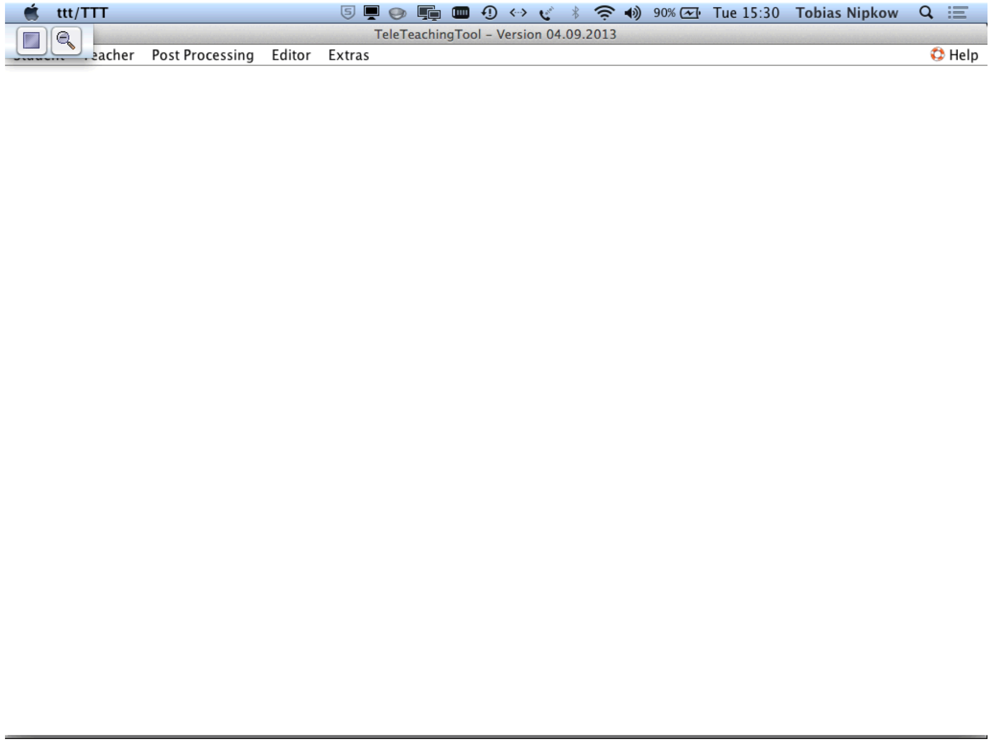
Script generated by TTT

Title: Nipkow: Info2 (29.10.2013)

Date: Tue Oct 29 15:30:02 CET 2013

Duration: 94:15 min

Pages: 166



4.2 Generic functions: Polymorphism

Polymorphism = one function can have many types

Example

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

⋮

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:

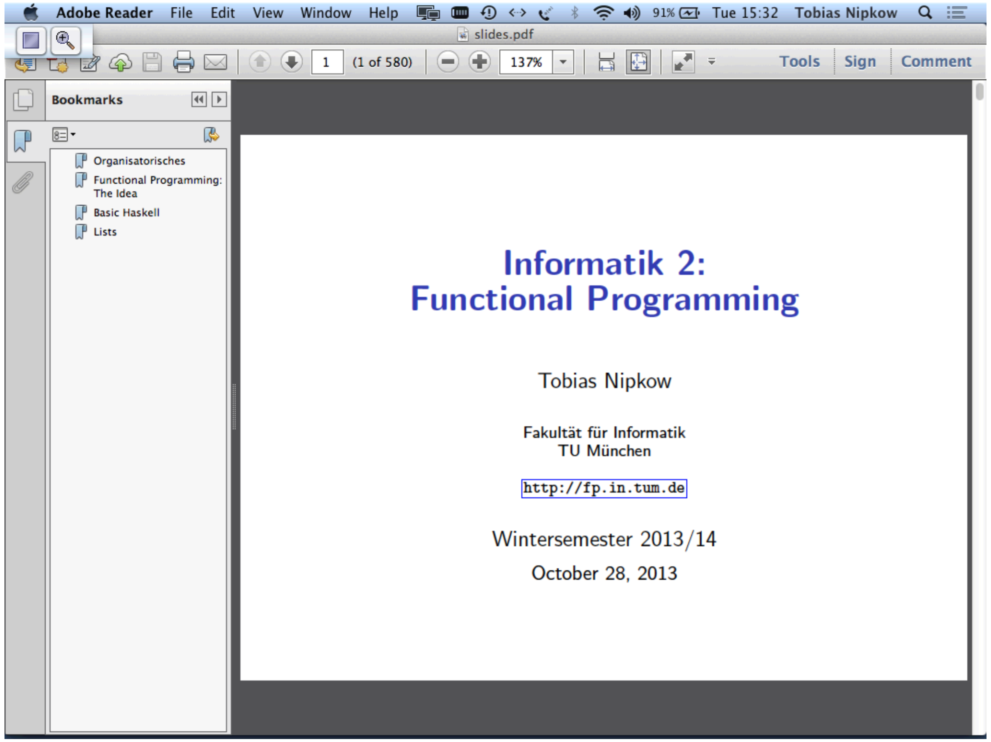
Types may contain type variables ("parameters")

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

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 (x,y)  =  x
```

Defining polymorphic functions

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

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

Defining polymorphic functions

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

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


swap (x,y)  =  (y,x)
```

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 x y  =  if x then 'c' else 'd'
```

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 x y  =  if x then x else y
```

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
```

Polymorphic list functions from the Prelude

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

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]
```

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]
```

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]
```

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
```

Polymorphic list functions from the Prelude

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

Polymorphic list functions from the Prelude

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

tail, init :: [a] -> [a]
```

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]
```


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"
```


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 ==
```


Polymorphic list functions from the Prelude


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

Polymorphic list functions from the Prelude

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

Polymorphic list functions from the Prelude

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

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

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')]
```

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

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")
```

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 =
  unzip(zip xs ys) ==
```

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 =
  unzip(zip xs ys) == (xs, ys)
```

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

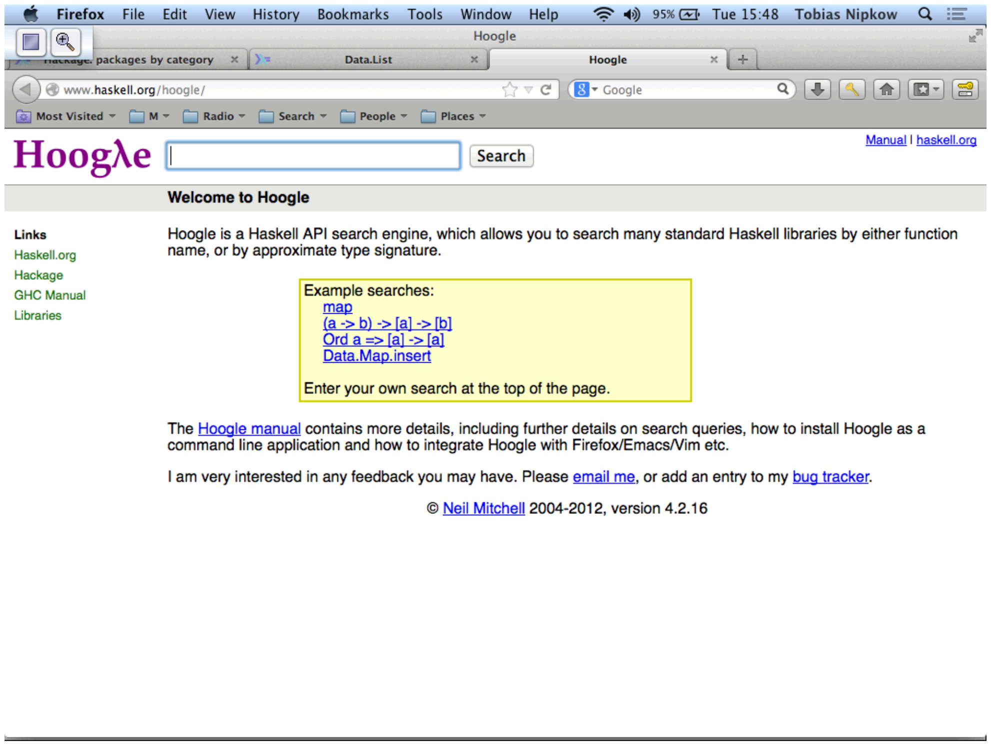
Firefox File Edit View History Bookmarks Tools Window Help 95% Tue 15:47 Tobias Nipkow

Haskell libraries

- Prelude and much more

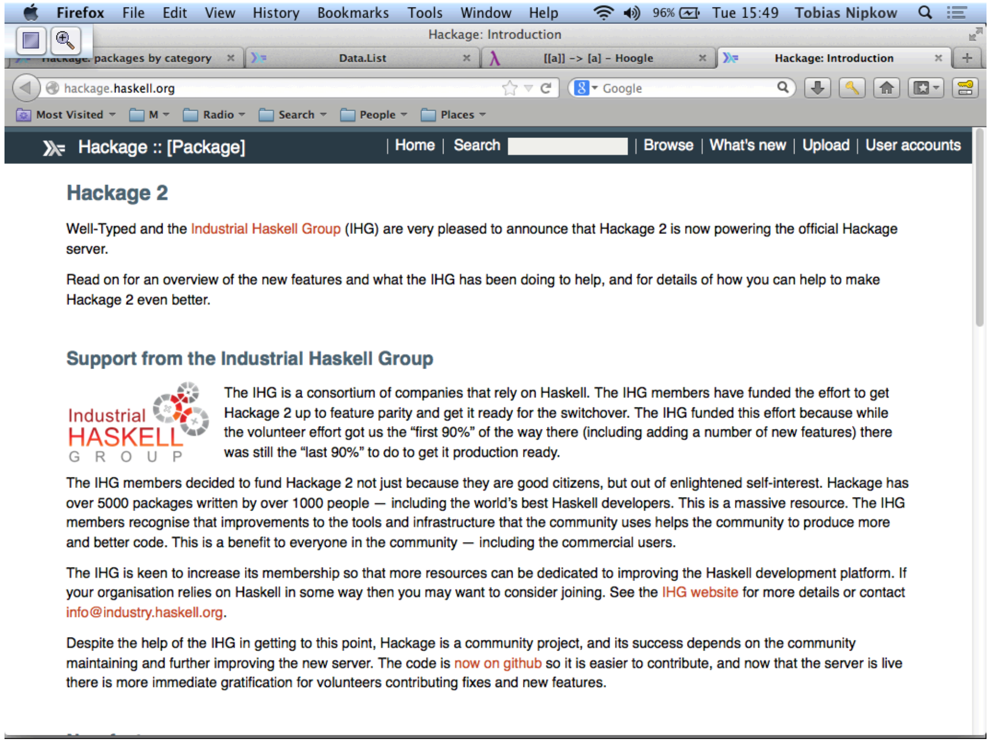
Haskell libraries

- Prelude and much more
- Hoogle — searching the Haskell libraries



Haskell libraries

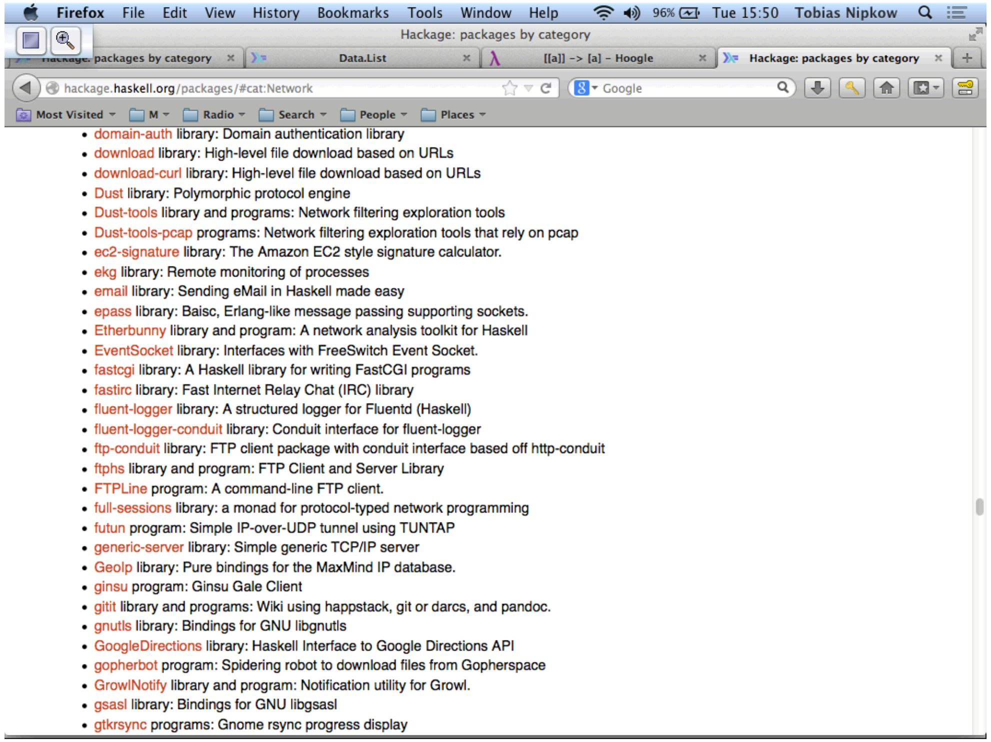
- Prelude and much more
- Hoogle — searching the Haskell libraries
- Hackage — a collection of Haskell packages



Haskell libraries

- Prelude and much more
- Hoogle — searching the Haskell libraries

  http://www.haskell.org/hoogle/
- Hackage — a collection of Haskell packages



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
```

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

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

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

Numeric types

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

Numeric types

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

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

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*.

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`

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
  ⋮
  ```

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
  ```

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.

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: functions
- The class `Ord` of *ordered types*, i.e. types that posess
  `(<)  ::  Ord a => a -> a -> Bool`

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: functions
- 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.

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: functions
- 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: == []

Warning: == []

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

Warning: == []

```
null :: Eq a => [a] -> Bool
null xs  =  xs == []
```

Warning: == []

```
null :: Eq a => [a] -> Bool
null xs  =  xs == []
```

Why?

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

Warning: == []

```
null :: Eq a => [a] -> Bool
null xs  =  xs == []
```

Why?

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

Warning: == []

```
null :: Eq a => [a] -> Bool
null xs  =  xs == []
```

Why?

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

Better:

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


Warning: == []

```
null :: Eq a => [a] -> Bool
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

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
```


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
```

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
```

Conditional properties have result type `Property`

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
```

4.3 Case study: Pictures

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


4.3 Case study: Pictures

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


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

4.3 Case study: Pictures

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

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

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

```
flipH :: Picture -> Picture
```

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

flipV :: Picture -> Picture
```

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

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

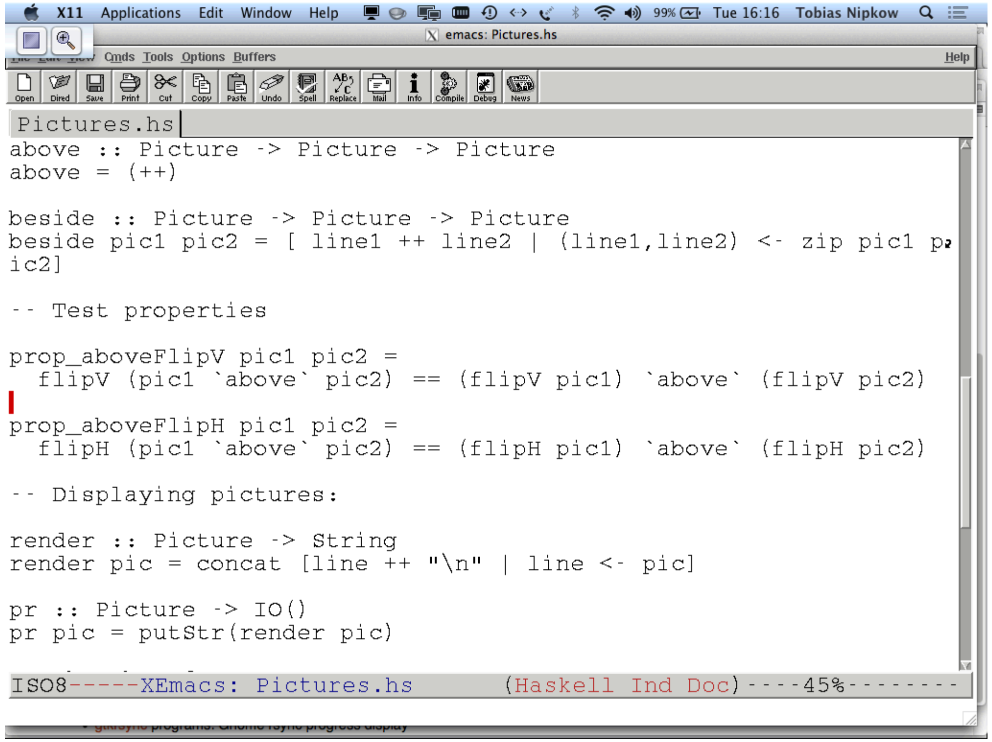
```
flipH :: Picture -> Picture
flipH  =  reverse

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

rarr :: Picture
rarr  =  flipV 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
```

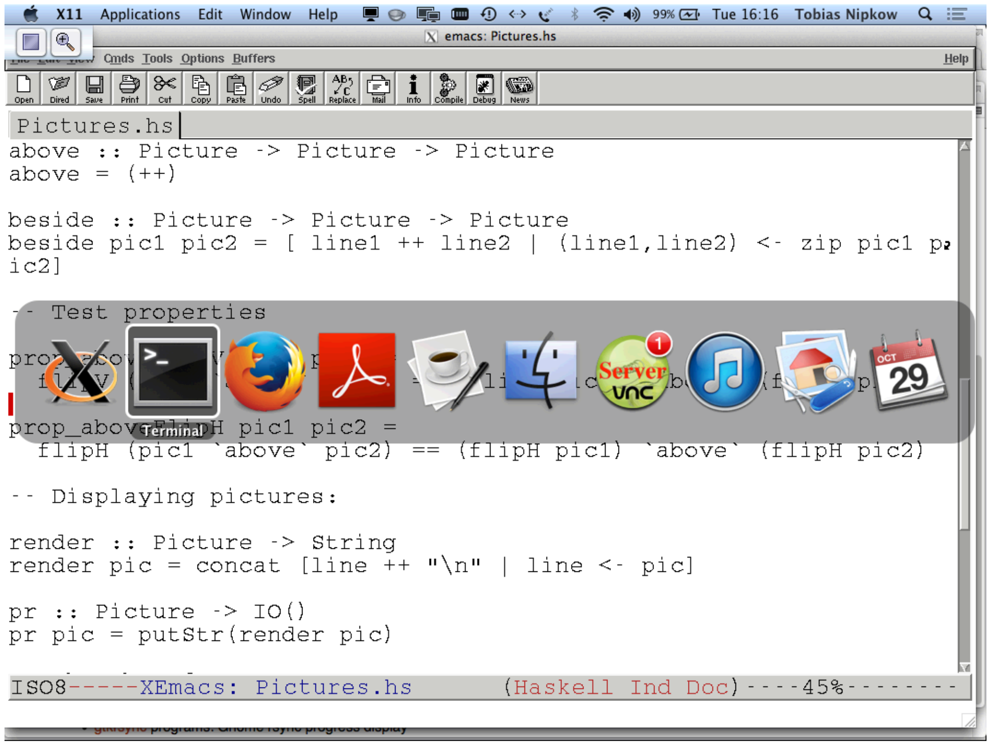
```
flipH :: Picture -> Picture
flipH  =  reverse

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

rarr :: Picture
rarr  =  flipV larr

darr :: Picture
darr  =  flipH uarr

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

```
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]
```

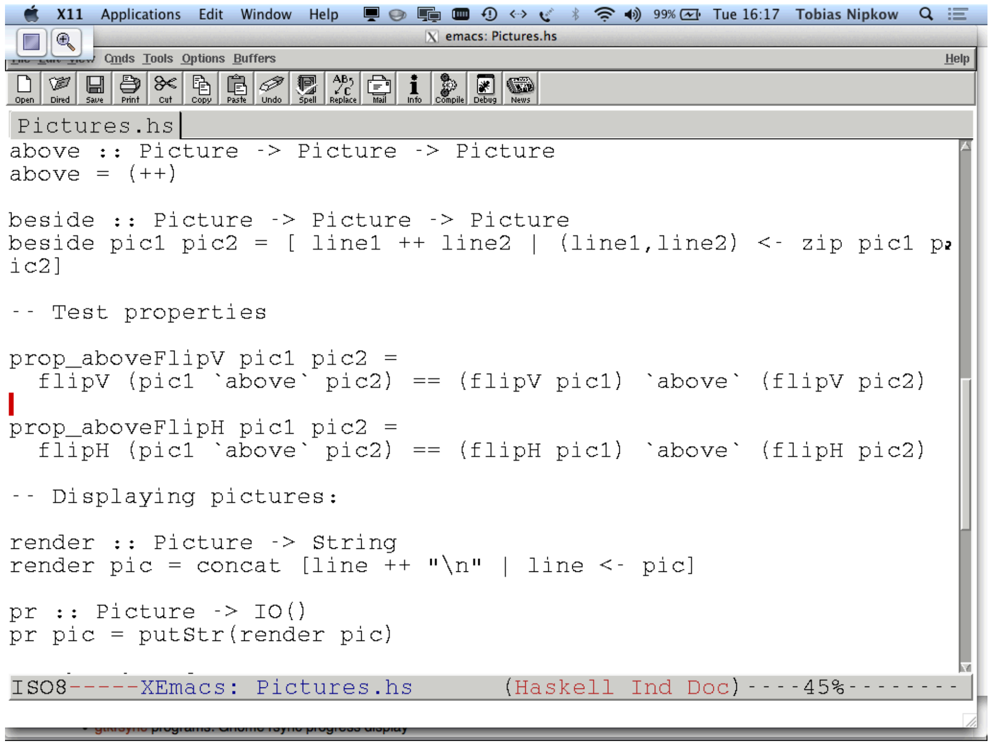
Pictures.hs



X11 Applications Edit Window Help Tue 16:16 Tobias Nipkow
emacs: Pictures.hs
Pictures.hs
```
above :: Picture -> Picture -> Picture
above = (++)

beside :: Picture -> Picture -> Picture
beside pic1 pic2 = [ line1 ++ line2 | (line1,line2) <- zip pic1 pic2]

-- Test properties

prop_aboveFlipV pic1 pic2 =
  flipV (pic1 `above` pic2) == (flipV pic1) `above` (flipV pic2)

prop_aboveFlipH pic1 pic2 =
  flipH (pic1 `above` pic2) == (flipH pic1) `above` (flipH pic2)

-- Displaying pictures:

render :: Picture -> String
render pic = concat [line ++ "\n" | line <- pic]

pr :: Picture -> IO()
pr pic = putStr(render pic)
```
ISO8-----XEmacs: Pictures.hs (Haskell Ind Doc)----45%--------



X11 Applications Edit Window Help Tue 16:16 Tobias Nipkow
emacs: Pictures.hs
Pictures.hs
```
above :: Picture -> Picture -> Picture
above = (++)

beside :: Picture -> Picture -> Picture
beside pic1 pic2 = [ line1 ++ line2 | (line1,line2) <- zip pic1 pic2]

-- Test properties

  flipH (pic1 `above` pic2) == (flipH pic1) `above` (flipH pic2)

-- Displaying pictures:

render :: Picture -> String
render pic = concat [line ++ "\n" | line <- pic]

pr :: Picture -> IO()
pr pic = putStr(render pic)
```
Terminal
ISO8-----XEmacs: Pictures.hs (Haskell Ind Doc)----45%--------



X11 Applications Edit Window Help Tue 16:17 Tobias Nipkow
emacs: Pictures.hs
Pictures.hs
```
above :: Picture -> Picture -> Picture
above = (++)

beside :: Picture -> Picture -> Picture
beside pic1 pic2 = [ line1 ++ line2 | (line1,line2) <- zip pic1 pic2]

-- Test properties

prop_aboveFlipV pic1 pic2 =
  flipV (pic1 `above` pic2) == (flipV pic1) `above` (flipV pic2)

prop_aboveFlipH pic1 pic2 =
  flipH (pic1 `above` pic2) == (flipH pic1) `above` (flipH pic2)

-- Displaying pictures:

render :: Picture -> String
render pic = concat [line ++ "\n" | line <- pic]

pr :: Picture -> IO()
pr pic = putStr(render pic)
```
ISO8-----XEmacs: Pictures.hs (Haskell Ind Doc)----45%--------

```
Pictures.hs
above :: Picture -> Picture -> Picture
above = (++)

beside :: Picture -> Picture -> Picture
beside pic1 pic2 = [ line1 ++ line2 | (line1,line2) <- zip pic1 pic2]

-- Test properties

prop_aboveFlipV pic1 pic2 =
  flipV (pic1 `above` pic2) == (flipV pic1) `above` (flipV pic2)

prop_aboveFlipH pic1 pic2 =
  flipH (pic1 `above` pic2) == (flipH pic1) `above` (flipH pic2)

-- Displaying pictures:

render :: Picture -> String
render pic = concat [line ++ "\n" | line <- pic]

pr :: Picture -> IO()
pr pic = putStr(render pic)

ISO8-----XEmacs: Pictures.hs      (Haskell Ind Doc)----45%--------
Undo!
```

Chessboards

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

wSq  =  replicate 5 (replicate 5 ' ')

alterH :: Picture -> Picture -> Int -> Picture
```

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)
```

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)
```

Chessboards

```haskell
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
```

Chessboards

```haskell
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
```

Terminal Shell Edit View Window Help 99% Tue 16:21 Tobias Nipkow

Code — ghc — 76×24

```
Loading package base ... linking ... done.
Prelude> :l Pictures
[1 of 1] Compiling Main             ( Pictures.hs, interpreted )
Ok, modules loaded: Main.
*Main>
*Main>
*Main>
*Main>
*Main>
*Main>
*Main> quickCheck prop_aboveFlipH
Loading package array-0.4.0.0 ... linking ... done.
Loading package deepseq-1.3.0.0 ... linking ... done.
Loading package old-locale-1.0.0.4 ... linking ... done.
Loading package time-1.4 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
Loading package containers-0.4.2.1 ... linking ... done.
Loading package pretty-1.1.1.0 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.5.1.1 ... linking ... done.
*** Failed! Falsifiable (after 3 tests and 4 shrinks):
[""]
["a"]
*Main> _
```

Terminal Shell Edit View Window Help 99% Tue 16:22 Tobias Nipkow

Code — ghc — 76×24

```
["a"]
*Main> chessboard 3
["#####     #####","#####     #####","#####     #####","#####     #####","##
###     #####","     #####     ","     #####     ","     #####     ","     #
####     ","     #####     ","#####     #####","#####     #####","#####
#####","#####     #####","#####     #####"]
*Main> pr (chessboard 3)
#####     #####
#####     #####
#####     #####
#####     #####
#####     #####
     #####
     #####
     #####
     #####
     #####
#####     #####
#####     #####
#####     #####
#####     #####
#####     #####
*Main> _
```

```
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
#####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
     #####     #####     #####     #####
*Main>
```

Exercise

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

4.4 Pattern matching

4.4 Pattern matching

Every list can be constructed from `[]`

4.4 Pattern matching

Every list can be constructed from `[]`
by repeatedly adding an element at the front

4.4 Pattern matching

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

4.4 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 : []`

4.4 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 : []`

4.4 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 : []`

4.4 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 : []`
$[x_1, \ldots, x_n]$	$x_1 : \ldots : x_n :$ `[]`

4.4 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 : []`
$[x_1, \ldots, x_n]$	$x_1 : \ldots : x_n :$ `[]`

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

$\Longrightarrow$
Every list is either
`[]` or of the form
$x : xs$

$\Longrightarrow$

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*)

$\Longrightarrow$

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 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.

$\Longrightarrow$

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]`

(++) 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.

$\Longrightarrow$

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
```

Patterns

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

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*.

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*.

Patterns

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

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

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"`, ...

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

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 $C\ p_1\ \dots\ p_n$
  where C is a constructor and each p_i is a pattern

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 $(p_1, \ldots, p_n)$ 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
```

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

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

Function definitions by pattern matching

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

If f has multiple arguments:

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

Function definitions by pattern matching

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

If f has multiple arguments:

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

Conditional equations:

$$f\ \ patterns\ \mid\ condition\ \ =\ \ e$$

Function definitions by pattern matching

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

If f has multiple arguments:

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

Conditional equations:

$$f\ \ patterns\ \mid\ condition\ \ =\ \ e$$

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

Function definitions by pattern matching

Example (contrived)

```
true12 (True : True : _)  =  True
true12 _  =  False
```

Function definitions by pattern matching

Example (contrived)

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


same12 (x : _) (_ : y : _)  =  x == y
```

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
```

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
```

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 (x : y : z : _)  =  x < y && y < z
```

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
```

4.5 Recursion over lists

Example

```
length []         =  0
```

4.5 Recursion over lists

Example

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

4.5 Recursion over lists

Example

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

reverse []        =  []
```

4.5 Recursion over lists

Example

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

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

4.5 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
```

4.5 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    -- base case
f (x : xs)  =  rec     -- recursive case
```

Primitive recursion on lists:

```
f []        =  base     -- base case
f (x : xs)  =  rec      -- recursive case
```

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

Finding primitive recursive definitions

Example

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

Finding primitive recursive definitions

Example

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

Finding primitive recursive definitions

Example

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

Finding primitive recursive definitions

Example

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

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

Finding primitive recursive definitions

Example

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

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

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

Insertion sort

Example

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

Insertion sort

Example

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

Insertion sort

Example

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

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

Insertion sort

Example

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

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

Insertion sort

Example

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

ins :: a -> [a] -> [a]
ins x []  =  [x]
ins x (y:ys)
```

Insertion sort

Example

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

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

Insertion sort

Example

```
inSort :: [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
```

Beyond primitive recursion: Complex patterns

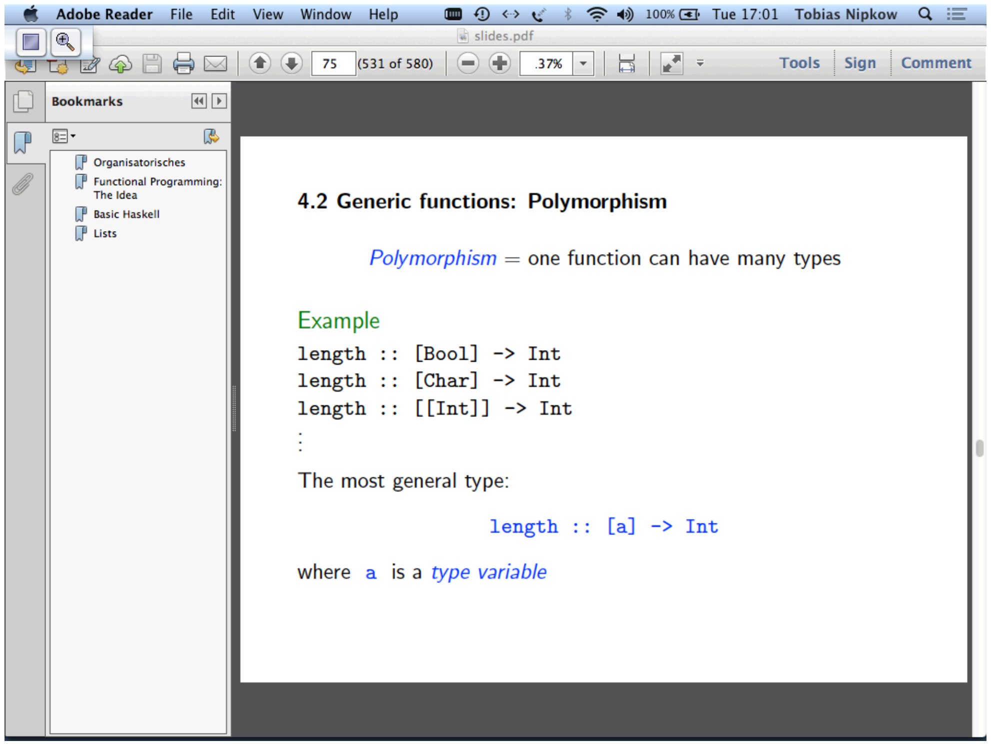
Example

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

Beyond primitive recursion: Complex patterns

Example

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



4.2 Generic functions: Polymorphism

Polymorphism = one function can have many types

Example

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

The most general type:

`length :: [a] -> Int`

where `a` is a *type variable*


Bitbanana
Collective Creations
Wir wollen, dass die
Welt eure Ideen sieht!
www.bit Page 4 of 12 com


Wir bieten dir...
Bitbanana
Collective Creations
Mitmacher
Page 5 of 12
www.bitbanana.com


Wir bieten dir...
Bitbanana
Collective Creations
Support
Page 6 of 12
www.bitbanana.com


Wir bieten dir...
Bitbanana
Collective Creations
Publishing
Page 7 of 12
www.bitbanana.com

Und DU machst den meisten Gewinn.

Egal wie es ausgeht, du siehst als erstes Geld (70% der Umsätze) und trägst 0% Risiko.

www.bitbanana.com

Bitbanana

Collective Creations

Be part of the team!

/bitbanana

www.bitbanana.com

Bitbanana

Collective Creations

Be part of the team!

/bitbanana

www.bitbanana.com