hat" \implies [(1, 'h'), (2, 'a'), (3, 't')]
Overview of predefined functions on lists

(++) :: [a] -> [a] -> [a]  concatenate 2 lists
(!!) :: [a] -> Int -> a   selecting $n$-th element
concat :: [[a]] -> [a]    flattening
null :: [a] -> Bool      testing if empty list
elem :: (Eq a) => a -> [a] -> Bool  testing if element in list
length :: [a] -> Int     length of a list
6.2 Recursion on lists

head, last :: [a] → a  first/last element

head, last :: [a] → a  first/last element

tail, init :: [a] → [a]  removing first/last element

replicate :: Int → a → [a]  \( n \) copies

take :: Int → [a] → [a]  take first \( n \) elements

drop :: Int → [a] → [a]  remove first \( n \) elts

splitAt :: Int → [a] → ([a], [a])  split at \( n \)-th pos

reverse :: [a] → [a]  reversing a list
zip :: [a] -> [b] -> [(a,b)]  \text{two lists into a list of pairs}\n
unzip :: [(a, b)] -> ([a], [b])  \text{inverse to zip}\n
and, or :: [Bool] -> Bool  \text{conjunction/disjunction}\n
sum :: Num a => [a] -> a  \text{sum}\n
product :: Num a => [a] -> a  \text{product}\n
Exercise for the lab classes on 5/11 and 6/11:
Define the functions above that haven’t yet been defined in the notes (use different names in order to avoid name conflicts with the Prelude library).
6.3 Higher-order functions on lists

- map: Applying a function to all its elements (predefined)

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f \ [\] \ = \ [] \\
\text{map } f \ (x:xs) \ = \ (f \ x) : (\text{map } f \ xs)
\]

Informally:

\[
\text{map } f \ [x_1, \ldots, x_n] \ = \ [f \ x_1, \ldots, f \ x_n]
\]
Example: Squaring all elements of a list of integers

\[
squareAll :: [\text{Int}] \rightarrow [\text{Int}]
\]
\[
squareAll \; \text{xs} = \text{map} \; (\wedge 2) \; \text{xs}
\]

or, even shorter,

\[
squareAll = \text{map} \; (\wedge 2)
\]
zipWith, a higher-order version of zip

Informally:

\[
\text{zipWith } f \ [x_1, \ldots, x_n] \ [y_1, \ldots, y_n] = [f \ x_1 \ y_1, \ldots, f \ x_n \ y_n]
\]

\[
\text{zipWith } :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]
\]

\[
\text{zipWith } f \ (x:xs) \ (y:ys) = f \ x \ y \ : \ \text{zip } xs \ ys
\]

\[
\text{zipWith } _ \ _ \ _ = []
\]

\[
\text{zipWith } (*) \ [2,3,4] \ [7,8,9] \ \mapsto \ [14,24,36]
\]
'Folding' lists (predefined)

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]
\[
\text{foldr } f \ e \ [] = e
\]
\[
\text{foldr } f \ e \ (x:xs) = f \ x \ (\text{foldr } f \ e \ xs)
\]

Informally:

\[
\text{foldr } f \ e \ [x_1, \ldots, x_n] = f \ x_1 (f \ x_2 (\ldots f \ x_n e))
\]

or, writing \( f \) infix:

\[
\text{foldr } f \ e \ [x_1, \ldots, x_n] = x_1 \ 'f' (x_2 \ 'f' (\ldots (x_n \ 'f' e)))
\]
• Defining \texttt{sum} using \texttt{foldr}

\[
\text{sum} :: \text{[Int]} \rightarrow \text{Int}\\
\text{sum} \; \text{xs} = \text{foldr} \; (+) \; 0 \; \text{xs}
\]

\[
\text{sum} \; [3,12] = 3 + \text{sum} \; [12] = 3 + 12 + \text{sum} \; [] = 3 + 12 + 0 = 15
\]
• Flattening lists using `foldr`

```haskell
concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

```haskell
concat [l1, l2, l3, l4] = l1 ++ l2 ++ l3 ++ l4 ++ []
```
Taking parts of lists

We had already `take`, `drop`.
Now we consider similar higher-order functions.

- Longest prefix for which property holds (predefined):

\[
\text{takeWhile} :: (a \to \text{Bool}) \to [a] \to [a]
\]
\[
\text{takeWhile } p \ [\] \ = \ [\]
\]
\[
\text{takeWhile } p \ (x:\text{x}:\text{xs})
\]
\[
\mid p \ x \ = \ x : \text{takeWhile } p \ \text{xs}
\]
\[
\mid \text{otherwise } = \ [\]
\]
6.3 Higher-order functions on lists

• Rest of this longest prefix (predefined):

\[
dropWhile :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
dropWhile p [] = []
dropWhile p x:xs
| p x = dropWhile p xs
| otherwise = x:xs
\]

• It holds: \(\text{takeWhile } p \ xs +\text{ } \text{dropWhile } p \ xs = xs\)

• Combination of both

\[
\text{span} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a],[a])
\text{span } p \ x = (\text{takeWhile } p \ xs, \text{dropWhile } p \ xs)
\]
6.3 Higher-order functions on lists

- Order preserving insertion

\[
\text{ins} :: \text{Int}\rightarrow \text{[Int]}\rightarrow \text{[Int]} \\
\text{ins} \ x \ \text{xs} = \text{lessx} ++ [x] ++ \text{grteqx} \ \\
\text{where} \\
(\text{lessx}, \text{grteqx}) = \text{span} (< x) \ \text{xs}
\]

- We use this to define insertion sort

\[
\text{isort} :: \text{[Int]} \rightarrow \text{[Int]} \\
\text{isort} \ \text{xs} = \text{foldr} \ \text{ins} \ [] \ \text{xs}
\]

Note that

\[
\text{isort} \ [2,3,1] = 2 \ \text{`ins`} (3 \ \text{`ins`} (1 \ \text{`ins`} []))
\]
Exercise

Make the functions \texttt{ins} and \texttt{isort} polymorphic.

\begin{verbatim}
ins :: Ord a => a -> [a] -> [a]
ins x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span (< x) xs

isort :: Ord a => [a] -> [a]
isort xs = foldr ins [] xs
\end{verbatim}
6.4 List comprehension

A schema for functions on lists:

- **Select** all elements of the given list
- which pass a given **test**
- and **transform** them in to a result.
Examples:

- Squaring all numbers in a list:
  
  \[ [\ x * x \ | \ x \ <- \ [1..10] \ ] \]

- Taking all numbers in a list that are divisible by 3:
  
  \[ [\ x \ | \ x \ <- \ [1..10], \ x \ 'mod' \ 3 == 0 \ ] \]

- Taking all numbers in a list that are divisible by 3 and squaring them:
  
  \[ [\ x * x \ | \ x \ <- \ [1..10], \ x \ 'mod' \ 3 == 0] \]
General form

\[ \{ E \mid c \leftarrow L, \text{test1}, \ldots, \text{testn} \} \]

With pattern matching:

\[
\text{addPairs} :: [(\text{Int},\text{Int})] \rightarrow [\text{Int}]
\]

\[
\text{addPairs} \ \text{ls} = [ x + y \mid (x, y) \leftarrow \text{ls} ]
\]

\[
\text{addPairs} \ (\text{zip} \ \ [1,2,3] \ [10,20,30]) \mapsto [11,22,33]
\]

Exercise: use \text{zipWith} instead of \text{addPairs} and \text{zip}. 

MMISS: List comprehension
Several generators

\[ \{ \text{E} \mid c_1 \leftarrow L_1, \ c_2 \leftarrow L_2, \ldots, \ \text{test}_1, \ldots, \ \text{test}_n \} \]

All pairings between elements of two lists (cartesian product):

\[ \{(x,y) \mid x \leftarrow [1,2,3], \ y \leftarrow [7,9]\} \implies \{(1,7),(1,9),(2,7),(2,9),(3,7),(3,9)\} \]

\[ \{(x,y) \mid y \leftarrow [7,9], \ x \leftarrow [1,2,3]\} \implies \{(1,7),(2,7),(3,7),(1,9),(2,9),(3,9)\} \]
Example: Quicksort

- Split a list into elements less-or-equal and greater than the first element,
- sort the parts,
- concatenate the results.

```haskell
qsort :: Ord a => [a] -> [a]
qusort [] = []
qusort (x:xs) = qsort [ y | y <- xs, y <= x ] ++ [x] ++ qsort [ y | y <- xs, y > x ]
```
### Filtering elements

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

or

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

What are the values of the following expressions?

\[ x \mid x \leftarrow [1..10], x > 5 \]

\[ \text{reverse } s \mid s \leftarrow ["Hello", "world", "!" ] \]

Expressing the same using \texttt{filter} and \texttt{map}:

\texttt{filter (> 5) [1..10]}

\texttt{map reverse ["Hello", "world", "!"]}
Exercises

- Determine the values of the following expressions (we use the predefined function \texttt{even :: Int -> Bool}).
  
  \begin{align*}
  \text{map even \ [1..10]} \\
  \text{filter even \ [1..10]} \\
  (\text{sum . filter even}) \ [1..10]
  \end{align*}

- Define a function that selects from a list of integers all elements that are even and smaller than 10.

- Define a function that takes a list \([x_1, x_2, \ldots, x_n]\) and returns \([(x_1, x_2), (x_2, x_3), \ldots (x_{n-1}, x_n)]\).
6.5 Examples

More examples on programming with lists:

- Shopping baskets
- Modelling a library
- Sorting with ordering as input
- Prime numbers: The sieve of Eratosthenes
- Forward composition
- Computing an Index
Shopping baskets

- An item consists of a name and a price.
  A basket is a list of items.

```haskell
type Item   = (String, Int)
type Basket = [Item]
```
• The total price of the content of a basket.

```
total :: Basket -> Int
total [] = 0
total ((name, price):rest) = price + total rest
```

or

```
total basket = sum [price | (_,price) <- basket]
```
Modelling a library

\[
\text{type Person} \quad = \text{String} \\
\text{type Book} \quad = \text{String} \\
\text{type DBase} \quad = [(\text{Person}, \text{Book})]
\]
• Borrowing and returning books.

```haskell
makeLoan :: DBase -> Person -> Book -> DBase
makeLoan dBase pers bk = (pers,bk) : dBase

returnLoan :: DBase -> Person -> Book -> DBase
returnLoan dBase pers bk
    = [ pair | pair <- dBase, pair /= (pers,bk) ]
```
● All books a person has borrowed.

\[
\text{books :: DBase} \rightarrow \text{Person} \rightarrow \text{[Book]}
\]

\[
\text{books db pers}
= [ \text{bk} \mid (p, \text{bk}) \leftarrow \text{db}, p == \text{pers} ]
\]

● Exercises

○ Returning all books.

○ Finding all persons that have borrowed a book.

○ Testing whether a person has borrowed any books.
Sorting with ordering as input

qsortBy :: (a -> a -> Bool) -> [a] -> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
    qsortBy ord [y| y <- xs, ord y x] ++ [x] ++
    qsortBy ord [y| y <- xs, not (ord y x)]

Sorting the library database by borrowers.

sortDB :: DBase -> DBase
sortDB = qsortBy (\(p1,\_\) -> \(p2,\_\) -> p1 < p2)
Computing prime numbers

- Sieve of Eratosthenes
  - When a prime number $p$ is found, remove all multiples of it, that is, leave only those $n$ for which $n \mod p \neq 0$.

  ```haskell
  sieve []     = []
  sieve (p:xs) =
                 p : sieve [n | n <- xs, n \mod p \neq 0]
  ```

- All primes in the interval $[1..n]$

  ```haskell
  primes :: Integer -> [Integer]
  primes n = sieve [2..n]
  ```
Forward composition

- Recall the (predefined) composition operator:

  \[(.) ::= (b \to c) \to (a \to b) \to a \to c\]
  \[(f \circ g) x = f (g x)\]

- Sometimes it is better to swap the functions:

  \[(\circ \circ) ::= (a \to b) \to (b \to c) \to a \to c\]
  \[(f \circ \circ g) x = g (f x)\]
• Example: the length of a vector. $|x_1, \ldots, x_n| = \sqrt{x_1^2 + \ldots + x_n^2}$.

```haskell
len :: [Float] -> Float
len = (map (^2)) >.> sum >.> sqrt
```
Computing an index

- Problem: Given a text

John Anderson my jo, John,\nwhen we were first aquent,
\nyour locks were like the raven,\nyour bonnie brow as bent;\nbut now your brow is beld, John,\nyour locks are like the snow;\nblessings on your frosty pow,\nJohn Anderson, my jo. (Robert Burns 1789)

produce for every word the list of lines were it occurs:

```python
[(8,"1789"),([1,8],"Anderson"),([8],"Burns"),([1,1,5,8],"John"),
([8],"Robert"),([2],"aquent"),([5],"belt"),([4],"bent"),
([7],"blessings"),([4],"bonnie"),([4,5],"brow"),([2],"first"),
([7],"frosty"),([3,6],"like"),([3,6],"locks"),([3],"raven"),
([6],"snow"),([2,3],"were"),([2],"when"),([3,4,5,6,7],"your")]
```
• Specification

```haskell
  type Doc  = String
  type Line = String
  type Wor   = String  -- Word predefined

  makeIndex :: Doc -> [[[Int], Wor]]
```
• Dividing the problem into small steps

(a) Split text into lines

(b) Tag each line with its number

(c) Split lines into words (keeping the line number)

(d) Sort by words:

(e) Collect same words with different numbers

(f) Remove words with less than 4 letters
Main program:

```haskell
makeIndex :: Doc -> [([Int],Wor)]
makeIndex = -- Doc
  lines >.> -- [Line]
numLines >.> -- [(Int,Line)]
numWords >.> -- [(Int,Wor)]
sortByWords >.> -- [(Int,Wor)]
amalgamate >.> -- [([Int],Wor)]
shorten -- [([Int],Wor)]
```
• Implementing the parts:

  ○ Split into lines: \texttt{lines} :: \texttt{Doc} \rightarrow \texttt{[Line]} predefined

  ○ Tag each line with its number:

    \texttt{numLines} :: \texttt{[Line]} \rightarrow \texttt{[(Int, Line)]}
    \texttt{numLines} \texttt{ls} = \texttt{zip} \texttt{[1.. length ls]} \texttt{ls}
Split lines into words:

```haskell
numWords :: [(Int, Line)] -> [(Int, Wor)]
```

Idea: Apply to each line: \( \text{words} :: \text{Line} \rightarrow \text{[Wor]} \) (predefined).

Problem:

- recognises spaces only
- therefore need to replace all punctuations by spaces
numWords :: [(Int, Line)] -> [(Int, Wor)]
numWords = concat . map oneLine where
  oneLine :: (Int, Line) -> [(Int, Wor)]
  oneLine (num, line) = map (\w -> (num, w)) (splitWords line)

\pagebreak

splitWords :: Line -> [Wor]
splitWords line =
  words [if c `elem` puncts then '' else c | c <- line ]
where puncts = ";:.,\"!()?{}-\[]"
Sort by words:

```haskell
sortByWords :: [(Int, Word)] -> [(Int, Word)]
s sortByWords = qsortBy ordWord where
    ordWord (n1, w1) (n2, w2) =
        w1 < w2 || (w1 == w2 && n1 <= n2)
```
Collect same words with different numbers

\[
\text{amalgamate} :: [(\text{Int}, \text{Wor})] \rightarrow [[[\text{Int}], \text{Wor}]]
\]

\[
\text{amalgamate} \ nws = \text{case} \ nws \ \text{of}
\]
\[
[] \quad \rightarrow \quad []
\]
\[
(_, w) : _ \quad \rightarrow \quad \text{let} \ ns = [ n | (n, v) \leftarrow nws, v == w ]
\]
\[
\text{other} = \text{filter} (\not (n, v) \rightarrow v /= w) \ nws
\]
\[
in (ns, w) : \text{amalgamate} \ \text{other}
\]
6.5 Examples

○ Remove words with less than 4 letters

\[
\text{shorten} :: \langle ([\text{Int}], \text{Wor}) \rangle \rightarrow \langle ([\text{Int}], \text{Wor}) \rangle \\
\text{shorten} = \text{filter} \ (\lambda (_-, \text{wd}) \rightarrow \text{length} \ \text{wd} \geq 4)
\]

**Alternative definition:**

\[
\text{shorten} = \text{filter} \ ((\geq 4) \ . \ \text{length} \ . \ \text{snd})
\]
7 Algebraic data types
Contents

• Simple data

• Composite data

• Recursive data
7.1 Simple data

A simple predefined algebraic data type are the booleans:

```haskell
data Bool = False | True
deriving (Eq, Show)
```

- The data type `Bool` contains exactly the constructors `False` and `True`.

- The expression `deriving (Eq, Show)` puts `Bool` into the type classes `Eq` and `Show` using default implementations of `==`, `/=` and `show`. 
Example: Days of the week

We may also define our own data types:

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  deriving(Eq, Ord, Enum, Show)
```

- **Ord**: Constructors are ordered from left to right

- **Enum**: Enumeration and numbering

  ```haskell
  fromEnum :: Enum a => a -> Int
  toEnum :: Enum a => Int -> a
  ```
Wed < Fri  \iff  True
Wed < Wed  \iff  False
Wed <= Wed  \iff  True
max Wed Sat  \iff  Sat
fromEnum Sun  \iff  0
fromEnum Mon  \iff  1
fromEnum Sat  \iff  6
toEnum 2  \iff  ERROR - Unresolved overloading ...
toEnum 2 :: Day  \iff  Tue
toEnum 7 :: Day  \iff  Program error ...
enumFromTo Mon Thu  \iff  [Mon,Tue,Wed,Thu]
[(Mon)..(Thu)]  \iff  [Mon,Tue,Wed,Thu]
Pattern matching

Example: Testing whether a day is a work day

- Using several equations

```
workDay :: Day -> Bool
workDay Sun = False
workDay Sat = False
workDay _   = True
```
• The same function using a new form of definition by cases:

```haskell
workDay :: Day -> Bool
workDay d = case d of
    Sun     -> False
    Sat     -> False
    _       -> True
```
• We can also use the derived ordering

```haskell
workDay :: Day -> Bool
workDay d = Mon <= d && d <= Fri
```

• or guarded equations

```haskell
workDay :: Day -> Bool
workDay d
  | d == Sat || d == Sun  = False
  | otherwise            = True
```
Example: Computing the next day

- Using enumeration

```haskell
 dayAfter :: Day -> Day
dayAfter d = toEnum ((fromEnum d + 1) 'mod' 7)
```

- The same with the composition operator

```haskell
 dayAfter :: Day -> Day
dayAfter = toEnum . ('mod' 7) . (+1) . fromEnum
```

Exercise: Define `dayAfter` without `mod` using case analysis instead.
Summary

- **data** is a keyword for introducing a new **data type** with **constructors**.

- Data types defined in this way are called **algebraic data types**.

- Names of data types and constructors begin with an **upper case** letter.

- **Definition by cases** can be done via **pattern matching** on constructors.
7.2 Composite data

We use representations of simple geometric shapes as an example for composite data.

```
type Radius = Float

type Pt = (Float,Float)

data Shape = Ellipse Pt Radius Radius
    | Polygon [Pt]
        deriving (Eq, Show)
```
• **Ellipse** \( p \ r_1 \ r_2 \) represents an ellipse with center \( p \) and radii \( r_1, r_2 \) (in \( x-, \) resp. \( y\)-direction).

• **Polygon** \( p_s \) represents a polygon with list of corners \( p_s \) (clockwise listing).

• For example, **Polygon** \[(-1,-1),(-1,1),(1,1),(1,-1)\] represents a centred square of side length 2.
The type of a constructor

- In a data type declaration
  
  ```haskell
data DataType = Constr Type1 ... Typen | ... 
  the type of the constructor is
  Constr :: Type1 -> ... -> Typen -> DataType
  ```

- In our examples:
  ```haskell
  True, False :: Bool
  Sun,...,Sat :: Day
  Elipse :: Pt -> Radius -> Radius -> Shape
  Polygon :: [Pt] -> Shape
  ```
Creating circles

A circle with center $p$ and radius $r$:

circle :: Pt -> Radius -> Shape
circle $p$ $r$ = Ellipse $p$ $r$ $r$
Creating rectangles

A rectangle is given by its bottom left and top right corner (we list the corners clockwise).

```haskell
rectangle :: Pt -> Pt -> Shape
rectangle (x,y) (x’,y’) =
  Polygon [(x,y),(x,y’),(x’,y’),(x’,y)]
```
Rotating points

Our next goal is to create a regular polygon with \( n \) corners and ‘radius’ \( r \). For this, we need a program that rotates a point around the origin (anticlockwise).

\[
\text{type Angle} = \text{Float}
\]

\[
\text{rotate} :: \text{Angle} -> \text{Pt} -> \text{Pt}
\]

\[
\text{rotate \ alpha \ (x,y)} = (a*x-b*y,b*x+a*y) \text{ where}
\]

\[
\begin{align*}
a &= \cos \alpha \\
b &= \sin \alpha
\end{align*}
\]
Creating a regular polygon

Creating a centered regular polygon with \( n \) corners and one corner at point \( p \).

\[
\text{regPol} :: \text{Int} \rightarrow \text{Pt} \rightarrow \text{Shape} \\
\text{regPol} ~ n ~ p = \\
\text{Polygon} \ [\text{rotate} \ (\text{fromIntegral} \ k \ * \ \theta) \ p \ |
\ k \ <- \ [0..(n-1)]] \\
\text{where} \\
\theta = - (2 \ * \ \pi \ / \ \text{fromIntegral} \ n)
\]
Shifting shapes

Shifting a shape:

\[
\text{addPt} :: \text{Pt} \rightarrow \text{Pt} \rightarrow \text{Pt} \\
\text{addPt} (x_1,y_1) (x_2,y_2) = (x_1+x_2,y_1+y_2)
\]

\[
\text{shiftShape} :: \text{Pt} \rightarrow \text{Shape} \rightarrow \text{Shape} \\
\text{shiftShape} p \text{ sh} = \text{case sh of} \\
\quad \text{Ellipse} q r1 r2 \rightarrow \text{Ellipse} (\text{addPt} p q) r1 r2 \\
\quad \text{Polygon} ps \rightarrow \text{Polygon} (\text{map} (\text{addPt} p) ps)
\]
Scaling shapes

scalePt :: Float -> Pt -> Pt
scalePt z (x,y) = (z * x, z * y)

scaleShape :: Float -> Shape -> Shape
scaleShape z sh = case sh of
  Ellipse p r1 r2 ->
    Ellipse (scalePt z p) (z*r1) (z*r2)
  Polygon ps -> Polygon (map (scalePt z) ps)
Summary

• Using the `data` keyword the user can introduce new structured data.

• In a data type declaration

```haskell
data DataType = Constr Type1 ... Typen | ... 
```

the type of the constructor is

```haskell
Constr :: Type1 -> ... -> Typen -> DataType
```

• **Definition by cases** on structured data can be done using the `case` construct. Wild cards (underscores) are allowed.
The most common recursive data type is the type of lists:

```haskell
data [a] = [] | a : [a]
```

Without using the special infix (and aroundfix) notation the definition of lists would read as follows:

```haskell
data List a = Nil | Cons a (List a)
```

This definition is recursive because the defined type constructor `List` occurs on the right hand side of the defining equation. We will discuss a few other common examples of recursive data.
Binary Trees

A binary tree is

- either empty,
- or a node with exactly two subtrees.
- Each node carries an integer as label.

data Tree = Null
            | Node Tree Int Tree

    deriving (Eq, Read, Show)
Examples of trees

tree0 = Null

tree1 = Node Null 3 Null
tree2 = Node (Node Null 5 Null)
  2
   (Node (Node Null 7 Null)
   5
   (Node Null 1 Null))

tree3 = Node tree2 0 tree2
Balanced trees

Creating balanced trees of depth $n$ with label indicating depth of node:

```haskell
balTree :: Int -> Tree
balTree n = if n == 0
    then Null
    else let t = balTree (n-1)
        in  Node t n t
```
Testing occurrence in a tree

Testing whether an integer occurs in a tree:

\[
\text{member} :: \text{Tree} \to \text{Int} \to \text{Bool} \\
\text{member} \ \text{Null} \ _ \ = \ \text{False} \\
\text{member} \ (\text{Node} \ l \ x \ r) \ y \ = \\
\ \ x =\ y \ || \ (\text{member} \ l \ y) \ || \ (\text{member} \ r \ y)
\]
Traversing trees: Prefix order

preord :: Tree -> [Int]
preord Null = []
preord (Node l x r) = [x] ++ preord l ++ preord r
Traversing trees: Infix and postfix

Infix and postfix order:

\[
inord \quad :: \quad \text{Tree} \rightarrow \ [\text{Int}] \\
inord \ \text{Null} \quad \quad = \quad [] \\
inord \ (\text{Node} \ l \ x \ r) \quad = \quad \text{inord} \ l \ +\ + \ [x] \ +\ + \ \text{inord} \ r
\]

\[
postord \quad :: \quad \text{Tree} \rightarrow \ [\text{Int}] \\
postord \ \text{Null} \quad \quad = \quad [] \\
postord \ (\text{Node} \ l \ x \ r) \quad = \\
\quad \quad \quad \quad \ \text{postord} \ l \ +\ + \ \text{postord} \ r \ +\ + \ [x]
\]
Structural recursion

The definitions of `member`, `preorder`, `inorder`, `postorder` are examples of structural recursion on trees.

The pattern is:

- **Base.** Define the function for the empty tree
- **Step.** Define the function for a composite tree using recursive calls to the left and right subtree.
Ordered trees

A tree \texttt{Node l x r} is \texttt{ordered} if

\texttt{member l y} implies \( y < x \) and
\texttt{member r y} implies \( x < y \) and
\texttt{l} and \texttt{r} are ordered.
Example:

$$t = \text{Node} \left( \text{Node} \left( \text{Node} \text{Null} 1 \text{Null} \right) 5 \right)$$

$$\quad \left( \text{Node} \text{Null} 7 \text{Null} \right)$$

$$9$$

$$\quad \left( \text{Node} \left( \text{Node} \text{Null} 13 \text{Null} \right) 15 \right)$$

$$\quad \left( \text{Node} \text{Null} 29 \text{Null} \right)$$
Efficient membership test

For ordered trees the test for membership is more efficient:

```haskell
member :: Tree -> Int -> Bool
member Null _ = False
member (Node l x r) y
  | y < x  = member l y
  | y == x = True
  | y > x  = member r y
```
Order preserving insertion

Order preserving insertion of a number in a tree:

\[
\text{insert :: Tree} \rightarrow \text{Int} \rightarrow \text{Tree}
\]

\[
\text{insert Null y} = \text{Node Null y Null}
\]

\[
\text{insert (Node l x r) y} =
\begin{align*}
& | y < x = \text{Node (insert l y) x r} \\
& | y == x = \text{Node l x r} \\
& | y > x = \text{Node l x (insert r y)}
\end{align*}
\]
Order preserving deletion

More difficult: Order preserving deletion.

```haskell
delete :: Int -> Tree -> Tree
delete y Null = Null
delete y (Node l x r)
  | y < x  = Node (delete y l) x r
  | y == x = join l r
  | y > x  = Node l x (delete y r)
```

join combines in an order preserving way two trees $l, r$ provided all labels in $l$ are smaller than those in $r$. 
join :: Tree -> Tree -> Tree
join t Null = t
join t s = Node t leftmost s1 where
  (leftmost, s1) = splitTree s
splitTree :: Tree -> (Int, Tree)
splitTree (Node Null x t) = (x, t)
splitTree (Node l x r) =
  (leftmost, Node l1 x r) where
  (leftmost, l1) = splitTree l
Regions

We define a recursive data type `Region` that contains shapes as basic objects and has constructors for rotation and boolean combinations of regions:

```haskell
data Region = Sh Shape
               | Rotate Float Region
               | Inter Region Region
               | Union Region Region
               | Compl Region
               | Empty
```
The meaning of regions

The meaning of a region is a set of points in the 2-dimensional plane. Therefore, in order to define the meaning of a region, we must decide when a point lies in the region.

\[
in\text{Region} :: \text{Region} \rightarrow \text{Pt} \rightarrow \text{Bool}
\]

If we set (as in the 2nd coursework)

\[
\text{type Bitmap} = \text{Pt} \rightarrow \text{Bool}
\]

then we have
\( \text{inRegion} :: \text{Region} \rightarrow \text{Bitmap} \)

i.e. \text{inRegion} transforms a region into a bitmap, which can in turn be transformed into a picture (see Coursework 2).

For example, a point \( p \) is in the region

\[ \text{Union \ reg1 \ reg2} \]

if and only if it is in \( \text{reg1} \) or in \( \text{reg2} \).
Examples of regions

A ring with outer radius 1.5 and inner radius 1:

\[
\text{ring} = \text{Inter} \left( \text{Sh} \left( \text{circle} \left(0,0\right) 1.5\right) \right) \\
\quad \quad \left( \text{Compl} \left( \text{Sh} \left( \text{circle} \left(0,0\right) 1\right) \right) \right)
\]

A tree:

\[
\text{tree} = \text{Union} \left( \text{Sh} \left( \text{rectangle} \left(-0.2,-1\right) \left(0.2,0.6\right)\right) \right) \\
\quad \quad \left( \text{Sh} \left( \text{circle} \left(0,1\right) 0.5\right) \right)
\]
Operations on Regions

Shifting:

\[
\text{shiftRegion} :: \text{Pt} \rightarrow \text{Region} \rightarrow \text{Region}
\]

\[
\text{shiftRegion} \ p \ (\text{Sh} \ sh) = \text{Sh} \ (\text{shiftShape} \ p \ sh)
\]

\[
\text{shiftRegion} \ p \ (\text{Union} \ \text{reg1} \ \text{reg2}) = \text{Union} \ (\text{shiftRegion} \ p \ \text{reg1}) \ (\text{shiftRegion} \ p \ \text{reg2})
\]

\[
\text{shiftRegion} \ p \ (\text{Inter} \ \text{reg1} \ \text{reg2}) = \text{Inter} \ (\text{shiftRegion} \ p \ \text{reg1}) \ (\text{shiftRegion} \ p \ \text{reg2})
\]
shiftRegion p (Compl reg) = Compl (shiftRegion p reg)

shiftRegion _ Empty = Empty

The union of a list of regions:

unionRegions :: [Region] -> Region

unionRegions = foldr Union Empty
Summary

• Using the `data` keyword the user can introduce new recursive data types

• Common recursive data types are lists and trees.

• Functions on recursive data types can be defined by `structural recursion`. For example, recursion on lists, trees.

• Maintaining special properties of recursive data can lead to efficient code. Example: membership for ordered trees.
Exercises

- Compute the circumference of a shape.
- Compute the area of a shape. Hint: see Hudak, page 33.
- Compute the number of corners of a shape.
- Compute the height (or depth) of a tree.
8 Proofs
Contents

• Verifying recursive programs by induction

• Timing analysis
8.1 Verifying recursive programs by induction

- Properties of recursive programs can often be proved by a suitable induction principle.

- Which induction principle is to be applied depends on the underlying data types.

- In the following we discuss
  - induction on natural numbers;
  - induction on lists;
  - induction on trees.
Induction on natural numbers

• In order to prove

   for all natural numbers $n$, property $P(n)$ holds

• one needs to prove

  ○ induction base: $P(0)$

  ○ induction step: assuming $P(n)$, show $P(n + 1)$.

Remark: The assumption $P(n)$ in the induction step is usually called induction hypothesis, abbreviated i.h.
8.1 Verifying recursive programs by induction

Example:

```haskell
f :: Int -> Int
f n = if n == 0 then 0 else f (n-1) + n
```

Show that for all natural numbers \( n \) (that is, integers \( n \geq 0 \)) the equation \( f n = \frac{n*(n+1)}{2} \) holds.

- Base: \( f 0 = 0 = \frac{0*(0+1)}{2} \).

- Step: Assume \( f n = \frac{n*(n+1)}{2} \) (i.h.). We have to show \( f (n + 1) = \frac{(n+1)*(n+2)}{2} \).

\[
f(n + 1) = f n + (n + 1) \quad \text{i.h.} \quad \frac{n * (n + 1)}{2} + n + 1 = \\
= \frac{n * (n + 1) + 2 * (n + 1)}{2} = \frac{(n + 1) * (n + 2)}{2}
\]
Induction on lists

- In order to prove

  for all finite lists $xs$, property $P(xs)$ holds

- one needs to prove

  - induction base: $P([])$
  
  - induction step: assuming $P(xs)$, show $P(x : xs)$ holds for all $x$.

Here $P(xs)$ is the induction hypothesis.
Example:

\[
\text{[a]} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]

\[
\text{ } \quad \quad \text{[] ++ } \text{ys} \quad = \text{ys}
\]

\[
\text{ } \quad \quad \text{(x:xs) ++ } \text{ys} \quad = \text{x : (xs ++ ys)}
\]

Show that \((++)\) is associative, that is the equation

\[
(\text{xs ++ ys}) ++ \text{zs} = \text{xs ++ (ys ++ zs)}
\]

holds for all \(x, y, z\).

We prove this by list induction on \(x\).
8.1 Verifying recursive programs by induction

- **Base:** 
  \( ([] ++ ys) ++ zs = ys ++ zs = [] ++ (ys ++ zs) \).

- **Step:** Assume 
  \( (xs ++ ys) ++ zs = xs ++ (ys ++ zs) \) (i.h.).

  We have to show

  \[
  ((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)
  \]

  for all \( x \).

  \[
  ((x:xs) ++ ys) ++ zs \\
  = (x : (xs ++ ys)) ++ zs \quad \text{(by def. of ++)} \\
  = x : ((xs ++ ys) ++ zs) \quad \text{(by def. of ++)} \\
  = x : (xs ++ (ys ++ zs)) \quad \text{(by i.h.)} \\
  = (x:xs) ++ (ys ++ zs) \quad \text{(by def. of ++)}
  \]
• Exercises

○ Prove $xs ++ [] = xs$

○ Recall

$$\text{length} :: [a] \rightarrow \text{Int}$$
$$\text{length} [] = 0$$
$$\text{length} (x : xs) = 1 + \text{length} \; xs$$

Prove $\text{length} (xs ++ ys) = \text{length} \; xs + \text{length} \; ys$. 
Consider the function

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

Prove \( \text{reverse} \ (xs \ ++ \ ys) = \text{reverse} \ ys \ ++ \ \text{reverse} \ xs \).
Induction on trees

data Tree a = Null
  | Node (Tree a) a (Tree a)

• In order to prove
  for all finite trees $xt$, property $P(xt)$ holds

• one needs to prove
  ○ induction base: $P(\text{Null})$
  ○ induction step: assuming $P(xt)$ and $P(yt)$, show $P(\text{Node } xt x yt)$ holds for all $x$.

Here we have two induction hypotheses: $P(xt)$ and $P(yt)$. 

MMISS: Recursion induction
• Example:

```
height :: Tree a -> Int
height Null     = 0
height (Node xt _ yt) = 1 + max (height xt) (height yt)
```

```
preord :: Tree a -> [a]
predord Null     = []
predord (Node xt x yt) = [x] ++ predord xt ++ predord yt
```

Show

```
length (predord xt) <= 2^(height xt) - 1
```

holds for all trees xt.

We prove this by tree induction:
• Base:

\[ \text{length (preord Null)} = \text{length} \; [] = 0 = 2^{\text{height Null}} - 1. \]

• Step: Assume

\[ \text{length (preord xt)} \leq 2^{\text{height xt}} - 1 \]

and

\[ \text{length (preord yt)} \leq 2^{\text{height yt}} - 1 \; (\text{i.h.}). \]

We have to show

\[ \text{length(preord(Node xt x yt)) \leq 2^{\text{height(Node xt x yt)}}-1} \]

for all \( x \).
\[
\text{length (preord (Node xt x yt))} \\
= \text{length ([x] ++ preord xt ++ preord yt)} \quad (\text{def. preord}) \\
= 1 + \text{length(preord xt)} + \text{length(preord yt)} \\
\leq 1 + 2^{\text{(height xt)}} - 1 + 2^{\text{(height yt)}} - 1 \quad (\text{by i.h.}) \\
= 2^{\text{(height xt)}} + 2^{\text{(height yt)}} - 1 \\
\leq 2 \times 2^{\text{(max (height xt) 2^(height yt))}} - 1 \\
= 2^{1 + \text{max (height xt) (height yt)}} - 1 \\
\leq 2^{\text{(height (Node xt x yt))}} - 1 \quad (\text{def. height})
\]
8.2 Timing analysis

- The **run-time** of an expression is measured in terms of the **number of reduction steps** necessary to evaluate it.

- The run-time of a function $f$ is the asymptotic behaviour of the runtime of $f \, x$ depending on the size of $x$.

- Asymptotic behaviour is expressed mathematically by the ‘big Oh-’ and similar notations (*Algorithms & Complex.*).

- In the following we discuss
  - the run-times of recursive functions on lists and trees;
  - evaluation strategy dependent run-times (lazy vs. eager).
Run-times of functions on lists

• We define the size of a list \( xs \) as its length.

• Example: To compute \( xs \ ++ \ ys \) takes \( n + 1 \) reduction steps where \( n \) is the length of \( xs \).

Proof by list induction on \( xs \):

○ Base: \([\ ] \ ++ \ ys \leadsto ys \) (one reduction)

○ Step: Assume computing \( xs \ ++ \ ys \) needs \( n + 1 \) reductions.

  We have to show \( (x:xs) \ ++ \ ys \) needs \( n + 2 \) reductions:

  \( (x:xs) \ ++ \ ys \leadsto x:(xs \ ++ \ ys) \leadsto \text{in} \ n + 1 \text{ steps to value.} \)

• Therefore \((++\) has run-time complexity \( O(n) \).
Example: Our program for reversing list has run-time complexity $O(n^2)$.

Proof: We show by list induction that evaluation of `reverse xs` needs

$$\frac{n \times (n + 3)}{2} + 1$$

reductions, where $n$ is the length of the list.

○ Base:

`reverse [] ⇝ []` (one step).

$$\frac{0 \times (n + 3)}{2} + 1 = 1$$
8.2 Timing analysis

○ Step:

\[ \text{reverse} \ (x : xs) \leadsto \text{reverse} \ xs \ ++ \ [x]. \]

By i.h. \( \text{reverse} \ xs \) needs \( \frac{n \times (n+3)}{2} + 1 \) steps.

It is easy to see that \( \text{reverse} \ xs \) has length \( n \) (Exercise).

Hence, by the previous result on \( (+++) \) the total number of steps is

\[ 1 + \left( \frac{n \times (n+3)}{2} + 1 \right) + (n + 1) = \ldots = \frac{(n + 1) \times (n + 4)}{2} + 1 \]
• Recall the linear program for reversing lists:

\[
\text{reverse1 :: } [a] \to [a] \\
\text{reverse1 } xs = \text{revConc } xs \; []
\]

\[
\text{revConc :: } [a] \to [a] \to [a] \\
\text{revConc } [] \; ys = ys \\
\text{revConc } (x:xs) \; ys = \text{revConc } xs \; (x:ys)
\]

It is easy to prove that the function \text{revConc} has run-time complexity \( O(n) \) (Exercise) and

\[
\text{reverseConc } xs \; ys = \text{reverse } xs ++ ys \; (Exercise).
\]
Run-times of functions on trees

- We define the size of a tree as the number of its nodes.

- Example: To compute \texttt{preord xt} takes $O(n^2)$ steps where $n$ is the size of the tree \texttt{xt}.

- Exercise: Prove this!
• A linear program for `preord` (using a similar idea as in the linear implementation of `reverse`):

```haskell
preord :: Tree a -> [a]
preord xt = preConc xt [] where
  preConc Null zs = zs
  preConc (Node xt x yt) zs =
    x : preConc xt (preConc yt zs)
```

It easy to prove that the function `preConc` has run-time complexity $O(n)$ (Exercise).
Lazy evaluation versus eager evaluation

- Recall that Haskell applies a lazy evaluation strategy: Subexpressions are reduced only when their value is needed. This strategy is achieved by outermost reduction, which means reduction takes place as close to the root of the expression as possible.

- Another reduction strategy is eager evaluation which is achieved by innermost reduction, i.e. reduction takes place as close to the leaves of the expression as possible.

- Most programming languages apply eager evaluation (e.g. ML, Lisp, Java, . . .).
8.2 Timing analysis

• None of the strategies outperforms the other in general.

• But, lazy evaluation is safe concerning termination: When an expression terminates by some strategy, then it also terminates with lazy evaluation.

• In the previous examples of timing analysis the evaluation strategy did not matter.

• In the following example the strategy *does* matter.
Computing the first $n$ labels of a tree in prefix order:

\[
\text{takeTree} :: \text{Int} \rightarrow \text{Tree } a \rightarrow [a] \\
\text{takeTree} \ n \ xt = \text{take} \ n \ (\text{preord} \ xt)
\]

\[
\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take} \ n \_ | n <= 0 = [] \\
\text{take} \_ [] = [] \\
\text{take} \ n \ (x:xs) = x : \text{take} \ (n-1) \ xs
\]

Applying lazy evaluating this program has time complexity $O(n)$.

Note that the complexity is independent of $xt$!
For example

\[
\text{takeTree 1 (Node } x t \ x \ y t) \rightsquigarrow \\
\text{take 1 (preord (Node } x t \ x \ y t)) \rightsquigarrow \\
\text{take 1 ([x] ++ preord } x t \ ++ \ \text{preord } y t) \rightsquigarrow \\
\text{take 1 (} x : (\text{preord } x t \ ++ \ \text{preord } y t)) \rightsquigarrow \\
x : \text{take 0 (preord } x t \ ++ \ \text{preord } y t) \rightsquigarrow \ [x]
\]

- With eager evaluation the time complexity would be \( O(\max(n, m)) \) where \( m \) is the size of \( x t \), because \text{preord } x t \) would be evaluated completely first.
Summary and concluding remarks

• In functional programming functions are defined by equations which can be used to apply rigorous reasoning to prove properties of programs.

• Important and useful proof principles are induction on numbers, lists and trees.
• Timing analysis can be done by counting reductions.

• The reduction strategy can have an impact on the run-time complexity.
Exercises

- Prove \( \text{length} \left( \text{filter} \ p \ \text{xs} \right) \leq \text{length} \ \text{xs} \).

- Prove \( \text{map} \ (f \ . \ g) \ \text{xs} = \text{map} \ f \ \left( \text{map} \ g \ \text{xs} \right) \),

recalling that \( (f \ . \ g) x = f (g \ x) \).
• **Prove** $\text{length} \ (\text{reverse} \ xs) = \text{length} \ xs$.

• **Prove** $\text{reverseConc} \ xs \ ys = \text{reverse} \ xs \ ++ \ ys$ and conclude from this the correctness of the function $\text{reverse1}$, that is,

$$\text{reverse1} \ xs = \text{reverse} \ xs.$$
9 Abstract Data Types
Contents

• The concept of an Abstract Data Type (ADT)

• Implementation of ADTs by modules

• The ADT of Sets

• Minimality vs. efficiency
9.1 The concept of an Abstract Data Type

• An Abstract Data Type (ADT) consists of a bunch of data types and functions.

• An ADT has a precise interface (also called signature) describing the available data types and function.

• Details of how data are constructed and functions are defined may be hidden.

• The main advantage of an ADT is that its implementation may be changed without affecting the validity of programs using the ADT.
9.2 Implementation of ADTs by modules

• A module consists of:
  ○ Definitions of types, functions, classes
  ○ Declaration of the definitions that are visible from outside (interface, signature).

• Syntax:

```
module Name (visible names) where body
```

  ○ visible names may be empty

• The main idea behind modules is to control visibility by encapsulation.
Example: Storage (Store)

- Type of storage parametrised by index (or address) type \( a \) and value type \( b \).

\[
\text{Store } a \ b
\]

- Empty storage: \( \text{initial} :: \text{Store } a \ b \)

- Reading a value:
  \[
  \text{value} :: \text{Eq } a \Rightarrow \text{Store } a \ b \rightarrow a \rightarrow \text{Maybe } b
  \]
  - Uses \( \text{data Maybe } b = \text{Nothing} \mid \text{Just } b \)

- Writing:
  \[
  \text{update} :: \text{Eq } a \Rightarrow \text{Store } a \ b \rightarrow a \rightarrow b \rightarrow \text{Store } a \ b
  \]

- Module declaration (interface)
module Store(  
    Store,  
    initial, -- :: Store a b  
    value,  -- :: Eq a => Store a b -> a -> Maybe b  
    update,  -- :: Eq a => Store a b -> a -> b -> Store a b  
) where

(Signature not necessary, but helpful)
9.2 Implementation of ADTs by modules

- Body of module: Implementation of storage as a list of pairs

```haskell
type Store a b = [(a, b)]

initial :: Store a b
initial  = []
```
value :: Eq a => Store a b -> a -> Maybe b
value ls a =
    case [b | (x,b) <- ls, a == x] of
    b : _   -> Just b
    []      -> Nothing

update :: Eq a => Store a b -> a -> b -> Store a b
update ls a b = (a, b) : ls
• The complete code for the module (interface + body) must be written in the file \texttt{Store.hs} (capital \texttt{S}!).

• The ADT \texttt{Store} is also known under the name \texttt{Map}.

• Note that nothins is exported that would reveal that we have implemented stores via lists. This is important since it gives us the possibility to change the implementation of the ADT \texttt{Store} without changing its interface.
Importing a module

- `import Name (identifiers)`
  - The `identifiers` declared in module `Name` are imported.
  - If `(identifiers)` is omitted, then everything is imported.

Example: The line

```
import Store
```

imports everything from the interface of the module `Store`, that is, the data type `Store` and the functions `initial`, `value`, `update`. 
Using the ADT Store

Storing the values $x_0, x_1, \ldots, x_n$
at the addresses $k, k+1, \ldots, k+n$.

```haskell
import Store (Store, initial, value, update)

mkStore :: [a] -> Int -> Store Int a
mkStore [] k = initial
mkStore (x:xs) k = update (mkStore xs (k+1)) k x
```
Reimplementing the ADT Store using functions

type Store a b = a -> Maybe b

initial :: Store a b
initial = \_ -> Nothing

value :: Eq a => Store a b -> a -> Maybe b
value f a = f a
update :: Eq a => Store a b -> a -> b -> Store a b

update f a b
  = \x -> if x == a then Just b else f x
9.3 The ADT of Sets

In the following we discuss the important ADT of sets in some detail.
Common requirements on a data type of sets are:

1. There is an **empty set**: \( \emptyset \)

2. We can **insert** an element in a set: \( \{ x \} \cup s \)

3. We can test if an element is a **member** of a set: \( x \in s \)

4. We can compute the **union** of two sets: \( s \cup t \)

5. **Filtering**: Given a set \( s \) and a property \( p \) of elements we can compute a set containing exactly those members of \( s \) that have property \( p \):
\[
\{ x \in s \mid p(x) = \text{True} \}
\]
6. **Extensionality:** Two sets $s, t$ are equal when they contain the same elements, that is, an element is a member of $s$ if and only if it is a member of $t$:

$$s = t \iff \forall x (x \in s \iff x \in t)$$

We do not (yet) require a test for deciding equality of sets.
Implementation of sets

- As type of elements of sets we admit any type \( a \) that has an equality test (i.e. \( a \) must be an instance of the class \( \text{Eq} \)).

- We will begin with a discussion of two implementations of sets:
  
  (a) Sets as boolean functions.

  (b) Sets as lists of elements.
module Set (  
Set,       -- type constructor  
empty,     -- Set a  
insert,    -- Eq a => a -> Set a -> Set a  
member,    -- Eq a => a -> Set a -> Bool  
union,     -- Eq a => Set a -> Set a -> Set a  
sfilter,   -- Eq a => (a -> Bool) -> Set a -> Set a  
) where
Implementing sets as boolean functions

type Set a = a -> Bool

empty :: Set a
empty = \x -> False

insert :: Eq a => a -> Set a -> Set a
insert x p = \y -> y == x || p y

member :: Eq a => a -> Set a -> Bool
member x p = p x
union :: Eq a => Set a -> Set a -> Set a
union p q = \x -> p x || q x

sfilter :: Eq a => (a -> Bool) -> Set a -> Set a
sfilter test p = \x -> p x && test x
Implementing sets as lists

type Set a = [a]

empty :: Set a
empty = []

insert :: Eq a => a -> Set a -> Set a
insert x xs = x : xs

member :: Eq a => a -> Set a -> Bool
member \( x \) \( \) \( xs \) = elem \( x \) \( xs \)

\[
\text{union :: Eq } a \Rightarrow \text{Set } a \rightarrow \text{Set } a \rightarrow \text{Set } a
\]
union \( xs \) \( ys \) = \( xs ++ ys \)

\[
\text{sfilter :: Eq } a \Rightarrow (a \rightarrow \text{Bool}) \rightarrow \text{Set } a \rightarrow \text{Set } a
\]
sfilter \( \text{test} \) \( xs \) = filter \( \text{test} \) \( xs \)
Using the ADT Set

import Set (Set, empty, insert, member, union, sfilter)

singleton :: Eq a => a -> Set a
singleton x = insert x empty

delete :: Eq a => a -> Set a -> Set a
delete x s = sfilter ( /= x) s
meet :: Eq a => Set a -> Set a -> Set a
meet s t = sfilter ('member' s) t

minus :: Eq a => Set a -> Set a -> Set a
minus s t = sfilter (\x -> not (x 'member' t)) s

unions :: Eq a => [Set a] -> Set a
unions = foldr union empty

listToSet :: Eq a => [a] -> Set a
listToSet xs = foldr insert empty xs
Extending the ADT Set

- Can we define infinite sets, in particular a universal set that contains all elements of type a?
- Can we compute the complement of a set?
- Easy with the implementation based on boolean functions:
  
  ```haskell
  universal :: Set a
  universal = \x -> True
  
  complement :: Set a -> Set a
  complement p = not . p
  ```

- None of these are possible with sets implemented by lists.
• Can we
  ○ test whether a set is empty?
  ○ compute the number of elements of a set?
  ○ decide inclusion/equality between sets?

• The implementation based on boolean functions inhibits the definition of any of these operations.

• On the other hand there are no problems with the list implementation:
9.3 The ADT of Sets

isEmpty :: Set a -> Bool
isEmpty xs = case xs of
  []   -> True
  _:_  -> False

card :: Eq a => Set a -> Int
card xs = length (removeDuplicates xs)

removeDuplicates :: Eq a => [a] -> [a]
removeDuplicates [] = []
removeDuplicates (x:xs) = (if x 'elem' ys then [] else [x]) ++ ys
where ys = removeDuplicates xs
9.3 The ADT of Sets

subset :: Eq a => Set a -> Set a -> Bool
subset [] ys = True
subset (x:xs) ys = x 'elem' ys && sub xs ys

eqset :: Eq a => Set a -> Set a -> Bool
eqset s t = s 'subset' t && t 'subset' s
In the following we work with the implementation of Set based on lists.

- Can we list the members of a set?
  First attempt:

  \[
  \text{setToList :: Eq } a \Rightarrow \text{Set } a \rightarrow [a] \\
  \text{setToList } xs = \text{removeDuplicates } xs \\
  \]

  Does not work because equality of sets is not respected:

  \[
  \text{setToList } [1,2] \rightarrow [1,2] \\
  \text{setToList } [2,1] \rightarrow [2,1] \\
  \]
The problem can be solved, if there is an order on elements:

```haskell
setToList :: Ord a => Set a -> [a]
setToList xs = qsort (removeDuplicates xs)
```

If we use only ordered and repetition-free lists as representations of sets, then we can simply define:

```haskell
setToList :: Ord a => Set a -> [a]
setToList xs = xs
```
9.4 Minimality versus Efficiency

• The interface of our first ADT of sets was minimal in the sense that we omitted a function if it could be defined by other functions (for example singleton, delete, meet, minus were defined in this way).

• The advantage of minimal declarations is that less work needs to be done when the implementation of the module is changed.

• But often functions can be implemented more efficiently by making direct use of the representation of sets. In this case it is advantageous to include these functions in the interface of the ADT.
For example, our definition of `meet` has time complexity \( O(n^2) \), but if sets are represented by ordered lists one can achieve linear time:

\[
\text{meet1 :: Ord a => } [a] \rightarrow [a] \rightarrow [a]
\]

\[
\text{meet1 } [] \_ = []
\]

\[
\text{meet1 } \_ [] = []
\]

\[
\text{meet1 } (x:xs) (y:ys)
\]

\[
| x < y = \text{meet1 } xs (y:ys)
\]

\[
| x == y = x : \text{meet1 } xs ys
\]

\[
| x > y = \text{meet1 } (x:xs) ys
\]
Without ordering:

```haskell
meet2 :: Eq a => [a] -> [a] -> [a]
meet2 xs ys = [x | x <- xs, x 'elem' ys]
```

-- `meet1` [1,3..10000] [2,4..10000]
-- `meet2` [1,3..10000] [2,4..10000]

- The implementation of sets as lists, or ordered lists is simple, but unsuitable for representing large finite sets. If one uses so-called ‘AVL-tree’ for representing sets, then membership test, as well as insertion and deletion of elements can be performed in logarithmic time (as opposed to linear time in the implementation with lists).
- **AVL-trees** are special (almost) balanced ordered trees. They are discussed in any standard textbook on data structures.
Summary

- An abstract data type (ADT) consists of one or more data types together with some functions. Details of how data are constructed and functions are defined are hidden.
- Abstract data types support modularization and make code more robust: The implementation of an ADT may change without other programs using it being affected.
- An important abstract data type is the ADT of sets. Sets can be represented in different ways allowing for implementations of different operations.
- The choice of representation may also affect the efficiency of set operations.
Exercises

- Implement the operations ‘set difference’, ‘powerset’, ‘set inclusion test’ and ‘set disjointness test’ with sets represented by lists.

- Does an ordering help improving efficiency?

- Estimate the (time) complexities of your implementations.

- Which of the operations above can be implemented with sets represented as boolean functions?
10 Input/Output
Contents

• Input/Output in functional languages

• Communication with files

• Actions as values

• Random numbers
10.1 Input/Output in functional languages

- **Problem:** Functions with side effects are not referentially transparent (that is, not all parameters the function depends on are visible).
  - for example `readString :: () -> String` ??

- **Solution:**
  Side effects are marked by `IO` — actions
  - Actions can be combined with actions only
  - „once IO, always IO“
The type `IO a`

- `IO a` is the type of actions returning a value of type `a` when executed.
- The value returned by an action can be used in the next action.
- Two actions are combined by the operator

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```
Actions as an abstract data type

```
type IO a
(>>=) :: IO a
  -> (a -> IO b)
  -> IO b

return :: a -> IO a
```

Environment
• If we are given

\[
\begin{align*}
\text{action1} & \quad :: \text{IO } a \\
\text{action2} & \quad :: a \rightarrow \text{IO } b
\end{align*}
\]

then the action

\[
\text{action1} \gg= \text{action2} \quad :: \text{IO } b
\]

first does \text{action1} and then \text{action2} \( x \) where \( x \) is the value returned by \text{action1}.
Some predefined actions (Prelude)

- Reading a line from stdin (standard input):
  
  ```haskell```
  ```
  getLine :: IO String
  ```
  
- Writing string to stdout (standard output):
  
  ```haskell```
  ```
  putStr :: String -> IO ()
  ```

- Writing string to stdout and inserting a new line:
  
  ```haskell```
  ```
  putStrLn :: String -> IO ()
  ```
A simple example

- **Echo:**

  ```haskell
echo :: IO ()
echo = getLine >>= putStrLn >>= \_ -> echo
  ```

- **Reversed echo:**

  ```haskell
  ohce :: IO ()
ohce = getLine >>= putStrLn . reverse >> ohce
  ```

  where `(>>>)` is predefined:

  ```haskell
  (>>>) :: IO a -> IO b -> IO b
  f >> g = f >>= \_ -> g
  ```
The meaning of \( f \gg g \) is

“First do \( f \), then \( g \),

throwing away the value returned by \( f \)”. 

The do-notation

- Syntactic sugar for IO:

\[
\text{echo} = \text{getLine} \triangleright= \text{putStrLn} \triangleright \text{echo}
\]

\[
\text{echo} = \text{do} \ s \leftarrow \text{getLine} \\
\quad \triangleright= \text{putStrLn} \ s \\
\quad \triangleright \text{echo}
\]

- Layout: Be careful with indentation in a do construction.
Example: A simple notebook

A simple notebook using the ADT Store.

- Updating: `u <Name> <Value>`

- Looking up a value: `v <Name>`

Note that the only way to modify and use the storage is via the functions `update` and `value` provided by the ADT Store.
import Store

notebook :: IO ()
notebook = loop initial where
  loop :: Store String String -> IO ()
  loop st =
    do s <- getLine
       case words s of
         "u"::name::value:_ -> loop (update st name value)
         "v"::name:_ ->
           do putStrLn (case value st name of
                        Just val -> val
                        Nothing -> "Unknown: "++name)
              loop st
         _ -> return ()
10.2 Communication with files

- In Prelude predefined:
  
  - Writing into files (override, append):
    
    ```haskell
    type FilePath = String
    writeFile :: FilePath -> String -> IO ()
    appendFile :: FilePath -> String -> IO ()
    ```

  - Reading a file (lazy):
    
    ```haskell
    readFile :: FilePath -> IO String
    ```
Example: Counting characters, words, lines

```haskell
wc :: String -> IO ()
wc file =
  do c <- readFile file
     putStrLn (show (length (lines c)) ++ " lines ")
     putStrLn (show (length (words c)) ++ " words, and ")
     putStrLn (show (length c) ++ " characters. ")
```
10.3 Actions as values

- Actions are normal values.
- This makes the definition of control structures very easy.
- Better than any imperative language.
Predefined control structures (Prelude)

- Sequencing a list of actions:

```haskell
sequence :: [IO a] -> IO [a]
sequence (c:cs) = do x <- c
                      xs <- sequence cs
                      return (x:xs)
```

`sequence [act1, ..., actn]` is the action that does the actions `act1, ..., actn` (in that order) and returns the list of results.
• **Special case:** \([()]\) as \((())\)

```haskell
sequence_ :: [IO ()] -> IO ()
```
10.4 Random numbers

- Example of actions as values: Random numbers.

Module `Random`, class `Random`

```haskell
class Random a where
  randomRIO :: (a,a) -> IO a
  randomIO :: IO a
```

- For example, `randomRIO (0,36)` is the action that returns a random number between 0 and 36.

- `Random` further contains
  - Random generators for pseudo random numbers
  - Infinite lists of random numbers
Example: Random words

- Repeating an action a random number of times:

  ```haskell
  atmost :: Int -> IO a -> IO [a]
  atmost most a =
      do l <- randomRIO (1, most)
         sequence (replicate l a)
  ```

- Random string:

  ```haskell
  randomStr :: IO String
  randomStr = atmost 10 (randomRIO ('a','z'))
  ```
Roulette

- Start with 100 pounds.
- Choose a number between 1 and 36.
- Say how many rounds you want to play.
- Each round costs 1 pound.
- In each round a number between 0 and 36 is randomly played.
- If it is your number, you win 36 pounds.
- The game stops if you run out of money.
We use a variant of `until` that accepts a `next` function with side effects:

```haskell
untilIO :: (a -> Bool) -> (a -> IO a) -> a -> IO a
untilIO stop next x =
  if stop x
  then return x
  else do x1 <- next x
          untilIO stop next x1
```

Compare this with

```haskell
until :: (a -> Bool) -> (a -> a) -> a -> a
until stop next x =
  if stop x
  then x
  else let x1 = next x
       in until stop next x1
```
import Random (randomRIO)

roulette :: IO ()
roulette =
    do putStr "You have 100 pounds. Which number do you choose? "
       n <- getLine
    putStrLn "How many rounds do you want to play maximally? "
    r <- getLine
    let mine = read n :: Int
        maxRounds = read r :: Int
        stop (k,money) = k >= maxRounds || money <= 0
        next (k,money) = do x <- randomRIO (0,36)
                          let win = if x == mine then 36 else 0
                              return (k+1,money - 1 + win)
        (rounds,endMoney) <- untilIO stop next (0,100)
    putStrLn ("\nYou played "+(show rounds)+" rounds "+
                      "and have now "+(show endMoney)+" pounds")
Summary

• Input/Output in Haskell by Actions
  ◦ Actions (type `IO a`) are functions with side effects
  ◦ Combination of actions by
    
    \[(>>=) \:: \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b\]
    
    \[
    \text{return } :: a \to \text{IO } a
    \]
  ◦ do-Notation

• Various functions from the standard library:
  ◦ Prelude: `getLine`, `putStr`, `putStrLn`, `writeFile`, `readFile`
  ◦ Modules: `IO`, `Random`,

MMISS: Input/Output
• **Actions are normal values.**

• **Useful combinators for actions:**

\[
\text{sequence} :: [\text{IO } a] \rightarrow \text{IO } [a]
\]

\[
\text{sequence\_} :: [\text{IO } ()] \rightarrow \text{IO } ()
\]

\[
\text{untilIO} :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow \text{IO } a) \rightarrow a \rightarrow \text{IO } a
\]
Exercises

- Write an interactive translation program, say from English to French and vice versa. Use a dictionary of the form 
  \[\{(apple, pomme), (street, rue), (sun, soleil), \ldots\}\].

- Modify the program `saveCW` such that the statistical data are written to the file as well.

- Modify the program `roulette` such that after each 100 rounds the player is informed about their current account and given the choice whether or not to continue.
11 Infinite lists
Contents

- Lazy infinite lists
- Programming with infinite lists
- Corecursion
Lazy infinite lists

Using recursion one can create an infinite list:

```
ones :: [Int]
ones = 1 : ones
```

Unfolding the definition one obtains

```
ones = 1 : ones
    = 1 : 1 : ones
    = 1 : 1 : 1 : ones
    = 1 : 1 : 1 : 1 : ones
    ...
```

Hence `ones` is an infinite list of 1s.

```
ones = [1,1,1,1, ...]
```
Although it is impossible to compute (and print) the complete list \texttt{ones} one can compute parts of it

\texttt{head ones = head (1 : ones) = 1}

Note that due to the lazy evaluation strategy the right occurrence of \texttt{ones} is not evaluated.

Infinite lists are often called \texttt{streams}. 
iterate: iterating a function arbitrarily often

In Haskell’s Prelude we find the following higher-order function:

\[
iterate \quad :: \quad (a \rightarrow a) \rightarrow a \rightarrow [a] \\
iterate \ f \ x \quad = \quad x : iterate \ f \ (f \ x)
\]

Unfolding the definition results in

\[
iterate \ f \ x = x : iterate \ f \ (f \ x) \\
= x : f \ x : iterate \ f \ (f \ (f \ x)) \\
= x : f \ x : f \ (f \ x) : iterate \ f \ (f \ (f \ (f \ x))) \\
\vdots
\]
For example

\[
\text{iterate } (+3) 0 = 0 : \text{iterate } (+3) 3 \\
\quad = 0 : 3 : \text{iterate } (+3) 6 \\
\quad = 0 : 3 : 6 : \text{iterate } (+3) 9 \\
\quad \ldots
\]

Hence

\[
\text{iterate } (+3) 0 = [0,3,6,9,\ldots]
\]

take 5 (iterate (+3) 0 = [0,3,6,9,12]

The same stream can be obtained by

\[
= [0,3\ldots]
\]
The lazy Fibonacci numbers

Recall

\[
\text{fib} :: \text{Integer} \rightarrow \text{Integer}
\]
\[
\text{fib } n \\
| \ n < 0 \quad \quad = \text{error } "\text{negative argument}" \\
| \ n == 0 || n == 1 \quad = 1 \\
| \ n > 0 \quad \quad = \text{fib } (n - 1) + \text{fib } (n - 2)
\]

We want to compute efficiently the infinite list of Fibonacci numbers

\[
\text{fibs} = [\text{fib } 0, \text{fib } 1, \text{fib } 2, \text{fib } 3, \ldots ]
\]
\[
= [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots ]
\]
Here is how one finds a recursive definition of \texttt{fibs}.

\texttt{fibs} = [1 , 1 , 2 , 3 , 5 , 8 , 13 , ... ]
\hspace{1cm} = 1 : 1 : [2 , 3 , 5 , 8 , 13 , ... ]
\hspace{1cm} = 1 : 1 : [1+1,1+2,2+3,3+5,5+8, ... ]
\hspace{1cm} = 1 : 1 : \texttt{zipWith (+) [1,1,2,3,5, ...] [1,2,3,5,8, ...]}
\hspace{1cm} = 1 : 1 : \texttt{zipWith (+) fibs (tail fibs)}

Therefore

\texttt{fibs :: [Integer]}
\texttt{fibs = 1 : 1 : \texttt{zipWith (+) fibs (tail fibs)}}

Compare \texttt{fib n} and \texttt{fibs !! n} for \texttt{n = 10, 20, 30, 40, ...}
Corecursion

The recursive definitions of infinite lists we have seen follow a common pattern:

\[ \text{xs} = \ldots : \ldots \text{xs} \ldots \]

This pattern is called corecursion:

This means an infinite list \( \text{xs} \) is produced by

- defining its head (without recursive call)
- defining its tail, possibly using a recursive call to \( \text{xs} \)
Note the difference to structural recursion on lists where a finite list is consumed as input and recursive calls can be made anywhere, but with smaller arguments only.

For proving properties of corecursively defined infinite streams there exists a proof principle of coinduction.

Explaining this principle is beyond the scope of this course.

An important practical application of infinite lists is lazy communication with files: Writing into and reading from files can be done incrementally: If only an initial segment of a file is used, the rest is not computed.