Functional Programming and Verification

Tobias Nipkow

Fakultät für Informatik
TU München

`http://fpv.in.tum.de`

Wintersemester 2020/21

February 18, 2021

0. Organisatorisches

Siehe `http://fpv.in.tum.de`

Literatur

- Vorlesung orientiert sich stark an
  Thompson: *Haskell, the Craft of Functional Programming*
- Für Freunde der kompakten Darstellung:
  Hutton: *Programming in Haskell*
- Für Naturtalente: Es gibt sehr viel Literatur online.
  Qualität wechselhaft, nicht mit Vorlesung abgestimmt.

Klausur und Hausaufgaben

- Klausur online am Ende der Vorlesung
- Notenbonus mit Hausaufgaben: siehe WWW-Seite
  Wer Hausaufgaben abschreibt oder abschreiben lässt,
  hat seinen Notenbonus sofort verwirkt.
- Klausurbestehensquote WS19/20, mit und ohne Bonus:

	Mit Bonus	Ohne Bonus
Klausur	96%	40%
Wdh.-Klausur	63%	32%

Programmierwettbewerb — Der Weg zum Ruhm

- Alle 1-2 Wochen eine Wettbewerbsaufgabe
- Punktetabellen im Internet:
  - Die Top 30 jeder Woche
  - Die kumulative Top 30
- Ende des Semesters: Trophäen fuer die Top k Studenten
  $k > 0$!

Piazza: Frage-und-Antwort Forum

- Sie können Fragen stellen und beantworten (auch anonym)
  Natürlich keine Lösungen posten!
- Fragen werden an alle Tutoren weitergeleitet
- Zugang zu Piazza für FPV über Vorlesungsseite
- Auch *SIE* können Fragen beantworten!

Haskell

- Wir benutzen die Programmiersprache Haskell
- Wir benutzen GHC (Glasgow Haskell Compiler)
  und empfehlen die Umgebung VSCodium/VSCode
- Installationshinweise auf Vorlesungsseite
- Bei Problemen mit der Installation des GHC:
  Beratungstermin, siehe Vorlesungsseite
- Tutoren leisten in der Übung keine Hilfestellung mehr!

1. Functional Programming: The Idea

Functions are pure/mathematical functions:
Always same output for same input

Computation = Application of functions to arguments

Example 1

In Haskell:

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

In Java:

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

Example 2

In Haskell:

```
wellknown [] = []
wellknown (x:xs) = wellknown ys ++ [x] ++ wellknown zs
   where ys = [y | y <- xs, y <= x]
         zs = [z | z <- xs, x < z]
```

In Java:

```
void sort(int[] values) {
   if (values ==null || values.length==0){ return; }
   this.numbers = values;
   number = values.length;
   quicksort(0, number - 1);
}

void quicksort(int low, int high) {
   int i = low, j = high;
   int pivot = numbers[low + (high-low)/2];
   while (i <= j) {
      while (numbers[i] < pivot) { i++; }
      while (numbers[j] > pivot) { j--; }
      if (i <= j) {exchange(i, j); i++; j--; }
   }
   if (low < j) quicksort(low, j);
   if (i < high) quicksort(i, high);
}

void exchange(int i, int j) {
   int temp = numbers[i];
   numbers[i] = numbers[j];
   numbers[j] = temp;
}
```

There are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies.

The other way is to make it so complicated that there are no obvious deficiencies.

From the Turing Award lecture by Tony Hoare (1985)

Characteristics of functional programs

elegant	
expressive	
concise	
readable	
predictable	pure functions, no side effects
provable	it's just (very basic) mathematics!

Aims of functional programming

- Program at a high level of abstraction:
  not bits, bytes and pointers but whole data structures
- Minimize time to read and write programs:
  $\Rightarrow$ reduced development and maintenance time and costs
- Increased confidence in correctness of programs:
  clean and simple syntax and semantics
  $\Rightarrow$ programs are easier to
  - understand
  - test (Quickcheck!)
  - prove correct

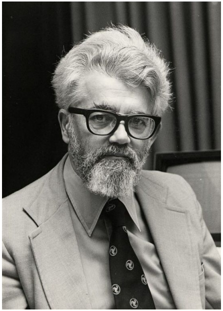
Historic Milestones

1930s

Alonzo Church develops the lambda calculus,
the core of all functional programming languages.

Historic Milestones

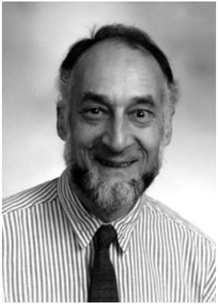
1950s



John McCarthy (Turing Award 1971) develops Lisp,
the first functional programming language.

Historic Milestones

1970s



Robin Milner (FRS, Turing Award 1991) & Co. develop ML, the first modern functional programming language with *polymorphic types* and *type inference*.

Historic Milestones

1987

Haskell
A Purely Functional Language

An international committee of researchers initiates the development of Haskell, a standard lazy functional language.

Popular languages based on FP

F#	(Microsoft)	=	ML for the masses
Erlang	(Ericsson)	=	distributed functional programming
Scala	(EPFL)	=	Java + FP

FP concepts in other languages

Garbage collection:	Java, C#, Python, Perl, Ruby, Javascript
Higher-order functions:	Java, C#, Python, Perl, Ruby, Javascript
Generics:	Java, C#
List comprehensions:	C#, Python, Perl 6, Javascript
Type classes:	C++ "concepts"

Why we teach FP

- FP is a fundamental programming style (like OO!)
- FP is everywhere: Javascript, Scala, Erlang, F# ...
- It gives you the edge over Millions of Java/C/C++ programmers out there
- FP concepts make you a better programmer, no matter which language you use
- To show you that programming need not be a black art with magic incantations like `public static void` but can be a science

2. Basic Haskell

- Notational conventions
- Type `Bool`
- Type `Integer`
- Guarded equations
- Recursion
- Syntax matters
- Types `Char` and `String`
- Tuple types
- Do's and Don'ts

2.1 Notational conventions

`e::T` means that expression e has type T

Function types:	Mathematics	Haskell
	$f : A \times B \rightarrow C$	`f :: A -> B -> C`

Function application:	Mathematics	Haskell
	$f(a)$	`f a`
	$f(a,b)$	`f a b`
	$f(g(b))$	`f (g b)`
	$f(a,g(b))$	`f a (g b)`

Prefix binds stronger than infix:

`f a + b` means `(f a) + b`

not `f (a + b)`

2.2 Type `Bool`

Predefined: `True` `False` `not` `&&` `||` `==`

Defining new functions:

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

xor2 :: Bool -> Bool -> Bool
xor2 True  True   =  False
xor2 True  False  =  True
xor2 False True   =  True
xor2 False False  =  False
```

This is an example of pattern matching.
The equations are tried in order. More later.

Is `xor x y == xor2 x y` true?

Testing with QuickCheck

Import test framework:

```
import Test.QuickCheck
```

Define property to be tested:

```
prop_xor2 x y =
  xor x y == xor2 x y
```

Note naming convention `prop_...`

Check property with GHCi:

```
> quickCheck prop_xor2
```

GHCi answers

```
+++ OK, passed 100 tests.
```

QuickCheck

- Essential tool for Haskell programmers
- Invaluable for regression tests
- Important part of exercises & homework
- Helps you to avoid bugs
- Helps us to discover them

Every nontrivial Haskell function
should come with one or more QuickCheck properties/tests

Typical test:

```
prop_f x y =
  f_efficient x y == f_naive x y
```

`V1.hs`

For GHCi commands (`:l` etc) see home page

2.3 Type `Integer`

Unlimited precision mathematical integers!
Predefined: `+ - * ^ div mod abs == /= < <= > >=`

There is also the type `Int` of at least 29-bit integers.

Warning: On my Mac:

- `(2::Integer) ^ 64 = 18446744073709551616`
- `(2::Int) ^ 64 = 0`

==, <= etc are overloaded and work on many types!

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

Evaluation:

```
sq (sq 3) = sq 3 * sq 3
          = (3 * 3) * (3 * 3)
          = 81
```

Evaluation of Haskell expressions
means
Using the defining equations from left to right.

2.4 Guarded equations

Example: maximum of 2 integers.

```
max2 :: Integer -> Integer -> Integer
max2 x y
  | x >= y      = x
  | otherwise   = y
```

Haskell also has `if-then-else`:

```
max2 x y  =  if x >= y then x else y
```

True?

```
prop_max2_assoc x y z =
  max2 x (max2 y z) == max2 (max2 x y) z
```

2.5 Recursion

Example: x^n (using only $*$, not ^)

```
-- pow x n returns x to the power of n
pow :: Integer -> Integer -> Integer
pow x n  =  ???
```

Cannot write $\underbrace{x * \cdots * x}_{n \text{ times}}$

Two cases:

```
pow x n
  | n == 0  = 1                -- the base case
  | n > 0   = x * pow x (n-1)  -- the recursive case
```

More compactly:

```
pow x 0  =  1
pow x n | n > 0  =  x * pow x (n-1)
```

Evaluating pow

```
pow x 0 = 1
pow x n | n > 0 = x * pow x (n-1)
```

```
pow 2 3 = 2 * pow 2 2
        = 2 * (2 * pow 2 1)
        = 2 * (2 * (2 * pow 2 0))
        = 2 * (2 * (2 * 1))
        = 8
```

```
> pow 2 (-1)
```

GHCi answers

```
*** Exception: PowDemo.hs:(1,1)-(2,33):
    Non-exhaustive patterns in function pow
```

Partially defined functions

```
pow x n | n > 0  =  x * pow x (n-1)
```

versus

```
pow x n  =  x * pow x (n-1)
```

- call outside intended domain raises exception
- call outside intended domain leads to arbitrary behaviour, including nontermination

In either case:

State your preconditions clearly!

As a guard, a comment or using QuickCheck:

```
P x ==> isDefined(f x)
```

where `isDefined y  =  y == y`.

Example `sumTo`

The sum from 0 to n = `n + (n-1) + (n-2) + ...  + 0`

```
sumTo :: Integer -> Integer
sumTo 0  =  0
sumTo n | n > 0  =
```

```
prop_sumTo n =
  n >= 0 ==> sumTo n == n*(n+1) `div` 2
```

Properties can be *conditional*

Typical recursion patterns for integers

```
f :: Integer -> ...
f 0  =  e                                -- base case
f n | n > 0  =  ... f(n - 1) ...         -- recursive call(s)
```

Always make the base case as simple as possible,
typically 0, not 1

Many variations:

- more parameters
- other base cases, e.g. `f 1`
- other recursive calls, e.g. `f(n - 2)`
- also for negative numbers

Recursion in general

- Reduce a problem to a *smaller* problem,
  e.g. `pow x n` to `pow x (n-1)`
- Must eventually reach a *base case*
- Build up solutions from smaller solutions

General problem solving strategy
in *any* programing language

2.6 Syntax matters

Functions are defined by one or more equations.
In the simplest case, each function is defined by one (possibly conditional) equation:

$$
\begin{array}{l}
f\ x_1\ \ldots\ x_n \\
\quad \mid\ test_1 \quad = \quad e_1 \\
\quad \vdots \\
\quad \mid\ test_n \quad = \quad e_n
\end{array}
$$

Each right-hand side e_i is an expression.
Note: `otherwise = True`

Function and parameter names must begin with a lower-case letter
(Type names begin with an upper-case letter)

An *expression* can be

- a *literal* like `0` or `"xyz"`,
- or an *identifier* like `True` or `x`,
- or a *function application* $f\ e_1\ \ldots\ e_n$
  where f is a function and $e_1\ \ldots\ e_n$ are expressions,
- or a parenthesized expression (e)

Additional syntactic sugar:

- `if then else`
- infix
- `where`
- ...

Local definitions: where

A defining equation can be followed by one or more local definitions.

```
pow4 x  =  x2 * x2 where x2 = x * x

pow4 x  =  sq (sq x) where sq x = x * x

pow8 x  =  sq (sq x2)
  where x2 = x * x
        sq y = y * y

myAbs x
  | x > 0       =  y
  | otherwise   = -y
  where y = x
```

Local definitions: `let`

`let` x `=` e_1 `in` e_2

defines x locally in e_2

Example:

```
let x = 2+3 in x^2 + 2*x
= 35
```

Like e_2 `where` x `=` e_1
But can occur anywhere in an expression
`where`: only after function definitions

Layout: the offside rule

```
a = 10
b = 20
c = 30
```

~~a = 10~~
~~ b = 20~~
~~c = 30~~

~~ a = 10~~
~~b = 20~~
~~ c = 30~~

In a sequence of definitions,
each definition must begin in the same column.

```
a = 10 +
  20
```

~~a = 10 +~~
~~20~~

~~ a = 10 +~~
~~20~~

A definition ends with the first piece of text
in or to the left of the start column.

Prefix and infix

Function application: `f a b`

Functions can be turned into infix operators by enclosing them in back quotes.

Example

```
5 `mod` 3  =  mod 5 3
```

Infix operators: `a + b`

Infix operators can be turned into functions by enclosing them in parentheses.

Example

```
(+) 1 2  =  1 + 2
```

Comments

Until the end of the line: `--`

```
id x = x    -- the identity function
```

A comment block: `{- ... -}`

```
{- Comments
   are
   important
-}
```

2.7 Types `Char` and `String`

Character literals as usual: `'a'`, `'$'`, `'\n'`, ...

Lots of predefined functions in module `Data.Char`

String literals as usual: `"I am a string"`

Strings are lists of characters.

Lists can be concatenated with `++`:

`"I am" ++ "a string"` = `"I ama string"`

More on lists later.

2.8 Tuple types

```
(True, 'a', "abc") :: (Bool, Char, String)
```

In general:

If	e_1 :: T_1 ... e_n :: T_n
then	(e_1,...,e_n) :: (T_1,...,T_n)

In mathematics: $T_1 \times \ldots \times T_n$

2.9 Do's and Don'ts

`True and False`

Never write

$$b \texttt{ == True}$$

Simply write

$$b$$

Never write

$$b \texttt{ == False}$$

Simply write

$$\texttt{not } b$$

```
isBig :: Integer -> Bool
```

~~isBig n~~
~~| n > 9999   = True~~
~~| otherwise = False~~

```
isBig n  =  n > 9999
```

~~if b then True else False~~ b

~~if b then False else True~~ not b

~~if b then True else b'~~ b || b'

...

Tuple

Try to avoid (mostly):

```
f (x,y)  =  ...
```

Usually better:

```
f x y  =  ...
```

Just fine:

```
f x y  =  (x + y, x - y)
```

3. Lists

List comprehension
Generic functions: Polymorphism
Case study: Pictures
Pattern matching
Recursion over lists

Lists are the most important data type
in functional programming

```
[1, 2, 3, -42] :: [Integer]

[False] :: [Bool]

['C', 'h', 'a', 'r'] :: [Char]
```
=
```
"Char" :: String
```
because
```
type String = [Char]

[not, not] ::

[] :: [T]
```
-- empty list for any type T
```
[[True],[]] ::
```

Typing rule

If $\quad e_1$ `::` $T \quad \dots \quad e_n$ `::` T
then $\quad$ `[`e_1`,...,`e_n`]` `::` `[`T`]`

Graphical notation:

$$\frac{e_1 :: T \quad \dots \quad e_n :: T}{[e_1, \dots, e_n] :: [T]}$$

`[True, 'c']` is not type-correct!!!

All elements in a list must have the same type

Test

```
(True, 'c') ::

[(True, 'c'), (False, 'd')] ::

([True, False], ['c', 'd']) ::
```

List ranges

```
[1 .. 3] = [1, 2, 3]
[3 .. 1] = []
['a' .. 'c'] = ['a', 'b', 'c']
```

Concatenation: ++

Concatenates two lists of the same type:

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

~~`[1, 2] ++ ['a']`~~

3.1 List comprehension

Set comprehensions:

$$\{x^2 \mid x \in \{1, 2, 3, 4, 5\}\}$$

The set of all x^2 such that x is an element of $\{1, 2, 3, 4, 5\}$

List comprehension:

```
[ x ^ 2 | x <- [1 .. 5]]
```

The list of all `x^2` *such that* `x` *is an element of* `[1 .. 5]`

List comprehension — Generators

```
[ x ^ 2 | x <- [1 .. 5]]
= [1, 4, 9, 16, 25]

[ toLower c | c <- "Hello, World!"]
= "hello, world!"

[ (x, even x) | x <- [1 .. 3]]
= [(1, False), (2, True), (3, False)]

[ x+y | (x,y) <- [(1,2), (3,4), (5,6)]]
= [3, 7, 11]
```

pattern <- *list expression*
is called a *generator*

Precise definition of *pattern* later.

List comprehension — Tests

```
[ x*x | x <- [1 .. 5], odd x]
= [1, 9, 25]

[ x*x | x <- [1 .. 5], odd x, x > 3]
= [25]

[ toLower c | c <- "Hello, World!", isAlpha c]
= "helloworld"
```

Boolean expressions are called *tests*

Defining functions by list comprehension

Example

```
factors :: Int -> [Int]
factors n  =  [m | m <- [1 .. n], n `mod` m == 0]
```

$\Longrightarrow$ `factors 15 = [1, 3, 5, 15]`

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

$\Longrightarrow$ `prime 15 = False`

```
primes :: Int -> [Int]
primes n  =  [p | p <- [1 .. n], prime p]
```

$\Longrightarrow$ `primes 100 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,`

List comprehension — General form

$$[\ expr \mid E_1,\ \ldots,\ E_n\ ]$$

where $expr$ is an expression and each E_i is a generator or a test

Multiple generators

```
[(i,j) | i <- [1 .. 2], j <- [7 .. 9]]
```

```
= [(1,7), (1,8), (1,9), (2,7), (2,8), (2,9)]
```

Analogy: each generator is a `for` loop:

```
for all i <- [1 .. 2]
  for all j <- [7 .. 9]
    ...
```

Key difference:

Loops *do* something
Expressions *produce* something

Dependent generators

```
[(i,j) | i <- [1 .. 3], j <- [i .. 3]]
= [(1,j) | j <- [1..3]] ++
  [(2,j) | j <- [2..3]] ++
  [(3,j) | j <- [3..3]]
= [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
```

The meaning of list comprehensions

$[e \mid x \texttt{ <- } [a_1,\ldots,a_n]]$
$= (\texttt{let } x = a_1 \texttt{ in } [e]) \texttt{ ++ } \cdots \texttt{ ++ } (\texttt{let } x = a_n \texttt{ in } [e])$

$[e \mid b]$
$= \texttt{if } b \texttt{ then } [e] \texttt{ else } []$

$[e \mid x \texttt{ <- } [a_1,\ldots,a_n],\ E_2,\ \ldots,\ E_n]$
$= (\texttt{let } x = a_1 \texttt{ in } [e \mid E_2,\ \ldots,\ E_n]) \texttt{ ++ } \cdots \texttt{ ++ }$
$\quad (\texttt{let } x = a_n \texttt{ in } [e \mid E_2,\ \ldots,\ E_n])$

$[e \mid b,\ E_2,\ \ldots,\ E_n]$
$= \texttt{if } b \texttt{ then } [e \mid E_2,\ \ldots,\ E_n] \texttt{ else } []$

Example: concat

```
concat xss  =  [x | xs <- xss, x <- xs]
```

```
concat [[1,2], [4,5,6]]
= [x | xs <- [[1,2], [4,5,6]], x <- xs]
= [x | x <- [1,2]] ++ [x | x <- [4,5,6]]
= [1,2] ++ [4,5,6]
= [1,2,4,5,6]
```

What is the type of `concat`?

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

3.2 Generic functions: Polymorphism

Polymorphism = one function can have many types

Example

```
length :: [Bool] -> Int
length :: [Char] -> Int
length :: [[Int]] -> Int
```

$\vdots$

The most general type:

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

where `a` is a *type variable*

$\Longrightarrow$ `length :: [`T`] -> Int` *for all types* T

Type variable syntax

Type variables must start with a lower-case letter

Typically: `a`, `b`, `c`, ...

Two kinds of polymorphism

Subtype polymorphism as in Java:

$$\frac{f :: T \to U \qquad T' \leq T}{f :: T' \to U}$$

(remember: horizontal line = implication)

Parametric polymorphism as in Haskell:

Types may contain type variables ("parameters")

$$\frac{f :: T}{f :: T[U/a]}$$

where $T[U/a]$ = "T with a replaced by U"
Example: $(a \to a)[Bool/a] \;=\; Bool \to Bool$

(Often called *ML-style polymorphism*)

Defining polymorphic functions

```
id :: a -> a
id x  =  x

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

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

silly :: Bool -> a -> Char
silly x y  =  if x then 'c' else 'd'

silly2 :: Bool -> Bool -> Bool
silly2 x y  =  if x then x else y
```

Quiz

```
f x y z = if x then y else z
```

```
f x y = [(x,y), (y,x)]
```

```
f x = [ length u + v | (u,v) <- x ]
```

```
f x y = [ u ++ x | u <- y, length u < x ]
```

```
f x y = [[ (u,v) | u <- w, u, v <- x] | w <- y]
```

Polymorphic list functions from the Prelude

```
length :: [a] -> Int
length [5, 1, 9] = 3

(++) :: [a] -> [a] -> [a]
[1, 2] ++ [3, 4] = [1, 2, 3, 4]

reverse :: [a] -> [a]
reverse [1, 2, 3] = [3, 2, 1]

replicate :: Int -> a -> [a]
replicate 3 'c' = "ccc"
```

Polymorphic list functions from the Prelude

```
head, last :: [a] -> a
head "list" = ’l’,    last "list" = ’t’

tail, init :: [a] -> [a]
tail "list" = "ist",    init "list" = "lis"

take, drop :: Int -> [a] -> [a]
take 3 "list" = "lis",    drop 3 "list" = "t"

-- A property:
prop_take_drop n xs  =
  take n xs ++ drop n xs == xs
```

Polymorphic list functions from the Prelude

```
concat :: [[a]] -> [a]
concat [[1, 2], [3, 4], [0]] = [1, 2, 3, 4, 0]

zip :: [a] -> [b] -> [(a,b)]
zip [1,2] "ab" = [(1, 'a'), (2, 'b')]

unzip :: [(a,b)] -> ([a],[b])
unzip [(1, 'a'), (2, 'b')] = ([1,2], "ab")

-- A property
prop_zip xs ys = length xs == length ys ==>
  unzip(zip xs ys) == (xs, ys)
```

Haskell libraries

- Prelude and much more
- Hoogle — searching the Haskell libraries
- Hackage — a collection of Haskell packages

See Haskell pages and Thompson's book for more information.

Further list functions from the Prelude

```
and :: [Bool] -> Bool
and [True, False, True] = False

or :: [Bool] -> Bool
or [True, False, True] = True

-- For numeric types a:
sum, product :: [a] -> a
sum [1, 2, 2] = 5,    product [1, 2, 2] = 4
```

What exactly is the type of `sum`, `prod`, `+`, `*`, `==`, . . . ???

Polymorphism versus Overloading

Polymorphism: one definition, many types

Overloading: different definition for different types

Example

Function `(+)` is overloaded:

- on type `Int`: built into the hardware
- on type `Integer`: realized in software

So what is the type of `(+)` ?

Numeric types

```
(+) :: Num a => a -> a -> a
```

Function `(+)` has type `a -> a -> a` for any type of class `Num`

- Class `Num` is the class of *numeric types*.
- Predefined numeric types: `Int`, `Integer`, `Float`
- Types of class Num offer the basic arithmetic operations:

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
⋮
sum, product :: Num a => [a] -> a
```

Other important type classes

- The class `Eq` of *equality types*, i.e. types that posess

```
(==) ::  Eq a => a -> a -> Bool
(/=) ::  Eq a => a -> a -> Bool
```

  Most types are of class `Eq`. Exception:

- The class `Ord` of *ordered types*, i.e. types that posess

```
(<)  ::  Ord a => a -> a -> Bool
(<=) ::  Ord a => a -> a -> Bool
```

More on type classes later.

Don't confuse with OO classes!

Warning: `==  []`

```
null xs  =  xs == []
```

Why?

`==` on `[a]` may call `==` on `a`

Better:

```
null :: [a] -> Bool
null []  =  True
null _   =  False
```

In Prelude!

Warning: QuickCheck and polymorphism

QuickCheck does not work well on polymorphic properties

Example

QuickCheck does not find a counterexample to

```
prop_reverse :: [a] -> Bool
prop_reverse xs  =  reverse xs == xs
```

The solution: specialize the polymorphic property, e.g.

```
prop_reverse :: [Int] -> Bool
prop_reverse xs  =  reverse xs == xs
```

Now QuickCheck works

Conditional properties have result type `Property`

Example

```
prop_rev10 :: [Int] -> Property
prop_rev10 xs =
  length xs <= 10 ==> reverse(reverse xs) == xs
```

3.3 Case study: Pictures

```
type Picture = [String]
```

```
uarr ::  Picture
uarr =
 ["  #  ",
  " ### ",
  "#####",
  "  #  ",
  "  #  "]
```

```
larr ::  Picture
larr =
 ["  #  ",
  " ##  ",
  "#####",
  " ##  ",
  "  #  "]
```

```
flipH :: Picture -> Picture
flipH  =  reverse

flipV :: Picture -> Picture
flipV pic  =  [ reverse line | line <- pic]

rarr :: Picture
rarr  =  flipV larr

darr :: Picture
darr  =  flipH uarr

above :: Picture -> Picture -> Picture
above  =  (++)

beside :: Picture -> Picture -> Picture
beside pic1 pic2 =  [ l1 ++ l2 | (l1,l2) <- zip pic1 pic2]
```

Pictures.hs

Chessboards

```
bSq  =  replicate 5 (replicate 5 '#')

wSq  =  replicate 5 (replicate 5 ' ')

alterH :: Picture -> Picture -> Int -> Picture
alterH pic1 pic2 1  =  pic1
alterH pic1 pic2 n  =  pic1 `beside` alterH pic2 pic1 (n-1)

alterV :: Picture -> Picture -> Int -> Picture
alterV pic1 pic2 1  =  pic1
alterV pic1 pic2 n  =  pic1 `above` alterV pic2 pic1 (n-1)

chessboard :: Int -> Picture
chessboard n  =  alterV bw wb n where
  bw = alterH bSq wSq n
  wb = alterH wSq bSq n
```

Exercise

Ensure that the lower left square of `chesboard n` is always black.

3.4 Pattern matching

Every list can be constructed from `[]`
by repeatedly adding an element at the front
with the "cons" operator `(:) :: a -> [a] -> [a]`

syntactic sugar	in reality
`[3]`	`3 : []`
`[2, 3]`	`2 : 3 : []`
`[1, 2, 3]`	`1 : 2 : 3 : []`
`[`x_1`, ..., `x_n`]`	x_1 `: ... :` x_n `: []`

Note: x `:` y `:` zs $=$ x `:` $(y$ `:` $zs)$
`(:)` associates to the right

$\Longrightarrow$

Every list is either

`[]` or of the form

$x : xs$ where

x is the *head* (first element, *Kopf*), and

xs is the *tail* (rest list, *Rumpf*)

`[]` and `(:)` are called *constructors*
because every list can be constructed uniquely from them.

$\Longrightarrow$

Every non-empty list can be decomposed uniquely into head and tail.

Therefore these definitions make sense:

```
head (x : xs)  =  x
tail (x : xs)  =  xs
```

`(++)` is not a constructor:

`[1,2,3]` is not uniquely constructable with `(++)`:

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

Therefore this definition does not make sense:

```
nonsense (xs ++ ys) = length xs - length ys
```

Patterns

Patterns are expressions
consisting only of constructors and variables.
No variable must occur twice in a pattern.

$\Longrightarrow$ Patterns allow unique decomposition = *pattern matching*.

A *pattern* can be

- a variable such as `x` or a wildcard `_` (underscore)
- a literal like `1`, `'a'`, `"xyz"`, ...
- a tuple `(`p_1`, ..., `p_n`)` where each p_i is a pattern
- a constructor pattern $C\ p_1\ \ldots\ p_n$
  where C is a constructor and each p_i is a pattern

Note: `True` and `False` are constructors, too!

Function definitions by pattern matching

Example

```
head :: [a] -> a
head (x : _)  =  x

tail :: [a] -> [a]
tail (_ : xs)  =  xs

null :: [a] -> Bool
null []       =  True
null (_ : _)  =  False
```

Function definitions by pattern matching

$$
\begin{aligned}
f\ \ pat_1 &\ =\ e_1 \\
&\vdots \\
f\ \ pat_n &\ =\ e_n
\end{aligned}
$$

If f has multiple arguments:

$$
\begin{aligned}
f\ \ pat_{11} \ldots\ pat_{1k} &\ =\ e_1 \\
&\vdots
\end{aligned}
$$

Conditional equations:

$$f\ \ patterns\ \ |\ \ condition\ \ =\ \ e$$

When f is called, the equations are tried in the given order

Function definitions by pattern matching

Example (contrived)

```
true12 :: [Bool] -> Bool
true12 (True : True : _)  =  True
true12 _  =  False

same12 :: Eq a => [a] -> [a] -> Bool
same12 (x : _) (_ : y : _)  =  x == y

asc3 :: Ord a => [a] -> Bool
asc3 (x : y : z : _)  =  x < y && y < z
asc3 (x : y : _)  =  x < y
asc3 _  =  True
```

3.5 Recursion over lists

Example

```
length []         =  0
length (_ : xs)   =  length xs + 1

reverse []        =  []
reverse (x : xs)  =  reverse xs ++ [x]

sum :: Num a => [a] -> a
sum []        =  0
sum (x : xs)  =  x + sum xs
```

Primitive recursion on lists:

$$f\ [\,] = base \qquad \texttt{-- base case}$$
$$f\ (x : xs) = rec \qquad \texttt{-- recursive case}$$

- *base*: no call of f
- *rec*: only call(s) $f\ xs$

f may have additional parameters.

Finding primitive recursive definitions

Example

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

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

Insertion sort

Example

```
inSort :: Ord a => [a] -> [a]
inSort []       =  []
inSort (x:xs)   =  ins x (inSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []  =  [x]
ins x (y:ys) | x <= y      =  x : y : ys
             | otherwise   =  y : ins x ys
```

Beyond primitive recursion: Complex patterns

Example

```
ascending :: Ord a => [a] -> Bool
ascending []  =  True
ascending [_]  =  True
ascending (x : y : zs)  =  x <= y && ascending (y : zs)
```

Beyond primitive recursion: Multiple arguments

Example

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

Alternative definition:

```
zip' [] []  =  []
zip' (x:xs) (y:ys)  =  (x,y) : zip' xs ys
```

`zip'` is undefined for lists of different length!

Beyond primitive recursion: Multiple arguments

Example

```
take :: Int -> [a] -> [a]
take 0 _   =  []
take _ []  =  []
take i (x:xs) | i>0  =  x : take (i-1) xs
```

General recursion: Quicksort

Example

```
quicksort :: Ord a => [a] -> [a]
quicksort []  =  []
quicksort (x:xs)  =
  quicksort below ++ [x] ++ quicksort above
    where
      below = [y | y <- xs, y <= x]
      above = [y | y <- xs, y > x]
```

Accumulating parameter

Idea: Result is accumulated in parameter and returned later

Example: list of all (maximal) ascending sublists in a list

`ups [3,0,2,3,2,4] = [[3], [0,2,3], [2,4]]`

```
ups :: Ord a => [a] -> [[a]]
ups xs = ups2 xs []

ups2 :: Ord a => [a] -> [a] -> [[a]]
-- 1st param: input list
-- 2nd param: partial ascending sublist (reversed)
ups2 (x:xs) (y:ys)
  | x >= y      =  ups2 xs (x:y:ys)
  | otherwise   =  reverse (y:ys) : ups2 (x:xs) []
ups2 (x:xs) []  =  ups2  xs  [x]
ups2 []     ys  =  [reverse ys]
```

How can we `quickCheck` the result of `ups`?

Warning

Accumulating parameters can increase efficiency

but tend to obfuscate the code

Avoid if possible

Identifiers of list type end in ‘s’:

`xs`, `ys`, `zs`, . . .

Example

```
even :: Int -> Bool
even n  =  n == 0 || n > 0 && odd (n-1) || odd (n+1)

odd :: Int -> Bool
odd n  =  n /= 0 && (n > 0 && even (n-1) || even (n+1))
```

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x

> f 3
```

Binding and bound occurrences

Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
```

Binding and bound occurrences
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

> f 3

Binding and bound occurrences
Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
```

Binding and bound occurrences

Scope of binding

Scoping by example

```
x = y + 5
y = x + 1 where x = 7
f y = y + x
```

```
> f 3
```

Binding and bound occurrences

Scope of binding

Scoping by example

Summary:

- Order of definitions is irrelevant
- Parameters and `where`-defs are local to each equation

4. Proofs

Proving properties
Definedness
Computation Induction
Interlude: Type inference/reconstruction

Aim

Guarentee functional (I/O) properties of software

- Testing can guarantee properties for some inputs.
- Mathematical proof can guarantee properties for all inputs.

QuickCheck is good, proof is better

Beware of bugs in the above code;
I have only proved it correct, not tried it.

Donald E. Knuth, 1977

4.1 Proving properties

What do we prove?

Equations `e1 = e2`

How do we prove them?

By using defining equations `f p = t`

A first, simple example

Remember:

```
        [] ++ ys = ys
    (x:xs) ++ ys = x : (xs ++ ys)
```

Proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])      -- by def of ++
= 1 : 2 : ([] ++ [])    -- by def of ++
= 1 : 2 : []            -- by def of ++
= 1 : ([] ++ 2:[])      -- by def of ++
= 1:[] ++ 2:[]          -- by def of ++
```

Observation: first used equations from left to right (ok), then from right to left (strange!)

A more natural proof of `[1,2] ++ [] = [1] ++ [2]`:

```
1:2:[] ++ []
= 1 : (2:[] ++ [])      -- by def of ++
= 1 : 2 : ([] ++ [])    -- by def of ++
= 1 : 2 : []            -- by def of ++

1:[] ++ 2:[]
= 1 : ([] ++ 2:[])      -- by def of ++
= 1 : 2 : []            -- by def of ++
```

Proofs of `e1 = e2` are often better presented as two reductions to some expression `e`:

```
e1 = ...   = e
e2 = ...   = e
```

Fact If an equation does not contain any variables, it can be proved by evaluating both sides separately and checking that the result is identical.

But how to prove equations with variables, for example *associativity* of ++:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Properties of recursive functions are proved by induction

Induction on natural numbers: see Diskrete Strukturen

Induction on lists: here and now

Structural induction on lists

To prove property `P(xs)` for all finite lists `xs`

Base case: Prove `P([])` and

Induction step: Prove `P(xs)` implies `P(x:xs)`

↑ *induction hypothesis* (*IH*)

↑ new variable `x`

One and the same fixed `xs`!

This is called *structural induction* on `xs`.
It is a special case of induction on the length of `xs`.

Example: associativity of ++

Lemma app_assoc: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`

Proof by structural induction on `xs`

Base case:

To show: `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`

```
 ([] ++ ys) ++ zs
 = ys ++ zs            -- by def of ++
 = [] ++ (ys ++ zs)    -- by def of ++
```

Induction step:

IH: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`

To show: `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`

```
 ((x:xs) ++ ys) ++ zs
 = (x : (xs ++ ys)) ++ zs  -- by def of ++
 = x : ((xs ++ ys) ++ zs)  -- by def of ++
 = x : (xs ++ (ys ++ zs))  -- by IH
 (x:xs) ++ (ys ++ zs)
 = x : (xs ++ (ys ++ zs))  -- by def of ++
```

Induction template

Lemma `P(xs)`
Proof by structural induction on `xs`

Base case:
To show: `P([])`

Proof of `P([])`

Induction step:
IH: `P(xs)`
To show: `P(x:xs)`

Proof of `P(x:xs)` using IH

Example: `length` of `++`

Lemma `length(xs ++ ys) = length xs + length ys`
Proof by structural induction on `xs`
Base case:
To show: `length ([] ++ ys) = length [] + length ys`

```
length ([] ++ ys)
= length ys              -- by def of ++
length [] + length ys
= 0 + length ys          -- by def of length
= length ys
```

Induction step:

IH: `length(xs ++ ys) = length xs + length ys`

To show: `length((x:xs)++ys) = length(x:xs) + length ys`

```
length((x:xs) ++ ys)
= length(x : (xs ++ ys))      -- by def of ++
= 1 + length(xs ++ ys)        -- by def of length
= 1 + length xs + length ys  -- by IH
length(x:xs) + length ys
= 1 + length xs + length ys  -- by def of length
```

Example: `reverse` of `++`

Lemma `reverse(xs ++ ys) = reverse ys ++ reverse xs`
Proof by structural induction on `xs`
Base case:
To show: `reverse ([] ++ ys) = reverse ys ++ reverse []`

```
reverse ([] ++ ys)
= reverse ys                  -- by def of ++
reverse ys ++ reverse []
= reverse ys ++ []            -- by def of reverse
= reverse ys                  -- by Lemma app_Nil2
```

Lemma app_Nil2: `xs ++ [] = xs`
Proof exercise

Induction step:

IH: `reverse(xs ++ ys) = reverse ys ++ reverse xs`

To show: `reverse((x:xs)++ys) = reverse ys ++ reverse(x:xs)`

```
reverse((x:xs) ++ ys)
= reverse(x : (xs ++ ys))              -- by def of ++
= reverse(xs ++ ys) ++ [x]             -- by def of reverse
= (reverse ys ++ reverse xs) ++ [x]  -- by IH
= reverse ys ++ (reverse xs ++ [x])  -- by Lemma app_assoc
```

```
reverse ys ++ reverse(x:xs)
= reverse ys ++ (reverse xs ++ [x])  -- by def of reverse
```

Proof heuristic

- Try QuickCheck
- Try to evaluate both sides to common term
- Try induction
  - Base case: reduce both sides to a common term using function defs and lemmas
  - Induction step: reduce both sides to a common term using function defs, IH and lemmas
- If base case or induction step fails: conjecture, prove and use new lemmas

Two further tricks

- Proof by cases
- Generalization

Example: proof by cases

```
rem x []  =  []
rem x (y:ys) | x==y        =  rem x ys
             | otherwise   =  y : rem x ys
```

Lemma `rem z (xs ++ ys) = rem z xs ++ rem z ys`

Proof by structural induction on `xs`

Base case:

To show: `rem z ([] ++ ys) = rem z [] ++ rem z ys`

```
 rem z ([] ++ ys)
 = rem z ys               -- by def of ++
 rem z [] ++ rem z ys
 = rem z ys               -- by def of rem and ++
```

```
rem x []   =  []
rem x (y:ys) | x==y        =  rem x ys
             | otherwise   =  y : rem x ys
```

Induction step:

IH: `rem z (xs ++ ys) = rem z xs ++ rem z ys`

To show: `rem z ((x:xs)++ys) = rem z (x:xs) ++ rem z ys`

Proof by cases:

Case `z == x`:

```
 rem z ((x:xs) ++ ys)
 = rem z (xs ++ ys)          -- by def of ++ and rem
 = rem z xs ++ rem z ys      -- by IH
 rem z (x:xs) ++ rem z ys
 = rem z xs ++ rem z ys      -- by def of rem
```

Case `z /= x`:

```
 rem z ((x:xs) ++ ys)
 = x : rem z (xs ++ ys)          -- by def of ++ and rem
 = x : (rem z xs ++ rem z ys)    -- by IH
 rem z (x:xs) ++ rem z ys
 = x : (rem z xs ++ rem z ys)    -- by def of rem and ++
```

Proof by cases

Works just as well for `if-then-else`, for example

```
rem x []  =  []
rem x (y:ys)  =  if x == y then rem x ys
                 else y : rem x ys
```

Inefficiency of `reverse`

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

reverse [1,2,3]
= reverse [2,3] ++ [1]
= (reverse [3] ++ [2]) ++ [1]
= ((reverse [] ++ [3]) ++ [2]) ++ [1]
= (([] ++ [3]) ++ [2]) ++ [1]
= ([3] ++ [2]) ++ [1]
= (3 : ([] ++ [2])) ++ [1]
= [3,2] ++ [1]
= 3 : ([2] ++ [1])
= 3 : (2 : ([] ++ [1]))
= [3,2,1]
```

An improvement: `itrev`

```
itrev :: [a] -> [a] -> [a]
itrev [] xs      = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

```
itrev [1,2,3] []
= itrev [2,3] [1]
= itrev [3] [2,1]
= itrev [] [3,2,1]
= [3,2,1]
```

Proof attempt

Lemma `itrev xs [] = reverse xs`

Proof by structural induction on `xs`

Induction step fails:

IH: `itrev xs [] = reverse xs`

To show: `itrev (x:xs) [] = reverse (x:xs)`

```
itrev (x:xs) []
= itrev xs [x]        -- by def of itrev
reverse (x:xs)
= reverse xs ++ [x]   -- by def of reverse
```

Problem: IH not applicable because too specialized: `[]`

Generalization

Lemma `itrev xs ys = reverse xs ++ ys`

Proof by structural induction on `xs`

Induction step:

IH: `itrev xs ys = reverse xs ++ ys`

To show: `itrev (x:xs) ys = reverse (x:xs) ++ ys`

```
itrev (x:xs) ys
= itrev xs (x:ys)              -- by def of itrev
= reverse xs ++ (x:ys)         -- by IH
reverse (x:xs) ++ ys
= (reverse xs ++ [x]) ++ ys   -- by def of reverse
= reverse xs ++ ([x] ++ ys)   -- by Lemma app_assoc
= reverse xs ++ (x:ys)         -- by def of ++
```

Note: IH is used with `x:ys` instead of `ys`

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

Justification: all variables are implicitly $\forall$-quantified, except for the induction variable.

4.2 Definedness

Simplifying assumption, implicit so far:

No undefined values

Two kinds of undefinedness:

`head []`	raises exception
`f x = f x + 1`	does not terminate

Undefinedness can be handled, too.
But it complicates life

What is the problem?

Many familiar laws no longer hold unconditionally:

```
x - x = 0
```

is true only if `x` is a defined value.

Two examples:

- Not true: `head [] - head [] = 0`
- From the nonterminating definition
  `f x = f x + 1`
  we could conclude that $0 = 1$.

Termination

Termination of a function means termination for all inputs.

Restriction:

The proof methods in this chapter assume that all recursive definitions under consideration terminate.

Most Haskell functions we have seen so far terminate.

How to prove termination

Example

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

terminates because ++ terminates and with each recursive call of `reverse`, the length of the argument becomes smaller.

A function `f :: T1 -> T` terminates
if there is a *measure function* `m :: T1 ->` $\mathbb{N}$ such that

- for every defining equation `f p = t`
- and for every recursive call `f r` in `t`: `m p > m r`.

Note:

- All primitive recursive functions terminate.
- `m` can be defined in Haskell or mathematics.
- The conditions above can be refined to take special Haskell features into account, eg sequential pattern matching.

More generally: `f :: T1 -> ... -> Tn -> T` terminates if there is a measure function `m :: T1 -> ... -> Tn -> ℕ` such that

- for every defining equation `f p1 ... pn = t`
- and for every recursive call `f r1 ... rn` in `t`:
  `m p1 ... pn > m r1 ... rn`.

Of course, all other functions that are called by `f` must also terminate.

Infinite values

Haskell allows infinite values, in particular infinite lists.

Example: `[1, 1, 1, ...]`

Infinite objects must be constructed by recursion:

```
ones = 1 : ones
```

Because we restrict to terminating definitions in this chapter, infinite values cannot arise.

Note:

- By termination of functions we really mean termination on *finite* values.
- For example `reverse` terminates only on finite lists.

This is fine because we can only construct finite values anyway.

How can infinite values be useful?
Because of “lazy evaluation”.
More later.

Exceptions

If we use arithmetic equations like `x - x = 0` unconditionally, we can "lose" exceptions:

`head xs - head xs = 0`
is only true if `xs /= []`

In such cases, we can prove equations `e1 = e2` that are only *partially correct*:

> If `e1` and `e2` do not produce a runtime exception
> then they evaluate to the same value.

Summary

- In this chapter everything must terminate
- This avoids undefined and infinite values
- This simplifies proofs

Notation

`P(e)`

means some property/formula `P` that contains the expression `e`.

Similarly: `P(e1,...,en)`

4.3 Computation Induction

Let `f` be a terminating function. (For simplicity: `f` is unary)
Every call `f e` leads to (0 or more) direct recursive calls
`f e1`, ..., `f en` where the `ei` are 'smaller' than `e`:
they are 1 step closer to termination.

Principle of *induction on the computation* of `f` (short: `f`-*induction*)
is an induction on the length of the computation.
To prove `P(x)` for all `x`, prove

`P(e)` is implied by the IHs `P(e1)`, ..., `P(en)`

for every defining equation

```
f e  =  ... f e1 ... f en ...
```

Note:

- `f`-induction is typically used to prove properties of `f`
- But it can be applied to prove arbitrary properties because the implication does not mention `f`

Example: `drop2`

```
drop2 [] = []
drop2 [x] = [x]
drop2 (x:y:xs) = x : drop2 xs
```

Principle of `drop2`-induction: To prove `P(xs)` (for all `xs`), prove

Case 1: `P([])`

Case 2: `P([x])` (`x` new variable)

Case 3: `P(xs)` $\Longrightarrow$ `P(x:y:xs)` (`x`,`y` new variables)

```
drop2 [] = [],    drop2 [x] = [x],
drop2 (x:y:xs) = x : drop2 xs
```

drop2-induction: To prove `P(xs)`
prove `P([])`, `P([x])` and `P(xs)` $\Longrightarrow$ `P (x:y:xs)`

Lemma `length(drop2 xs) = (length xs + 1) ‘div‘ 2`
`Proof` by `drop2`-induction on `xs`

Case 1:
To show: `length(drop2 []) = (length [] + 1) ‘div‘ 2`

```
drop2 [] = [],    drop2 [x] = [x],
drop2 (x:y:xs) = x : drop2 xs
```

drop2-induction: To prove `P(xs)`
prove `P([])`, `P([x])` `and` `P(xs)` $\Longrightarrow$ `P (x:y:xs)`

Lemma `length(drop2 xs) = (length xs + 1) ‘div‘ 2`
`Proof` by `drop2`-induction on `xs`

Case 2:
To show: `length(drop2 [x]) = (length [x] + 1) ‘div‘ 2`

```
drop2 [] = [],     drop2 [x] = [x],
drop2 (x:y:xs) = x : drop2 xs
```

drop2-induction: To prove `P(xs)`
prove `P([])`, `P([x])` and `P(xs)` $\Longrightarrow$ `P(x:y:xs)`

Lemma `length(drop2 xs) = (length xs + 1) ‘div‘ 2`
Proof by drop2-induction on `xs`

Case 3:
IH: `length(drop2 xs) = (length xs + 1) ‘div‘ 2`
To show:
`length(drop2(x:y:xs)) = (length(x:y:xs) + 1) ‘div‘ 2`

```
 length(drop2(x:y:xs))
 = length (drop2 xs) + 1          by def of drop2, length
 = (length xs + 1) ‘div‘ 2 + 1    by IH
 = (length xs + 3) ‘div‘ 2        by arithmetic
 = (length(x:y:xs) + 1) ‘div‘ 2   by def of drop2
```

Example: splice

```
splice [] ys = ys
splice (x:xs) ys = x : splice ys xs
```

Principle of `splice`-induction:
To prove `P(xs,ys)` (for all `xs` and `ys`), prove

Case 1: `P([],ys)`

Case 2: `P(ys,xs)` $\Longrightarrow$ `P(x:xs,ys)` (`x` new variable)

```
splice [] ys = ys
spice (x:xs) ys = x : splice ys xs
```

`splice`-induction: To prove `P(xs,ys)`
prove `P([],ys)` and `P(ys,xs)` $\Longrightarrow$ `P(x:xs,ys)`

Lemma `length(splice xs ys) = length xs + length ys`
`Proof` by `splice`-induction on `xs` and `ys`

Case 1:
To show: `length(splice [] ys) = length [] + length ys`

```
splice [] ys = ys
spice (x:xs) ys = x : splice ys xs
```

splice-induction: To prove `P(xs,ys)`
prove `P([],ys)` and `P(ys,xs)` $\Longrightarrow$ `P(x:xs,ys)`

Lemma `length(splice xs ys) = length xs + length ys`
Proof by splice-induction on `xs` and `ys`

Case 2:
IH: `length(splice ys xs) = length ys + length xs`
To show:
`length(splice (x:xs) ys) = length (x:xs) + length ys`

```
 length(splice (x:xs) ys)
 = length(x : splice ys xs)      by def of splice
 = length(splice ys xs) + 1      by def of length
 = (length ys + length xs) + 1   by IH
 = (length xs + 1) + length ys   by arithmetic
 = length (x:xs) + length ys     by def of length
```

Structural induction does not work for `splice`!

Computation induction: the full story

`f`-Induction:

To prove `P(x1,...,xk)` (for all `x1,...,xk`):
For every defining equation

```
f pat1 ...  patk  =  rhs
```

prove `P(pat1,...,patk)` (replace all `xi` by `pati` in `P`)
assuming the IHs `P(e1,...,ek)`
for every recursive call `f e1 ...  ek` in `rhs`.

If the recursive call occurs in the context of guards or conditions `b1`, ..., `bn` then all of them must become preconditions of the IH:

```
b1 && ...&& bn ==> P(e1,...,ek)
```

Example: `f pat | b1 = if b2 then f e1 else f e2`:

IH1: `b1 && b2 ==> P(e1)`

IH2: `b1 && not(b2) ==> P(e2)`

Computation induction: Requirements

Function `f` must terminate

Otherwise: `f x = f x`

The defining equations for `f` must cover all possible arguments

Otherwise: `f [] = 0`

4.4 Interlude: Type inference/reconstruction

How to infer/reconstruct the type of an expression
(and all subexpressions)

Given: an expression e

Type inference:

1. Give all variables and functions in e their most general type
2. From e set up a system of equations between types
3. Simplify the equations

Example: `concat (replicate x y)`

Initial type table:

```
x :: a
y :: b
replicate :: Int -> c -> [c]
concat :: [[d]] -> [d]
```

For each subexpression $f\ e_1 \ldots e_n$ generate n equations:

`a = Int`, `b = c`

`[c] = [[d]]`

Simplify equations: `[c] = [[d]]` ⇝ `c = [d]`

`b = c` ⇝ `b = [d]`

Solution to equation system: `a = Int`, `b = [d]`, `c = [d]`

Final type table:

```
x :: Int
y :: [d]
replicate :: Int -> [d] -> [[d]]
concat :: [[d]] -> [d]
```

Algorithm

1. Give the variables $x_1, \ldots, x_n$ in e the types $a_1, \ldots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \ldots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1,\ \ldots,\ \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable a by τ everywhere (if a does not occur in τ)
   - $T\ \sigma_1 \ldots \sigma_n = T\ \tau_1 \ldots \tau_n \ \leadsto\ \sigma_1 = \tau_1,\ \ldots,\ \sigma_n = \tau_n$ (where T is a type constructor, e.g. `[.]`, `.->.`, etc)
   - $a = T \ldots a \ldots$ or $T \ldots a \ldots = a$: type error!
   - $T \ldots = T' \ldots$ where $T \neq T'$: type error!

- For simple expressions you should be able to infer types "durch scharfes Hinsehen"
- Use the algorithm if you are unsure or the expression is complicated
- Or use the Haskell interpreter

5. Higher-Order Functions

- Applying functions to all elements of a list: `map`
- Filtering a list: `filter`
- Combining the elements of a list: `foldr`
- Lambda expressions
- Extensionality
- Curried functions
- More library functions
- Case study: Counting words

Recall [`Pic` is short for `Picture`]

```
alterH :: Pic -> Pic -> Int -> Pic
alterH pic1 pic2 1 = pic1
alterH pic1 pic2 n = beside pic1 (alterH pic2 pic1 (n-1))

alterV :: Pic -> Pic -> Int -> Pic
alterV pic1 pic2 1 = pic1
alterV pic1 pic2 n = above pic1 (alterV pic2 pic1 (n-1))
```

Very similar. Can we avoid duplication?

```
alt :: (Pic -> Pic -> Pic) -> Pic -> Pic -> Int -> Pic
alt f pic1 pic2 1 = pic1
alt f pic1 pic2 n = f pic1 (alt f pic2 pic1 (n-1))

alterH pic1 pic2 n = alt beside pic1 pic2 n

alterV pic1 pic2 n = alt above pic1 pic2 n
```

Higher-order functions:
Functions that take functions as arguments

```
... -> (... -> ...) -> ...
```

Higher-order functions capture patterns of computation

5.1 Applying functions to all elements of a list: `map`

Example

```
map even [1, 2, 3]
= [False, True, False]

map toLower "R2-D2"
= "r2-d2"

map reverse ["abc", "123"]
= ["cba", "321"]
```

What is the type of `map`?

```
map :: (a -> b) -> [a] -> [b]
```

`map`: The mother of all higher-order functions

Predefined in `Prelude`.
Two possible definitions:

```
map f xs  =  [ f x | x <- xs ]
```

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

Evaluating map

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

map sqr [1, -2]
= map sqr (1 : -2 : [])
= sqr 1 : map sqr (-2 : [])
= sqr 1 : sqr (-2) : (map sqr [])
= sqr 1 : sqr (-2) : []
= 1 : 4 : []
= [1, 4]
```

Some properties of `map`

```
length (map f xs) = length xs

map f (xs ++ ys) = map f xs ++ map f ys

map f (reverse xs) = reverse (map f xs)
```

Proofs by induction

QuickCheck and function variables

QuickCheck does not work automatically
for properties of function variables

It needs to know how to generate and print functions.

Cheap alternative: replace function variable by specific function(s)

Example

```
prop_map_even :: [Int] -> [Int] -> Bool
prop_map_even xs ys =
  map even (xs ++ ys) = map even xs ++ map even ys
```

5.2 Filtering a list: `filter`

Example

```
filter even [1, 2, 3]
= [2]

filter isAlpha "R2-D2"
= "RD"

filter null [[], [1,2], []]
= [[], []]
```

What is the type of `filter`?

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

filter

Predefined in Prelude.
Two possible definitions:

```
filter p xs  =  [ x | x <- xs, p x ]

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

Some properties of `filter`

True or false?

`filter p (xs ++ ys) = filter p xs ++ filter p ys`

`filter p (reverse xs) = reverse (filter p xs)`

`filter p (map f xs) = map f (filter p xs)`

Proofs by induction

5.3 Combining the elements of a list: `foldr`

Example

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

sum [x_1, ..., x_n] = x_1 + ... + x_n + 0

```
concat []        =  []
concat (xs:xss)  =  xs ++ concat xss
```

concat [xs_1, ..., xs_n] = xs_1 ++ ... ++ xs_n ++ []

$$\texttt{foldr } (\oplus)\ z\ [x_1,\ \ldots,\ x_n] = x_1 \oplus \ldots \oplus x_n \oplus z$$

Defined in Prelude:

```
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f a []        =  a
foldr f a (x:xs)    =  x `f` foldr f a xs
```

Applications:

```
sum xs  =  foldr (+) 0 xs

concat xss =  foldr (++) [] xss
```

What is the most general type of foldr?

foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x `f` foldr f a xs
```

`foldr f a` replaces `(:)` by `f` and `[]` by `a`

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x `f` foldr f a xs
```

```
foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + -2 + (foldr (+) 0 [])
= 1 + -2 + 0
= -1
```

More applications of `foldr`

```
product xs  =  foldr  (*)   1      xs
and xs      =  foldr  (&&)  True   xs
or xs       =  foldr  (||)  False  xs
inSort xs   =  foldr  ins   []     xs
```

Quiz

What is

```
foldr (:) ys xs
```

Example: `foldr (:) ys (1:2:3:[]) = 1:2:3:ys`

```
foldr (:) ys xs = ???
```

Proof by induction on `xs` (Exercise!)

Defining functions via `foldr`

- means you have understood the art of higher-order functions
- allows you to apply properties of `foldr`

Example

If `f` is associative and `a `f` x = x` then
`foldr f a (xs++ys) = foldr f a xs `f` foldr f a ys`.

Proof by induction on `xs`. Induction step:

```
foldr f a ((x:xs) ++ ys) = foldr f a (x : (xs++ys))
= x `f` foldr f a (xs++ys)
= x `f` (foldr f a xs `f` foldr f a ys)      -- by IH
foldr f a (x:xs) `f` foldr f a ys
= (x `f` foldr f a xs) `f` foldr f a ys
= x `f` (foldr f a xs `f` foldr f a ys)      -- by assoc.
```

Therefore, if `g xs = foldr f a xs`,
then `g (xs ++ ys) = g xs `f` g ys`.

Therefore `sum (xs++ys) = sum xs + sum ys`,
`product (xs++ys) = product xs * product ys`, ...

5.4 Lambda expressions

Consider

```
squares xs  =  map sqr xs  where  sqr x = x * x
```

Do we really need to define `sqr` explicitly? No!

```
\x -> x * x
```

is the anonymous function with

formal parameter `x` and result `x * x`

In mathematics: $x \mapsto x * x$

Evaluation:

```
(\x -> x * x) 3 = 3 * 3 = 9
```

Usage:

```
squares xs  =  map (\x -> x * x) xs
```

Terminology

$$(\backslash x \texttt{ -> } e_1)\ e_2$$

x: formal parameter
e_1: result
e_2: actual parameter

Why "lambda"?

The logician Alonzo Church invented *lambda calculus* in the 1930s

Logicians write $\lambda x.\ e$ instead of `\x ->` e

Typing lambda expressions

Example

`(\x -> x > 0) :: Int -> Bool`

because `x :: Int` implies `x > 0 :: Bool`

The general rule:

$$\texttt{(\textbackslash}x\ \texttt{->}\ e\texttt{)}\ \texttt{::}\ T_1\ \texttt{->}\ T_2$$

$$\text{if } x\ \texttt{::}\ T_1 \text{ implies } e\ \texttt{::}\ T_2$$

Evaluating lambda expressions

`(\x -> body) arg` = *body* with *x* replaced by *arg*

Example

```
(\xs -> xs ++ xs) [1] = [1] ++ [1]
```

Sections of infix operators

```
(+ 1)   means   (\x -> x + 1)
(2 *)   means   (\x -> 2 * x)
(2 ^)   means   (\x -> 2 ^ x)
(^ 2)   means   (\x -> x ^ 2)
```

etc

Example

```
squares xs  =  map (^ 2) xs
```

List comprehension

Just syntactic sugar for combinations of `map`

```
[ f x | x <- xs ]     =     map f xs
```

`filter`

```
[ x | x <- xs, p x ]     =     filter p xs
```

and `concat`

```
            [f x y | x <- xs, y <- ys] =
concat (                                  )
```

5.5 Extensionality

Two functions are equal
if for all arguments they yield the same result

f, g :: T_1 -> T:

$$\frac{\forall a.\ f\ a = g\ a}{f = g}$$

f, g :: T_1 -> T_2 -> T:

$$\frac{\forall a, b.\ f\ a\ b = g\ a\ b}{f = g}$$

5.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int        f :: Int -> (Int -> Int)
f x y  =  x+y                 f x  =  \y -> x+y
```

Both mean the same:

```
f a b                         (f a) b
= a + b                       = (\y -> a + y) b
                              = a + b
```

The trick: any function of two arguments
can be viewed as a function of the first argument
that returns a function of the second argument

In general

Every function is a function of one argument
(which may return a function as a result)

$$T_1 \text{ -> } T_2 \text{ -> } T$$

is just syntactic sugar for

$$T_1 \text{ -> } (T_2 \text{ -> } T)$$

$$f\ e_1\ e_2$$

is just syntactic sugar for

$$\underbrace{(f\ e_1)}_{::\,T_2 \text{ -> } T}\ e_2$$

Analogously for more arguments

`->` is not associative:

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } T) \neq (T_1 \texttt{ -> } T_2) \texttt{ -> } T$$

Example

```
f :: Int -> (Int -> Int)
f x y  =  x + y
```

```
g :: (Int -> Int) -> Int
g h  =  h 0 + 1
```

Application is not associative:

$$(f\ e_1)\ e_2 \neq f\ (e_1\ e_2)$$

Example

`(f 3) 4` $\neq$ `f (3 4)`

`g (id abs)` $\neq$ `(g id) abs`

Quiz

```
head tail xs
```

Correct?

Partial application

**Every function of n parameters
can be applied to less than n arguments**

Example

Instead of `sum xs = foldr (+) 0 xs`
just define `sum = foldr (+) 0`

In general:

If f `::` T_1 `->` ... `->` T_n `->` T
and a_1 `::` T_1, ..., a_m `::` T_m and $m \leq n$
then $f\ a_1 \ldots a_m$ `::` T_{m+1} `->` ... `->` T_n `->` T

5.7 More library functions

```
(.) :: (b -> c) -> (a -> b) ->
f . g  =  \x -> f (g x)
```

Example

```
head2  =  head . tail
```

```
head2 [1,2,3]
= (head . tail) [1,2,3]
= (\x -> head (tail x)) [1,2,3]
= head (tail [1,2,3])
= head [2,3]
= 2
```

```
const :: a -> (b -> a)
const x  =  \ _ -> x

curry :: ((a,b) -> c) -> (a -> b -> c)
curry f  =  \ x y -> f(x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f  =  \(x,y) -> f x y
```

```haskell
all :: (a -> Bool) -> [a] -> Bool
all p xs  =  and [p x | x <- xs]
```

Example

```
all (>1) [0, 1, 2]
= False
```

```haskell
any :: (a -> Bool) -> [a] -> Bool
any p  =  or [p x | x <- xs]
```

Example

```
any (>1) [0, 1, 2]
= True
```

```haskell
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []       = []
takeWhile p (x:xs)
    | p x            = x : takeWhile p xs
    | otherwise      = []
```

Example

```haskell
takeWhile (not . isSpace) "the end"
= "the"
```

```haskell
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p []       = []
dropWhile p (x:xs)
    | p x            = dropWhile p xs
    | otherwise      = x:xs
```

Example

```haskell
dropWhile (not . isSpace) "the end"
= " end"
```

5.8 Case study: Counting words

Input: A string, e.g. `"never say never again"`

Output: A string listing the words in alphabetical order, together with their frequency,
e.g. `"again: 1\nnever: 2\nsay: 1\n"`

Function `putStr` yields

```
again: 1
never: 2
say: 1
```

Design principle:

Solve problem in a sequence of small steps
transforming the input gradually into the output

Unix pipes!

Step 1: Break input into words

```
"never say never again"
```

function ↓ `words`

```
["never", "say", "never", "again"]
```

Predefined in Prelude

Step 2: Sort words

```
["never", "say", "never", "again"]
```

function ↓ `sort`

```
["again", "never", "never", "say"]
```

Predefined in `Data.List`

Step 3: Group equal words together

```
["again", "never", "never", "say"]
```

function ↓ `group`

```
[["again"], ["never", "never"], ["say"]]
```

Predefined in `Data.List`

Step 4: Count each group

```
[["again"], ["never", "never"], ["say"]]

    |  map (\ws -> (head ws, length ws)
    ↓

[("again", 1), ("never", 2), ("say", 1)]
```

Step 5: Format each group

```
[("again", 1), ("never", 2), ("say", 1)]
        |
        |  map (\(w,n) -> (w ++ ": " ++ show n)
        ↓
["again: 1", "never: 2", "say: 1"]
```

Step 6: Combine the lines

```
["again: 1", "never: 2", "say: 1"]
```

function ↓ `unlines`

```
"again: 1\nnever: 2\nsay: 1\n"
```

Predefined in Prelude

The solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
```

Merging `maps`

Can we merge two consecutive `maps`?

```
map f . map g = ???
```

The optimized solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\ws -> head ws ++ ": " ++ show(length ws))
  . group
  . sort
  . words
```

Proving `map f . map g = map (f.g)`

First we prove (why?)

```
map f (map g xs) = map (f.g) xs
```

by induction on `xs`:

- Base case:
  ```
  map f (map g []) = []
  map (f.g) [] = []
  ```
- Induction step:
  ```
  map f (map g (x:xs))
  = f (g x) : map f (map g xs)
  = f (g x) : map (f.g) xs      -- by IH
  map (f.g) (x:xs)
  = f (g x) : map (f.g) xs
  ```

$\Longrightarrow$ `(map f . map g) xs = map f (map g xs) = map (f.g) xs`

$\Longrightarrow$ `(map f . map g) = map (f.g)`   by extensionality

6. Type Classes

Remember: type classes enable overloading

Example

```
elem ::
elem x  =  any (== x)
```

where `Eq` is the class of all types with ==

In general:

Type classes are collections of types
that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces:
a set of function names with their types

Example

```
class Eq a where
  (==) :: a -> a -> Bool
```

Note: the type of `(==)` outside the class context is
`Eq a => a -> a -> Bool`

The general form of a class declaration:

```
class C a where
  f1 :: T1
  ...
  fn :: Tn
```

where the `Ti` may involve the type variable `a`

Type classes support generic programming:
Code that works not just for one type
but for a whole class of types,
all types that implement the functions of the class.

Instance

A type T is an instance of a class C
if T supports all the functions of C.
Then we write $C\ T$.

Example

Type `Int` is an instance of class `Eq`, i.e., `Eq Int`

Therefore `elem :: Int -> [Int] -> Bool`

Warning Terminology clash:
Type T_1 is an *instance* of type T_2
if T_1 is the result of replacing type variables in T_2.
For example `(Bool,Int)` is an instance of `(a,b)`.

`instance`

The `instance` statement makes a type an instance of a class.

Example

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False
```

Instances can be constrained:

Example

```
instance Eq a => Eq [a] where
  []     == []       =  True
  (x:xs) == (y:ys)   =  x == y && xs == ys
  _      == _        =  False
```

Possibly with multiple constraints:

Example

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2)  =  x1 == x2 && y1 == y2
```

The general form of the `instance` statement:

```
instance (context) => C T where
  definitions
```

T	is a type
context	is a list of assumptions $C_i\ T_i$
definitions	are definitions of the functions of class *C*

Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class `Ord` inherits all the operations of class `Eq`

Because `Bool` is already an instance of `Eq`,
we can now make it an instance of `Ord`:

```
instance Ord Bool where
  b1 <= b2  =  not b1 || b2
  b1 < b2  =  b1 <= b2 && not(b1 == b2)
```

From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y  =  not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y  =  x <= y && x /= y
  x > y  =  y < x
  x >= y =  y <= x

class Show a where
  show :: a -> String
```

7. Algebraic `data` Types

- `data` by example
- The general case
- Case study: boolean formulas
- Structural induction

So far: no really new types,
just compositions of existing types

Example: `type String = [Char]`

Now: `data` defines *new* types

Introduction by example: From enumerated types
to recursive and polymorphic types

7.1 `data` by example

Bool

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False

(&&) :: Bool -> Bool -> Bool
False && q  =  False
True  && q  =  q

(||) :: Bool -> Bool -> Bool
False || q  = q
True  || q  = True
```

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False

instance Show Bool where
  show True   =  "True"
  show False  =  "False"
```

Better: let Haskell write the code for you:

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

`deriving` supports many more classes: `Ord`, `Read`, ...

Warning

Do not forget to make your `data` types instances of `Show`

Otherwise Haskell cannot even print values of your type

Warning

QuickCheck does not automatically work for `data` types

You have to write your own test data generator. Later.

Season

```haskell
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq, Show)

next :: Season -> Season
next Spring = Summer
next Summer = Autumn
next Autumn = Winter
next Winter = Spring
```

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`
`Rect 0.9 1.1`
`Circle (-2.0)`

```
area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rect w h)  =  w * h
```

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []  =
lookup key ((x,y):xys)
  | key == x   =
  | otherwise  =
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type `Nat`:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
add Zero n   =  n
add (Suc m) n  =

mul :: Nat -> Nat -> Nat
mul Zero n   =  Zero
mul (Suc m) n  =
```

Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
           deriving Eq
```

The result of deriving `Eq`:

```
instance Eq a => Eq [a] where
  []     == []       =  True
  (x:xs) == (y:ys)   =  x == y && xs == ys
  _      == _        =  False
```

Defined explicitly:

```
instance Show a => Show [a] where
  show xs  =  "[" ++ concat cs ++ "]"
    where cs = Data.List.intersperse ", "  (map show xs)
```

Tree

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq, Show)
```

Some trees:

```haskell
Empty
Node 1 Empty Empty
Node 1 (Node 2 Empty Empty) Empty
Node 1 Empty (Node 2 Empty Empty)
Node 1 (Node 2 Empty Empty) (Node 3 Empty Empty)
⋮
```

```
-- assumption: < is a linear ordering
find :: Ord a => a -> Tree a -> Bool
find _ Empty  =  False
find x (Node a l r)
  | x < a  =  find x l
  | a < x  =  find x r
  | otherwise  =  True
```

```
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty  =  Node x Empty Empty
insert x (Node a l r)
  | x < a  =  Node a (insert x l) r
  | a < x  =  Node a l (insert x r)
  | otherwise  =  Node a l r
```

Example

```
insert 6 (Node 5 t1 (Node 7 Empty t2))
= Node 5 t1 (insert 6 (Node 7 Empty t2))
= Node 5 t1 (Node 7 (insert 6 Empty) t2)
= Node 5 t1 (Node 7 (Node 6 Empty Empty) t2)
```

`t1` and `t2` are not copied but shared!

`delete`?

QuickCheck for Tree

```
import Control.Monad
import Test.QuickCheck

-- for QuickCheck: test data generator for Trees
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree
    where
    tree 0 = return Empty
    tree n | n > 0 =
      oneof [return Empty,
             liftM3 Node arbitrary (tree (n `div` 2))
                                   (tree (n `div` 2))]
```

```
prop_find_insert :: Int -> Int -> Tree Int -> Bool
prop_find_insert x y t =
  find x (insert y t) == ???
```

Edit distance (see Thompson)

Problem: how to get from one word to another, with a *minimal* number of "edits".

Example: from `"fish"` to `"chips"`

Applications: DNA Analysis, Unix `diff` command

```haskell
data Edit = Change Char
          | Copy
          | Delete
          | Insert Char
          deriving (Eq, Show)

transform :: String -> String -> [Edit]

transform [] ys   =  map Insert ys
transform xs []   =  replicate (length xs) Delete
transform (x:xs) (y:ys)
  | x == y        =  Copy : transform xs ys
  | otherwise     =  best [Change y : transform xs ys,
                           Delete   : transform xs (y:ys),
                           Insert y : transform (x:xs) ys]
```

```
best :: [[Edit]] -> [Edit]
best [x]   = x
best (x:xs)
  | cost x <= cost b    = x
  | otherwise           = b
  where b = best xs

cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

Example: What is the edit distance
from `"trittin"` to `"tarantino"`?

```
transform "trittin" "tarantino" = ?
```

Complexity of `transform`: time $O(\qquad)$

The edit distance problem can be solved in time $O(mn)$
with *dynamic programming*

7.2 The general case

$$
\begin{array}{l}
\texttt{data}\ T\ a_1\ \ldots\ a_p\ \texttt{=} \\
\quad C_1\ t_{11}\ \ldots\ t_{1k_1}\ \ | \\
\quad \vdots \\
\quad C_n\ t_{n1}\ \ldots\ t_{nk_n}
\end{array}
$$

defines the *constructors*

$$
\begin{array}{l}
C_1\ \texttt{::}\ t_{11}\ \texttt{->}\ \ldots\ t_{1k_1}\ \texttt{->}\ T\ a_1\ \ldots\ a_p \\
\vdots \\
C_n\ \texttt{::}\ t_{n1}\ \texttt{->}\ \ldots\ t_{nk_n}\ \texttt{->}\ T\ a_1\ \ldots\ a_p
\end{array}
$$

Constructors are functions too!

Constructors can be used just like other functions

Example

```
map Just [1, 2, 3]  =  [Just 1, Just 2, Just 3]
```

But constructors can *also* occur in patterns!

Patterns revisited

Patterns are expressions that consist only of constructors and variables (which must not occur twice):
A *pattern* can be

- a variable (incl. `_`)
- a literal like `1`, `'a'`, `"xyz"`, ...
- a tuple `(`p_1`, ..., `p_n`)` where each p_i is a pattern
- a constructor pattern $C\ p_1\ \ldots\ p_n$ where
  C is a `data` constructor (incl. `True`, `False`, `[]` and `(:)`)
  and each p_i is a pattern

7.3 Case study: boolean formulas

```
type Name = String

data Form = F | T
          | Var Name
          | Not Form
          | And Form Form
          | Or Form Form
          deriving Eq
```

Example: `Or (Var "p") (Not(Var "p"))`

More readable: symbolic infix constructors, must start with `:`

```
data Form = F | T | Var Name
          | Not Form
          | Form :&: Form
          | Form :|: Form
          deriving Eq
```

Now: `Var "p" :|: Not(Var "p")`

```
par :: String -> String
par s = "(" ++ s ++ ")"

instance Show Form where
  show F = "F"
  show T = "T"
  show (Var x) = x
  show (Not p) = par("~" ++ show p)
  show (p :&: q) = par(show p ++ " & " ++ show q)
  show (p :|: q) = par(show p ++ " | " ++ show q)
```

```
> Var "p" :&: Not(Var "p")
(p & (~p))
```

Syntax versus meaning

`Form` is the *syntax* of boolean formulas, not their meaning:

`Not(Not T)` and `T` mean the same but are different:

```
Not(Not T) /= T
```

What is the meaning of a `Form`?

Its value!?

But what is the value of `Var "p"` ?

```
-- Wertebelegung
type Valuation = [(Name,Bool)]

eval :: Valuation -> Form -> Bool
eval _ F = False
eval _ T = True
eval v (Var x) = fromJust(lookup x v)
eval v (Not p) = not(eval v p)
eval v (p :&: q) = eval v p && eval v q
eval v (p :|: q) = eval v p || eval v q
```

```
> eval [("a",False), ("b",False)]
   (Not(Var "a") :&: Not(Var "b"))
True
```

All valuations for a given list of variable names:

```
vals :: [Name] -> [Valuation]
vals [] = [[]]
vals (x:xs) = [ (x,False):v | v <- vals xs ] ++
              [ (x,True):v  | v <- vals xs ]

vals ["b"]
= [("b",False):v | v <- vals []] ++
  [("b",True):v  | v <- vals []]
= [("b",False):[]] ++ [("b",True):[]]
= [[("b",False)], [("b",True)]]

vals ["a","b"]
= [("a",False):v | v <- vals ["b"]] ++
  [("a",True):v  | v <- vals ["b"]]
= [[("a",False),("b",False)], [("a",False),("b",True)]] ++
  [[("a",True), ("b",False)], [("a",True), ("b",True)]]
```

Does `vals` construct *all* valuations?

```
prop_vals1 xs =
  length(vals xs) ==  2 ^ length xs

prop_vals2 xs =
  distinct (vals xs)

distinct :: Eq a => [a] -> Bool
distinct [] = True
distinct (x:xs) = not(elem x xs) && distinct xs
```

Demo

Restrict size of test cases:

```
prop_vals1' xs =
  length xs <= 10 ==>
  length(vals xs) == 2 ^ length xs

prop_vals2' xs =
  length xs <= 10 ==> distinct (vals xs)
```

Demo

Satisfiable and tautology

```
satisfiable :: Form -> Bool
satisfiable p = or [eval v p | v <- vals(vars p)]

tautology :: Form -> Bool
tautology = not . satisfiable . Not

vars :: Form -> [Name]
vars F = []
vars T = []
vars (Var x) = [x]
vars (Not p) = vars p
vars (p :&: q) = nub (vars p ++ vars q)
vars (p :|: q) = nub (vars p ++ vars q)
```

```
p0 :: Form
p0 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))
```

```
> vals (vars p0)
[[("a",False),("b",False)], [("a",False),("b",True)],
 [("a",True), ("b",False)], [("a",True), ("b",True )]]
```

```
> [ eval v p0 | v <- vals (vars p0) ]
[True, False, False, True]
```

```
> satisfiable p0
True
```

Simplifying a formula: Not inside?

```
isSimple :: Form -> Bool
isSimple (Not p)     =  not (isOp p)
  where
  isOp (Not p)     =  True
  isOp (p :&: q)   =  True
  isOp (p :|: q)   =  True
  isOp p           =  False
isSimple (p :&: q)   =  isSimple p && isSimple q
isSimple (p :|: q)   =  isSimple p && isSimple q
isSimple p           =  True
```

Simplifying a formula: Not inside!

```
simplify :: Form -> Form
simplify (Not p)     =  pushNot (simplify p)
  where
  pushNot (Not p)     =  p
  pushNot (p :&: q)   =  pushNot p :|: pushNot q
  pushNot (p :|: q)   =  pushNot p :&: pushNot q
  pushNot p           =  Not p
simplify (p :&: q)   =  simplify p :&: simplify q
simplify (p :|: q)   =  simplify p :|: simplify q
simplify p           =  p
```

Quickcheck

```
-- for QuickCheck: test data generator for Form
instance Arbitrary Form where
  arbitrary = sized prop
    where
    prop 0  =
      oneof [return F,
             return T,
             liftM Var arbitrary]
    prop n | n > 0 =
      oneof
        [return F,
         return T,
         liftM Var arbitrary,
         liftM Not (prop (n-1)),
         liftM2 (:&:) (prop(n `div` 2)) (prop(n `div` 2)),
         liftM2 (:|:) (prop(n `div` 2)) (prop(n `div` 2))]
```

```
prop_simplify p  =  isSimple(simplify p)
```

7.4 Structural induction

Structural induction for `Tree`

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

To prove property `P(t)` for all finite `t :: Tree a`

Base case: Prove `P(Empty)` and

Induction step: Prove `P(Node x t1 t2)`
assuming the induction hypotheses `P(t1)` and `P(t2)`.
(`x`, `t1` and `t2` are new variables)

Example

```
flat :: Tree a -> [a]
flat Empty  =  []
flat (Node x t1 t2)  =
  flat t1 ++ [x] ++ flat t2

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Empty  =  Empty
mapTree f (Node x t1 t2)  =
  Node (f x) (mapTree f t1) (mapTree f t2)
```

Lemma `flat (mapTree f t) = map f (flat t)`

Proof by structural induction on `t`

Induction step:

IH1: `flat (mapTree f t1) = map f (flat t1)`
IH2: `flat (mapTree f t2) = map f (flat t2)`

To show: `flat (mapTree f (Node x t1 t2)) =`
`map f (flat (Node x t1 t2))`

```
flat (mapTree f (Node x t1 t2))
= flat (Node (f x) (mapTree f t1) (mapTree f t2))
= flat (mapTree f t1) ++ [f x] ++ flat (mapTree f t2)
= map f (flat t1) ++ [f x] ++ map f (flat t2)
    -- by IH1 and IH2
```

```
map f (flat (Node x t1 t2))
= map f (flat t1 ++ [x] ++ flat t2)
= map f (flat t1) ++ [f x] ++ map f (flat t2)
    -- by lemma distributivity of map over ++
```

Note: Base case and `-- by def of ...` omitted

The general (regular) case

```
data T a = ...
```

Assumption: `T` is a *regular* data type:

Each constructor C_i of `T` must have a type

t_1 `->` ... `->` t_{n_i} `-> T a`

such that each t_j is either `T a` or does not contain `T`

To prove property $P(t)$ for all finite t `:: T a`:

prove for each constructor C_i that $P(C_i\ x_1 \dots x_{n_i})$

assuming the induction hypotheses $P(x_j)$ for all j s.t. $t_j =$ `T a`

Example of non-regular type: `data T = C [T]`

8. I/O

File I/O

- So far, only batch programs:
  given the full input at the beginning,
  the full output is produced at the end
- Now, interactive programs:
  read input and write output
  while the program is running

The problem

- Haskell programs are pure mathematical functions:

  Haskell programs have no side effects

- Reading and writing are side effects:

  Interactive programs have side effects

An impure solution

Most languages allow functions to perform I/O
without reflecting it in their type.

Assume that Haskell were to provide an input function

```
inputInt :: Int
```

Now all functions potentially perform side effects.

Now we can no longer reason about Haskell like in mathematics:

```
inputInt - inputInt   =   0
inputInt + inputInt   =   2*inputInt
                     ...
```

are no longer true.

The pure solution

Haskell distinguishes expressions without side effects
from expressions with side effects (*actions*) by their type:

`IO a`

is the type of (I/O) actions that return a value of type `a`.

Example

`Char`: the type of pure expressions that return a `Char`

`IO Char`: the type of actions that return a `Char`

`IO ()`: the type of actions that return no result value

`()`

- Type `()` is the type of empty tuples (no fields).
- The only value of type `()` is `()`, the empty tuple.
- Therefore `IO ()` is the type of actions
  that return the dummy value `()`
  (because every action must return some value)

Basic actions

- `getChar :: IO Char`

  Reads a `Char` from standard input,
  echoes it to standard output,
  and returns it as the result

- `putChar :: Char -> IO ()`

  Writes a `Char` to standard output,
  and returns no result

- `return :: a -> IO a`

  Performs no action,
  just returns the given value as a result

Sequencing: do

A sequence of actions can be combined into a single action with the keyword do

Example

```
get2 :: IO (Char,Char)
get2 = do x <- getChar    -- result is named x
          getChar         -- result is ignored
          y <- getChar
          return (x,y)
```

General format (observe layout!):

$$\texttt{do}\ \ a_1$$
$$\vdots$$
$$a_n$$

where each a_i can be one of

- an action
  Effect: execute action
- x `<-` *action*
  Effect: execute *action* `::` `IO` a, give result the name $x :: a$
- `let` x `=` *expr*
  Effect: give *expr* the name x
  Lazy: *expr* is only evaluated when x is needed!

Derived primitives

Write a string to standard output:

```
putStr :: String -> IO ()
putStr []       =  return ()
putStr (c:cs)   =  do putChar c
                      putStr cs
```

Write a line to standard output:

```
putStrLn :: String -> IO ()
putStrLn cs  =  putStr (cs ++ "\n")
```

Read a line from standard input:

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

Actions are normal Haskell values and can be combined as usual, for example with `if-then-else`.

Example

Prompt for a string and display its length:

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

```
> strLen

Enter a string: abc
The string has 3 characters
```

How to read other types

Input string and convert

Useful class:

```
class Read a where
  read :: String -> a
```

Most predefined types are in class `Read`.

Example:

```
getInt :: IO Integer
getInt = do xs <- getLine
            return (read xs)
```

Case study

The game of Hangman
in file `hangman.hs`

```
main :: IO ()
main = do putStr "Input secret word: "
          word <- getWord ""
          clear_screen
          guess word
          main
```

```
guess :: String -> IO ()
guess word = loop "" "" gallows  where
  loop :: String -> String -> [String] -> IO()
  loop guessed missed gals =
    do let word' =
             map (\x -> if x `elem` guessed
                        then x else '-')
               word
       writeAt (1,1)
         (head gals ++ "\n" ++ "Word: " ++ word' ++
          "\nMissed: " ++ missed ++ "\n")
       if length gals == 1
       then putStrLn ("YOU ARE DEAD: " ++ word)
       else if word' == word then putStrLn "YOU WIN!"
       else do c <- getChar
               let ok = c `elem` word
               loop (if ok then c:guessed else guessed)
                    (if ok then missed else missed++[c])
                    (if ok then gals else tail gals)
```

Once IO, always IO

You cannot add I/O to a function without giving it an IO type

For example

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

Let us try to make sq print out some message:

```
sq x = do putStr("I am in sq!")
          return(x*x)
```

What is the type of sq now? `Int -> IO Int`
And this is what happens to cube:

```
cube x = do x2 <- sq x
            return(x * x2)
```

Haskell is a pure functional language
Functions that have side effects must show this in their type
I/O is a side effect

Separate I/O from processing to reduce `IO` creep:

```
main :: IO ()
main = do s <- getLine
          let r = process s
          putStrLn r
          main

process :: String -> String
process s = ...
```

8.1 File I/O

The simple way

- `type FilePath = String`
- `readFile :: FilePath -> IO String`

  Reads file contents *lazily*,
  only as much as is needed
- `writeFile :: FilePath -> String -> IO ()`

  Writes whole file
- `appendFile :: FilePath -> String -> IO ()`

  Appends string to file

```
import System.IO
```

Handles

`data Handle`

Opaque type, implementation dependent

> *Haskell defines operations to read and write characters from and to files, represented by values of type* `Handle`*. Each value of this type is a handle: a record used by the Haskell run-time system to manage I/O with file system objects.*

Files and handles

- ```
  data IOMode = ReadMode | WriteMode
              | AppendMode | ReadWriteMode
  ```
- `openFile :: FilePath -> IOMode -> IO Handle`

  Creates handle to file and opens file
- `hClose :: Handle -> IO ()`

  Closes file
  ```

By convention
all IO actions that take a handle argument begin with `h`

In `ReadMode`

- `hGetChar :: Handle -> IO Char`
- `hGetLine :: Handle -> IO String`
- `hGetContents :: Handle -> IO String`

  Reads the whole file *lazily*

In `WriteMode`

- `hPutChar :: Handle -> Char -> IO ()`
- `hPutStr :: Handle -> String -> IO ()`
- `hPutStrLn :: Handle -> String -> IO ()`
- `hPrint :: Show a => Handle -> a -> IO ()`

stdin and stdout

- `stdin :: Handle`
  `stdout :: Handle`
- `getChar = hGetChar stdin`
  `putChar = hPutChar stdout`

There is much more in the Standard IO Library
(including exception handling for IO actions)

Example (interactive cp: icp.hs)

```
main :: IO()
main =
  do fromH <- readOpenFile "Copy from: " ReadMode
     toH <- readOpenFile "Copy to: " WriteMode
     contents <- hGetContents fromH
     hPutStr toH contents
     hClose fromH
     hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
  do putStrLn prompt
     name <- getLine
     handle <- openFile name mode
     return handle
```

Executing `xyz.hs`

If `xyz.hs` contains a definition of `main`:

- `runhaskell xyz`
  or
- `ghc xyz` $\rightsquigarrow$ executable file `xyz`

9. Modules and Abstract Data Types

Modules
Abstract Data Types
Correctness

9.1 Modules

Module = collection of type, function, class etc definitions

Purposes:

- Grouping
- Interfaces
- Division of labour
- Name space management: `M.f` vs `f`
- Information hiding

GHC: one module per file

Recommendation: module `M` in file `M.hs`

Module header

```
module M where   -- M must start with capital letter
↑
```

All definitions must start in this column

- Exports everything defined in `M` (at the top level)

Selective export:

```
module M (T, f, ...)  where
```

- Exports only `T, f, ...`

Exporting data types

```
module M (T) where
data T = ...
```

- Exports only `T`, but not its constructors

```
module M (T(C,D,...)) where
data T = ...
```

- Exports `T` and its constructors `C`, `D`, ...

```
module M (T(..)) where
data T = ...
```

- Exports `T` and all of its constructors

Not permitted: `module M (T,C,D) where` (why?)

Exporting modules

By default, modules do not export names from imported modules

```
module B where
import A
...
```

```
module A where
f = ...
...
```

$\Longrightarrow$ B does not export f

Unless the names are mentioned in the export list

```
module B (f) where
import A
...
```

Or the whole module is exported

```
module B (module A) where
import A
...
```

import

By default, everything that is exported is imported

```
module B where
import A
...
```

```
module A where
f = ...
g = ...
```

⟹ B imports `f` and `g`

Unless an import list is specified

```
module B where
import A (f)
...
```

⟹ B imports only `f`

Or specific names are hidden

```
module B where
import A hiding (g)
...
```

`qualified`

```
import A
import B
import C
... f ...
```

Where does `f` come from??

Clearer: *qualified names*

```
... A.f ...
```

Can be enforced:

```
import qualified A
```

$\Longrightarrow$ must always write `A.f`

Renaming modules

```
import TotallyAwesomeModule

... TotallyAwesomeModule.f ...
```

Painful

More readable:

```
import qualified TotallyAwesomeModule as TAM

... TAM.f ...
```

For the full description of the module system
see the Haskell report

9.2 Abstract Data Types

Abstract Data Types do not expose their internal representation

Why? Example: sets implemented as lists without duplicates

- Could create illegal value: `[1, 1]`
- Could distinguish what should be indistinguishable:
  `[1, 2] /= [2, 1]`
- Cannot easily change representation later

Example: Sets

```
module Set where
-- sets are represented as lists w/o duplicates
type Set a = [a]
empty  :: Set a
empty  =  []
insert :: a -> Set a -> Set a
insert x xs  =  ...
isin :: a -> Set a -> Set a
isin x xs  =  ...
size :: Set a -> Integer
size xs  =  ...
```

Exposes everything

Allows nonsense like `Set.size [1,1]`

Better

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty  :: Set a
insert :: Eq a => a -> Set a -> Set a
isin   :: Eq a => a -> Set a -> Bool
size   :: Set a -> Int
-- Implementation
type Set a = [a]
...
```

- Explicit export list/interface
- But representation still not hidden
  Does not help: hiding the type name Set

Hiding the representation

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = S [a]

empty = S []
insert x (S xs) = S(if elem x xs then xs else x:xs)
isin x (S xs) = elem x xs
size (S xs) = length xs
```

Cannot construct values of type Set outside of module Set except through interface because S is not exported

```
Test.hs:3:11:  Not in scope:  data constructor ‘S’
```

Uniform naming convention: S $\rightsquigarrow$ Set

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
data Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Which `Set` is exported?

Slightly more efficient: newtype

```
module Set (Set, empty, insert, isin, size) where
-- Interface
...
-- Implementation
newtype Set a = Set [a]

empty = Set []
insert x (Set xs) = Set(if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

Conceptual insight

Data representation can be hidden
by wrapping data up in a constructor that is not exported

What if Set is a hidden local data type?

```
module SetByTree (Set, empty, insert, isin, size) where

-- Interface
empty  :: Set a
insert :: Ord a => a -> Set a -> Set a
isin   :: Ord a => a -> Set a -> Bool
size   :: Set a -> Integer

-- Implementation
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
```

No need for `newtype`:
The representation of `Tree` is hidden
as long as its constructors are hidden

Beware of ==

```
module SetByTree (Set, empty, insert, isin, size) where
...
type Set a = Tree a
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
...
```

Class instances are automatically exported and cannot be hidden

Client module:

```
import SetByTree
... insert 2 (insert 1 empty) ==
    insert 1 (insert 2 empty)
...
```

Result is probably `False` — representation is partly exposed!

The proper treatment of ==

Some alternatives:

- Do not make `Tree` an instance of `Eq`
- Hide representation:

```
-- do not export constructor Set:
newtype Set a = Set (Tree a)
data Tree a = Empty | Node a (Tree a) (Tree a)
              deriving (Eq)
```

- Define the right == on `Tree`:

```
instance Eq a => Eq(Tree a) where
  t1 == t2  =  inorder t1 == inorder t2
    where
    inorder Empty = []
    inorder (Node x t1 t2) =
      inorder t1 ++ [x] ++ inorder t2
```

Similar for all class instances,
not just `Eq`

9.3 Correctness

Why is module `Set` a correct implementation of (finite) sets?

Because	`empty`	simulates	$\{\}$
and	`insert _ _`	simulates	$\{\_\} \cup \_$
and	`isin _ _`	simulates	$\_ \in \_$
and	`size _`	simulates	$\|\_\|$

Each concrete operation on the implementation type of lists
simulates its abstract counterpart on sets

NB: We relate Haskell to mathematics

For uniformity we write $\{\mathtt{a}\}$ for the type of finite sets over type `a`

From lists to sets

Each list $[x_1, \ldots, x_n]$ represents the set $\{x_1, \ldots, x_n\}$.

Abstraction function α `:: [a] ->` $\{$`a`$\}$

$$\alpha[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$$

In Haskell style: α `[] =` $\{\}$

α `(x:xs) =` $\{$`x`$\} \cup \alpha$ `xs`

What does it mean that "lists simulate (implement) sets":

α (concrete operation)	=	abstract operation
α `empty`	=	$\{\}$
α `(insert x xs)`	=	$\{$`x`$\} \cup \alpha$ `xs`
`isin x xs`	=	`x` $\in \alpha$ `xs`
`size xs`	=	$\lvert\alpha$ `xs`$\rvert$

For the mathematically enclined:
α must be a homomorphism

Implementation I: lists with duplicates

```
empty          =  []
insert x xs    =  x : xs
isin x xs      =  elem x xs
size xs        =  length(nub xs)
```

The simulation requirements:

$$\begin{aligned} \alpha\ \texttt{empty} &= \{\} \\ \alpha\ (\texttt{insert x xs}) &= \{\texttt{x}\} \cup \alpha\ \texttt{xs} \\ \texttt{isin x xs} &= \texttt{x} \in \alpha\ \texttt{xs} \\ \texttt{size xs} &= |\alpha\ \texttt{xs}| \end{aligned}$$

Two proofs immediate, two need lemmas proved by induction

Implementation II: lists without duplicates

```
empty          =  []
insert x xs    =  if elem x xs then xs else x:xs
isin x xs      =  elem x xs
size xs        =  length xs
```

The simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

$$\texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$$

$$\texttt{size xs} = |\alpha\ \texttt{xs}|$$

Needs *invariant* that `xs` contains no duplicates

```
invar :: [a] -> Bool
invar []       =  True
invar (x:xs)   =  not(elem x xs) && invar xs
```

Implementation II: lists without duplicates

```
empty        =  []
insert x xs  =  if elem x xs then xs else x:xs
isin x xs    =  elem x xs
size xs      =  length xs
```

Revised simulation requirements:

$$\alpha\ \texttt{empty} = \{\}$$

$$\texttt{invar xs} \Longrightarrow \alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$$

$$\texttt{invar xs} \Longrightarrow \texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$$

$$\texttt{invar xs} \Longrightarrow \texttt{size xs} = |\alpha\ \texttt{xs}|$$

Proofs omitted. Anything else?

`invar` must be invariant!

In an imperative context:

If `invar` is true before an operation,
it must also be true after the operation

In a functional context:

If `invar` is true for the arguments of an operation,
it must also be true for the result of the operation

`invar` is *preserved* by every operation

```
             invar empty
invar xs ⟹  invar (insert x xs)
```

Proofs do not even need induction

Implementation III: BST

```
module SetByTree (Set, empty, insert, isin, size) where

-- Interface: see earlier slide

-- Implementation
type Set a = Tree a
datatype Tree a = Empty | Node a (Tree a) (Tree a)

empty  =  Empty

insert x Empty  =  Node x Empty Empty
insert x (Node a l r)
  | x < a  =  Node a (insert x l) r
  | a < x  =  Node a l (insert x r)
  | otherwise  =  Node a l r
```

Implementation III: BST

```
isin _ Empty   =  False
isin x (Node a l r)
  | x < a   =  isin x l
  | a < x   =  isin x r
  | otherwise  =  True

size Empty = 0
size (Node _ l r) = size l + size r + 1
```

Abstraction function

$$\alpha :: \texttt{Tree a -> } \{\texttt{a}\}$$
$$\alpha\ \texttt{Empty} = \{\}$$
$$\alpha\ \texttt{(Node a l r)} = \{\texttt{a}\} \cup \alpha\ \texttt{l} \cup \alpha\ \texttt{r}$$

Simulation requirements: as before, with invariant!

Invariant

```
bst :: Tree a -> Bool
```

The pedestrian way:

```
bst Empty   =  True
bst (Node a l r)   =  isLess l a && isGr a r &&
                      bst l && bst r

isLess Empty _   =  True
isLess (Node x l r) a = x < a && isLess l a && isLess r a

isGr a Empty   =  True
isGr a (Node x l r)   =  a > x && isGr a l && isGr a r
```

Invariant

```
bst :: Tree a -> Bool
```

The higher-order way:

```
bst Empty  =  True
bst (Node a l r)  =  all (\x -> x < a) l &&
                     all (\x -> a < x) r &&
                     bst l && bst r

all :: (a -> Bool) -> Tree a -> Bool
all p Empty  =  True
all p (Node x l r) = p x && all p l && all p r
```

Invariant

```
bst :: Tree a -> Bool
```

The list way:

```
bst t  =  sorted (inorder t)

inorder :: Tree a -> [a]
...

sorted :: Ord a => [a] -> Bool
...
```

Summary

Let C and A be two modules that have the same interface:

a type T and a set of functions F

To prove that C is a correct implementation of A define

an *abstraction function* α `::` $C.T$ `->` $A.T$

and an *invariant* `invar :: ` $C.T$ `-> Bool`

and prove for each $f \in F$:

- `invar` is invariant:

$$\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies \texttt{invar}\ (C.f\ x_1\ \ldots\ x_n)$$

  (where `invar` is `True` on types other than $C.T$)

- $C.f$ simulates $A.f$:

$$\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies$$
$$\alpha(C.f\ x_1\ \ldots\ x_n) = A.f\ (\alpha\ x_1)\ \ldots\ (\alpha\ x_n)$$

  (where α is the identity on types other than $C.T$)

10. Case Study: Two Efficient Algorithms

This lecture covers two classic efficient algorithms in functional style on the blackboard:

Huffman Coding
: See the Haskell book by Thompson for a detailed exposition.

Skew Heaps
: See the original paper for an imperative presentation and the derivation of the amortized complexity:

  Daniel Sleator and Robert Tarjan. Self-adjusting heaps. *SIAM Journal on Computing* 15(1):52–69, 1986.

The Haskell source files are on the course web page.

Huffman Coding

- Aim: encode text with as few bits as possible.
  Lossless compression, not encryption.
- Method: each character is mapped to a bit list.
  (Length of bit list depends on frequency of character.)

Example

$\mathtt{e} \mapsto 0$, $\mathtt{m} \mapsto 10$, $\mathtt{n} \mapsto 11$

$\Longrightarrow \mathtt{enem} \mapsto 011010$ (which is uniquely decodable)

Strings are encoded character by character

Prefix-free Codes

Definition

- A code is a mapping from characters to bit lists.
- A code is uniquely decodable if every bit list is the image of at most one string.
- A code is prefix-free if for no two different characters x and y the code for x is a prefix of the code for y.

Example

$\texttt{a} \mapsto 1$, $\texttt{b} \mapsto 11$
Not prefix free and not uniquely decodable: $\texttt{aa} \mapsto 11$ and $\texttt{b} \mapsto 11$.

Fact Prefix-free codes are uniquely decodable.

We are only interested in prefix-free codes.

Decoding

A prefix-free code can be represented as a binary tree.

Example

$\texttt{e} \mapsto 0$, $\texttt{m} \mapsto 10$, $\texttt{n} \mapsto 11$

```
      •
  0/     \1
   e       •
       0/     \1
        m       n
```

Huffman's Algorithm

Constructs an optimal code (tree) for a given frequency table based on the string to be encoded.

Example

String: `"go go gopher"`
Table: `[('g',3), ('o',3), (' ',2), ('p',1), ...]`

A code `t` is optimal for a string `cs` if for all codes `t'`:

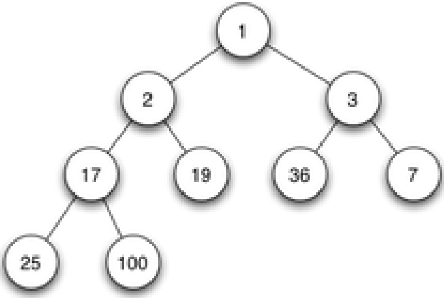
```
length (encode t cs) <= length (encode t' cs)
```

Key algorithmic ideas:

- Construct code tree bottom up
- Work on list of trees
- Always combine the "least frequent" trees into a new tree

Skew Heap

Implementation of *priority queue* as a *heap*, i.e., a binary tree where every chid is larger than the parent:



11. Lazy evaluation

Applications of lazy evaluation

Infinite lists

Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called

lazy evaluation („verzögerte Auswertung")

Advantages:

- Avoids unnecessary evaluations
- Terminates as often as possible
- Supports infinite lists
- Increases modularity

Therefore Haskell is called a *lazy functional language*.
Haskell is the only mainstream lazy functional language.

Evaluating expressions

Expressions are evaluated (*reduced*) by successively applying definitions until no further reduction is possible.

Example:

```
sq :: Integer -> Integer
sq n  =  n * n
```

One evaluation:

```
sq(3+4)   =   sq 7   =   7 * 7   =   49
```

Another evaluation:

```
sq(3+4)   =   (3+4) * (3+4)   =   7 * (3+4)   =   7 * 7   =   49
```

Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example

Let `n` have value 0 initially.

Two evaluations:

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

Reduction strategies

An expression may have many reducible subexpressions:

`sq (3+4)`

Terminology: *redex* = **red**ucible **ex**pression

Two common reduction strategies:

Innermost reduction Always reduce an innermost redex.
Corresponds to *call by value*:
Arguments are evaluated
before they are substituted into the function body

```
sq (3+4)  =  sq 7  =  7 * 7
```

Outermost reduction Always reduce an outermost redex.
Corresponds to *call by name:*
The unevaluated arguments
are substituted into the the function body

```
sq (3+4)  =  (3+4) * (3+4)
```

Comparison: termination

Definition:

```
loop = tail loop
```

Innermost reduction:

```
fst (1,loop)  =  fst(1,tail loop)
              =  fst(1,tail(tail loop))
              =  ...
```

Outermost reduction:

```
fst (1,loop)  =  1
```

Theorem If expression `e` has a terminating reduction sequence, then outermost reduction of `e` also terminates.

Outermost reduction terminates as often as possible

Why is this useful?

Example

Can build your own control constructs:

```
switch :: Int -> a -> a -> a
switch n x y
  | n > 0       =  x
  | otherwise   =  y

fac :: Int -> Int
fac n = switch n (n * fac(n-1)) 1
```

Comparison: Number of steps

Innermost reduction:

```
sq (3+4) =  sq 7  =  7 * 7  =  49
```

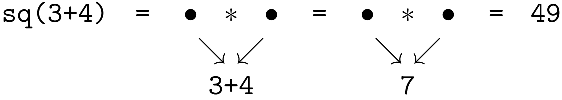
Outermost reduction:

```
sq(3+4)  =  (3+4)*(3+4)  =  7*(3+4)  =  7*7  =  49
```

More outermost than innermost steps!

How can outermost reduction be improved?

Sharing!



The expression 3+4 is only evaluated *once!*

Lazy evaluation := outermost reduction + sharing

Theorem

Lazy evaluation never needs more steps than innermost reduction.

The principles of lazy evaluation:

- Arguments of functions are evaluated only
  if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully,
  but only far enough to evaluate the function.
  (Remember `fst (1,loop)`)
- Each argument is evaluated at most once (sharing!)

Lazy evaluation does not require sharing of all common subexpressions!

No shared evaluation: `(3+4) * (3+4)`

Only if the same variable occurs multiple times.
Example: `let x = 3+4 in x*x`

Pattern matching

Example

```
f :: [Int] -> [Int] -> Int
f []     ys     =  0
f (x:xs) []     =  0
f (x:xs) (y:ys) =  x+y
```

Lazy evaluation:

```
f [1..3] [7..9]                   -- does f.1 match?
=  f (1 : [2..3]) [7..9]          -- does f.2 match?
=  f (1 : [2..3]) (7 : [8..9])    -- does f.3 match?
=  1+7
=  8
```

Example

```
f m n p | m >= n && m >= p  =  m
        | n >= m && n >= p  =  n
        | otherwise         =  p
```

Lazy evaluation:

```
f (2+3) (4-1) (3+9)
  ?  2+3 >= 4-1 && 2+3 >= 3+9
  ?  =  5 >= 3 && 5 >= 3+9
  ?  =  True   && 5 >= 3+9
  ?  =  5 >= 3+9
  ?  =  5 >= 12
  ?  =  False
  ?  3 >= 5 && 3 >= 12
  ?  =  False && 3 >= 12
  ?  =  False
  ?  otherwise = True
=  12
```

`where` and `let`

Same principle: definitions in `where`/`let` clauses are only evaluated when needed and only as much as needed.

Lambda

Haskell never reduces inside a lambda

Example: `\x -> False && x` cannot be reduced

Reasons:

- Functions are black boxes
- All you can do with a function is apply it

Example:

```
(\x -> False && x) True  =  False && True  =  False
```

Built-in functions

Arithmetic operators and other built-in functions
evaluate their arguments first

Example

`3 * 5` is a redex

`0 * head (...)` is not a redex

Predefined functions from Prelude

They behave like their Haskell definition:

```
(&&) :: Bool -> Bool -> Bool
True  && y  =  y
False && y  =  False
```

Slogan

Lazy evaluation evaluates an expression only when needed
and only as much as needed.

(*"Call by need"*)

11.1 Applications of lazy evaluation

Minimum of a list

```
min   =  head . inSort

inSort :: Ord a => [a] -> [a]
inSort []       =  []
inSort (x:xs)   =  ins x (inSort xs)

ins :: Ord a => a -> [a] -> [a]
ins x []   =  [x]
ins x (y:ys) | x <= y      =  x : y : ys
             | otherwise   =  y : ins x ys
```

$\Longrightarrow$ `inSort [6,1,7,5]`

```
   =  ins 6 (ins 1 (ins 7 (ins 5 [])))
```

```
min [6,1,7,5]  =  head(inSort [6,1,7,5])
=  head(ins 6 (ins 1 (ins 7 (ins 5 [])))))
=  head(ins 6 (ins 1 (ins 7 (5 : [])))))
=  head(ins 6 (ins 1 (5 : ins 7 [])))
=  head(ins 6 (1 : 5 : ins 7 []))
=  head(1 : ins 6 (5 : ins 7 [])))
=  1
```

Lazy evaluation needs only linear time
although `inSort` is quadratic
because the sorted list is never constructed completely

Warning: this depends on the exact algorithm and does not work so nicely with all sorting functions!

Maximum of a list

```
max  =  last . inSort
```

Complexity?

Takeuchi Function

```
t :: Int -> Int -> Int -> Int
t x y z | x <= y    = y
        | otherwise = t (t (x-1) y z)
                        (t (y-1) z x)
                        (t (z-1) x y)
```

In C:

```
int t(int x, int y, int z) {
  if (x <= y)
    return y;
  else
    return t(t(x-1, y, z), t(y-1, z, x), t(z-1, x, y));
}
```

Try `t 15 10 0` — Haskell beats C!

11.2 Infinite lists

Example

A recursive definition

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

that defines an infinite list of 1s:

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

What GHCi has to say about it:

```
> ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
```

Haskell lists can be finite or infinite

Printing an infinite list does not terminate

But Haskell can compute with infinite lists, thanks to lazy evaluation:

```
> head ones
1
```

Remember:

Lazy evaluation evaluates an expression only as much as needed

Outermost reduction: `head ones   =   head (1 : ones)   =   1`

Innermost reduction:

```
head ones
=  head (1 : ones)
=  head (1 : 1 : ones)
=  ...
```

Haskell lists are never actually infinite but only potentially infinite

Lazy evaluation computes as much of the infinite list as needed

This is how partially evaluated lists are represented internally:

`1 : 2 : 3 :` | code pointer to compute rest |

In general: finite prefix followed by code pointer

Why (potentially) infinite lists?

- They come for free with lazy evaluation
- They increase modularity:
  list producer does not need to know
  how much of the list the consumer wants

Example: The sieve of Eratosthenes

1. Create the list 2, 3, 4, ...
2. Output the first value p in the list as a prime.
3. Delete all multiples of p from the list
4. Goto step 2

2 3 4 5 6 7 8 9 10 11 12 ...
2 3 5 7 11 ...

In Haskell:

```
primes :: [Int]
primes  =  sieve [2..]

sieve :: [Int] -> [Int]
sieve (p:xs)  =  p : sieve [x | x <- xs, x `mod` p /= 0]
```

Lazy evaluation:

```
primes  =  sieve [2..]  =  sieve (2:[3..])
=  2 : sieve [x | x <- [3..], x `mod` 2 /= 0]
=  2 : sieve [x | x <- 3:[4..],  x `mod` 2 /= 0]
=  2 : sieve (3 : [x | x <- [4..],  x `mod` 2 /= 0])
=  2 : 3 : sieve [x | x <- [x|x <- [4..],  x `mod` 2 /= 0],
                      x `mod` 3 /= 0]
= ...
```

Modularity!

The first 10 primes:

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The primes between 100 and 150:

```
> takeWhile (<150) (dropWhile (<100) primes)
[101,103,107,109,113,127,131,137,139,149]
```

All twin primes:

```
> [(p,q) | (p,q) <- zip primes (tail primes), p+2==q]
[(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61),(71,73
```

Primality test?

```
> 101 `elem` primes
True

> 102 `elem` primes
nontermination

prime n  =  n == head (dropWhile (<n) primes)
```

Sharing!

There is only one copy of `primes`

Every time part of `primes` needs to be evaluated

Example: when computing `take 5 primes`

primes is (invisibly!) updated to remember the evaluated part

Example: `primes = 2 : 3 : 5 : 7 : 11 : sieve ...`

The next uses of `primes` are faster:

Example: now `primes !! 2` needs only 3 steps

Nothing special, just the automatic result of sharing

The list of Fibonacci numbers

Idea:

$$\begin{array}{rl} & 0\ 1\ 1\ 2\ \ldots \\ + & \ \ \ 0\ 1\ 1\ \ldots \\ = & 0\ 1\ 2\ 3\ \ldots \end{array}$$

From Prelude: `zipWith`

Example:
```
zipWith f [a1, a2, ...] [b1, b2, ...]
= [f a1 b1, f a2 b2, ...]
```

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

How about

```
fibs = 0 : 1 : [x+y | x <- fibs, y <- tail fibs]
```

Hamming numbers

Definition

$H = \{1\} \cup \{2 * h \mid h \in H\} \cup \{3 * h \mid h \in H\} \cup \{5 * h \mid h \in H\}$

(Due to Richard Hamming, Turing award winner 1968)

Problem: list H in increasing order: 1, 2, 3, 4, 5, 6, 8, 9, ...

```
hams :: [Int]
hams  =  1 : merge [2*h | h <- hams]
                   (merge [3*h | h <- hams]
                          [5*h | h <- hams])
```

```
merge (x:xs) (y:ys)
  | x < y      =  x : merge xs (y:ys)
  | x > y      =  y : merge (x:xs) ys
  | otherwise =  x : merge xs ys
```

Game tree

```
data Tree p v = Tree p v [Tree p v]
```

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally,
  as much as is needed
- No part of the game tree is computed twice

```
gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
gameTree next val = tree where
  tree p = Tree p (val p) (map tree (next p))

chessTree = gameTree ...
```

```
minimax :: Ord v => Int -> Bool -> Tree p v -> v
minimax d player1 (Tree p v ts) =
  if d == 0 || null ts then v
  else let vs = map (minimax (d-1) (not player1)) ts
       in if player1 then maximum vs else minimum vs
```

`> minimax 3 True chessTree`

Generates `chessTree` up to level 3

`> minimax 4 True chessTree`

Needs to search 4 levels, but only level 4 needs to be generated

12. Complexity and Optimization

Time complexity analysis

Optimizing functional programs

How to analyze and improve the time (and space) complexity of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

- Analysis much easier
- Most languages follow cbv
- Number of lazy evaluation steps $\leq$ number of cbv steps
  $\Longrightarrow$ O-analysis under cbv also correct for Haskell
  but can be too pessimistic

12.1 Time complexity analysis

Basic assumption:

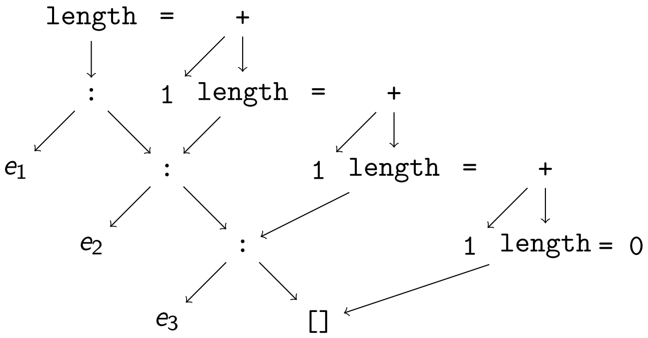
One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

Justification:

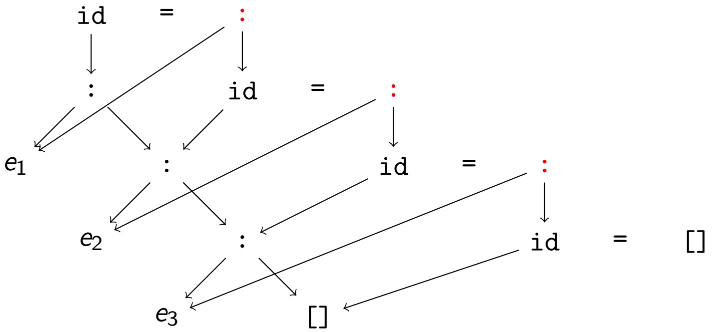
The implementation does not copy data structures
but works with pointers and sharing

Example: Reduction of `length`



Example: Reduction of `id`

```
id [] = []
id (x:xs) = x : id xs
```



Copies list but shares elements

$T_{\mathtt{f}}(n) =$ number of steps required for the evaluation of `f`
when applied to an argument of size n
in the worst case

What is "size"?

- Number of bits. Too low level.
- Better: specific measure based on the argument type of `f`
- Measure may differ from function to function.
- Frequent measure for functions on lists: the length of the list
  We use this measure unless stated otherwise
  Sufficient if `f` does not compute with the elements of the list
  Not sufficient for function . . .

How to calculate (not mechanically!) $T_{\mathtt{f}}(n)$:

1. From the equations for `f` derive equations for $T_{\mathtt{f}}$
2. If the equations for $T_{\mathtt{f}}$ are recursive, solve them

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$\begin{aligned} T_{++}(0, n) &= O(1) \\ T_{++}(m+1, n) &= T_{++}(m, n) + O(1) \end{aligned}$$

$$\Longrightarrow T_{++}(m, n) = O(m)$$

Note: `(++)` creates copy of first argument

Principle:

Every constructor of an algebraic data type takes time $O(1)$.
A constant amount of space needs to be allocated.

Example

```
reverse []       =  []
reverse (x:xs)   =  reverse xs ++ [x]
```

$$\begin{array}{lcl} T_{\texttt{reverse}}(0) & = & O(1) \\ T_{\texttt{reverse}}(n+1) & = & T_{\texttt{reverse}}(n) + T_{\texttt{++}}(n,1) \end{array}$$

$$\implies T_{\texttt{reverse}}(n) = O(n^2)$$

Observation:

Complexity analysis may need functional properties of the algorithm

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

Complications: `if-then-else`, `case`, higher-order functions, ...

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$, both for cbv and lazy evaluation. (Why?)

Consider `min xs = head(sort xs)`

$$T_{\mathtt{min}}(n) = T_{\mathtt{sort}}(n) + T_{\mathtt{head}}(n)$$

For cbv also a lower bound, but not for lazy evaluation.

Complexity analysis is *compositional* under cbv

12.2 Optimizing functional programs

> *Premature optimization is the root of all evil*
>
> *Don Knuth*

But we are in week $n - 1$ now `;-)`

The ideal of program optimization:

1. Write (possibly) inefficient but correct code
2. Optimize your code *and prove equivelence to correct version*

No duplication

Eliminate common subexpressions with `where` (or `let`)

Example

~~`f x = g (h x) (h x)`~~

```
f x = g y y  where  y = h x
```

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv
if every recursive call is in "end position",
= it is the last function call before leaving `f`,
= nothing happens afterwards
= no call of `f` is nested in another function call

Example

```
length []       =  0
length (x:xs)   =  length xs + 1
```

```
length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Compare executions:

```
length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
= length2 [b,c] 1
= length2 [c]   2
= length2 []    3
= 3
```

Fact Tail recursive definitions can be compiled into loops.
Not just in functional languages.

No (additional) stack space is needed
to execute tail recursive functions

Example

```
length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

$\rightsquigarrow$

```
loop: if null xs then return n
      xs := tail xs
      n := n+1
      goto loop
```

What does tail recursive mean for

$f\ x$ `=` `if` b `then` e_1 `else` e_2

- f does not occur in b
- if f occurs in e_i then only at the outside: $e_i = f \ldots$

Tail recursive example:

```
f x = if x > 0 then f(x-1) else f(x+1)
```

Similar for guards and `case` e `of`:

- f does not occur in e
- if f occurs in any branch then only at the outside: $f \ldots$

Accumulating parameters

An accumulating parameter is a parameter where intermediate results are accumulated.
Purpose:

- tail recursion
- replace (++) by (:)

```
length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)

length' xs  =  length2 xs 0
```

Correctness:

Lemma `length2 xs n = length xs + n`

$\Longrightarrow$ `length' xs = length xs`

Accumulating parameter: `reverse`

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$T_{\texttt{reverse}}(n) = O(n^2)$$

```
itrev [] xs      = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

Not just tail recursive also linear:

$$\begin{array}{lcl} T_{\texttt{itrev}}(0,n) & = & O(1) \\ T_{\texttt{itrev}}(m+1,n) & = & T_{\texttt{itrev}}(m,n) + O(1) \end{array}$$

$$\Longrightarrow T_{\texttt{itrev}}(m,n) = O(m)$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` $= 1$)

$$
\begin{aligned}
T_{\texttt{flat}}(1) &= O(1) \\
T_{\texttt{flat}}(h+1) &= 2 * T_{\texttt{flat}}(h) + T_{\texttt{++}}(2^h, 2^h) \\
&= 2 * T_{\texttt{flat}}(h) + O(2^h)
\end{aligned}
$$

$$\Longrightarrow T_{\texttt{flat}}(h) = O(h * 2^h)$$

With accumulating parameter:

```
flat2 :: Tree a -> [a] -> [a]
```

Accumulating parameter: `foldl`

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

```
foldr f z [x1,...,xn] = x1 `f` (... `f` (xn `f` z)...)
```

Tail recursive, second parameter accumulator:

```
foldl f z [] = z
foldl f z (x:xs) = foldl (f z x) xs
```

```
foldl f z [x1,...,xn] = (...(z `f` x1) `f` ...) `f` xn
```

Relationship between `foldr` and `foldl`:

Lemma `foldl f e = foldr f e`
if `f` is associative and ``e `f` x = x `f` e``.

Proof by induction over `xs`.

Tupling of results

Typical application:

Avoid multiple traversals of the same data structure

```
average :: [Float] -> Float
average xs = (sum xs) / (length xs)
```

Requires two traversals of the argument list.

Probably better:

```
average xs = s/n where (s,n) = sum_len xs

sum_len :: [Float] -> (Float,Int)
```

Avoid intermediate data structures

Typical example: `map g . map f = map (g . f)`

Another example: `sum [n..m]`

Precompute expensive computations

```
search :: String -> String -> Bool
search text s =
  table_search (hash_table text) (hash s,s)

bsearch = search bible

> map bsearch ["Moses", "Goethe"]
```

Better:

```
search text = \s -> table_search ht (hash s,s)
  where ht = hash_table text
```

Strong hint for compiler

Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Example: `length2` under lazy evaluation

In general: tail recursion not always better under lazy evaluation

Problem: lazy evaluation may leave many expressions unevaluated until the end, which requires more space

Space is time because it requires garbage collection — not counted by number of reductions!

13. Case Study: Parsing

Basic Parsing

Application: Parsing pico-Haskell expressions

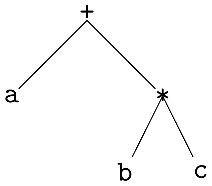
Improved Parsing

13.1 Basic Parsing

Parsing is the translation of a string into a syntax tree according to some grammar.

Example

`"a+b*c"` $\mapsto$



Parser type

```
type Parser = String -> Tree
```

```
type Parser a = String -> a
```

What if something is left over, e.g., `"a+b*c#"` ?

```
type Parser a = String -> (a,String)
```

What if there is a syntax error, e.g., `"++"` ?

```
type Parser a = String -> [(a,String)]
```

`[]`	syntax error
`[x]`	one result x
`[x,y,...]`	multiple results, ambiguous language

Alternative parser type

For unambiguous languages:

```
type Parser a = String -> Maybe (a,String)
```

Basic parsers

```
one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)
```

Example

```
char 'a' "abc" = [('a',"bc")]
char 'b' "abc" = []
```

Combining parsers

Parse anything that `p1` or `p2` can parse:

```
(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2  =  \cs -> p1 cs ++ p2 cs
```

Example

```
(char 'b' ||| char 'a') "abc" = [('a',"bc")]
```

Combining parsers

Parse first with `p1`, then the remainder with `p2`:

```
(***) :: Parser a -> Parser b -> Parser (a,b)
(p1 *** p2) xs =
  [((a,b),zs) | (a,ys) <- p1 xs, (b,zs) <- p2 ys]
```

Example

```
(char 'b' *** char 'a') "bac" = [(('b','a'), "c")]
```

```
(one isAlpha *** one isDigit *** one isDigit) "a12"
= [(('a',('1','2')), "")]
```

Transforming the result

Parse with `p`, transform result with `f`:

```
(>>>) :: Parser a -> (a -> b) -> Parser b
p >>> f  =  \xs -> [(f a,ys) | (a,ys) <- p xs]
```

Example

```
((char 'b' *** char 'a') >>> (\(x,y) -> [x,y])) "bac"
= [("ba", "c")]
```

Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b)  =  f a b

success :: a -> Parser a
success a xs  =  [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p  =  (p *** list p) >>> uncurry (:)
           |||  success []
```

Example

```
list (one isAlpha) "ab1"
= [("ab", "1"), ("a", "b1"),("", "ab1")]
```

Parsing a non-empty list of objects

```
list1 :: Parser a -> Parser [a]
list1 p  =  (p *** list p) >>> uncurry (:)
```

Parsing identifiers

```
ident :: Parser String
ident  =  (list1(one isAlpha) *** list(one isDigit))
          >>> uncurry (++)
```

Example

```
ident "ab0" = [("ab0",""), ("ab","0"), ("a","b0")]
```

Handling spaces

```
spaces :: Parser String
spaces  =  list (one isSpace)

sp :: Parser a -> Parser a
sp p  =  (spaces *** p) >>> snd
```

Example

```
(sp ident) " ab c" = [("ab", " c"), ("a", "b c")]
```

13.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{identifier} \\ & | & (\ \mathit{expr}\ \mathit{expr}\ ) \\ & | & (\ \backslash\ \mathit{identifier}\ \texttt{.}\ \mathit{expr}\ ) \end{array}$$

Examples `a`, `(f x)`, `(\x. (f x))`

The tree representation:

```
data Expr = Id String | App Expr Expr | Lam String Expr
```

Examples
```
Id "a"
App (Id "f") (Id "x")
Lam "x" (App (Id "f") (Id "x"))
```

Pico-Haskell parser

```
ch c  =  sp (char c)
id    =  sp ident

expr  =
 id >>> Id
 |||
 (ch '(' *** expr *** expr *** ch ')')
   >>> (\(_,(e1,(e2,_))) -> App e1 e2)
 |||
 (ch '(' *** ch '\' *** id *** ch '.' *** expr *** ch ')')
   >>> (\(_,(_,(x,(_,(e,_))))) -> Lam x e)
```

13.3 Improved Parsing

$$\texttt{String} \xrightarrow{\text{Lexer}} \texttt{[Token]} \xrightarrow{\text{Parser}} \texttt{Tree}$$

Example

```
data Token =
  LParant | RParant | BSlash | Dot | Ident String
```

`"(\x1 .  x2)"` $\xrightarrow{\text{Lexer}}$

```
  [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]
```

Why?

- Lexer based on regular expressions
  $\Longrightarrow$ lexer can be more efficient than general parser
- Lexer can already remove spaces and comments
  $\Longrightarrow$ simplifies parsing

Generalizing the implementation

So far:

```
type Parser a = String -> [(a,String)]
```

Now:

```
type Parser a b = [a] -> [(b,[a])]
```

None of the parser combinators `***`, `|||`, `>>>` change,
only their types become more general!

So far:

```
(***) ::  Parser a -> Parser b -> Parser (a,b)
```

Now:

```
(***) ::  Parser a b -> Parser a c -> Parser a (b,c)
```

Some literature:

- Chapter 8 of Hutton's *Programming in Haskell*
- Section 17.5 in Thompson's Haskell book (3rd edition)
- Many papers on functional parsers

14. Monads

Beyond `IO`: *Monads*

- The type `IO` is a special instance of the general class of monads
- Monads are a general approach to effectful computations (‘actions’)
- Idea: carry around data in the background implicitly

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

where `m` is a type constructor, for example `IO`

>>= ('bind'), or what do really means

Primitive:

```
(>>=) :: m a -> (a -> m b) -> m b
```

How it works:

act `>>=` *f* execute action *act* `:: m a`
which returns a result *v* `:: a`
then execute action *f* *v*

`do` *x* `<-` act_1
    act_2

is syntax for act_1 `>>= (\x ->` act_2`)`

Example

```
do x <- getChar
   putChar x
```

⇝

```
getChar >>= (\x -> putChar x)
```

In general

$$\begin{array}{l} \texttt{do}\ \ x_1\ \texttt{<-}\ a_1 \\ \quad\ \ \vdots \\ \quad\ \ x_n\ \texttt{<-}\ a_n \\ \quad\ \ act \end{array} \qquad \text{is syntax for} \qquad \begin{array}{l} a_1\ \texttt{>>=}\ \texttt{\textbackslash}x_1\ \texttt{->} \\ \vdots \\ a_n\ \texttt{>>=}\ \texttt{\textbackslash}x_n\ \texttt{->} \\ act \end{array}$$

More Monads!

Maybe as a monad

A frequent code pattern when working with Maybe:

```
case m of
  Nothing -> Nothing
  Just x -> ...
```

This pattern can be hidden inside >>=:

```
instance Monad Maybe where
  m >>= f  =  case m of
                Nothing -> Nothing
                Just x -> f x
  return v  =   Just v
```

Failure (= `Nothing`) propagation and unwrapping of `Just` is now built into `do`!

```
instance Monad Maybe where
  m >>= f  =  case m of
                Nothing -> Nothing
                Just x -> f x
  return v  =  Just v
```

Example: evaluation of `Form`

```
eval :: [(Name,Bool)] -> Form -> Maybe Bool
eval _ T  =  return True
eval _ F  =  return False
eval v (Var x)  =  lookup x v
eval v (f1 :&: f2)  =  do b1 <- eval v f1
                          b2 <- eval v f2
                          return (b1 && b2)
...
```

Example:

```
p1 *** p2  =  \xs ->
  case p1 xs of
    Nothing -> Nothing
    Just(b,ys) -> case p2 ys of
                    Nothing -> Nothing
                    Just(c,zs) -> Just((b,c),zs)
```

$\rightsquigarrow$

```
p1 *** p2  = \xs ->
  do (b,ys) <- p1 xs
     (c,zs) <- p2 ys
     return ((b,c),zs)
```

The `do` version has a much more general type `Monad m => ...`

`Maybe` models possible failure with `Just`/`Nothing`

The `do` of the `Maybe` monad hides `Just`/`Nothing` and propagates failure automatically

List as a monad

```
instance Monad [] where
  xs >>= f   =  concat(map f xs)
  return v   =  [v]
```

Now we can compose computations on list nicely (via `do`).

Example

```
dfs :: (a -> [a]) -> (a -> Bool) -> a -> [a]
dfs nexts found start  =  find start
  where
  find x  =  if found x then return x
             else do x' <- nexts x
                     find x'
```

The Haskell way of backtracking
Lazy evaluation produces only as many elements as you ask for.

A “time” monad

Combine value with number of computation steps:

```
data Time a = Time a Integer

return :: a -> Time a
return a  =  Time a 1

(>>=) :: Time a -> (a -> Time b) -> Time b
Time a m >>= f  =  let Time b n = f a in Time b (m+n)
```

Example

```
tapp :: [a] -> [a] -> Time [a]
tapp [] ys = return ys
tapp (x:xs) ys = do  zs <- tapp xs ys; return (x:zs)
```

The Monad laws

```
return a >>= f  =  f a
m >>= return  =  m
m >>= (\x -> (f x >>= g))  =  (m >>= f) >>= g
```

The Monad laws in do form (1)

Functional:

```
return a >>= f   =   f a
```

"Imperativ":

```
do { x' <- return x;     =     do { f x }
     f x'}
```

The Monad laws in do form (2)

Functional:

```
m >>= return  =  m
```

"Imperativ":

```
do { x <- m;              =     do { m }
     return x}
```

The full Monad story

There are two more classes before Monad:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)

class Functor f => Applicative f where
  ...

class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
  return :: a -> f a
```

- Doc & code: `Control.Monad`
- HaskellWiki: *Functor-Applicative-Monad Proposal*