**4.2 Generic functions: Polymorphism**

Polymorphism = one function can have many types

Example: `concat`

```haskell
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]```
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
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:

\[
\begin{align*}
  f : : T \to U & \quad T' \leq T \\
  \quad f : : T' \to U
\end{align*}
\]

(remember: horizontal line = implication)

Parametric polymorphism as in Haskell:

- Types may contain type variables ("parameters")

\[
\begin{align*}
  f : : T \\
  f : : T[U/a]
\end{align*}
\]

where \( T[U/a] = "T with a replaced by U" \)
Two kinds of polymorphism

Subtype polymorphism as in Java:
\[
f :: T \to U \quad T' \leq T
\]
\[
f :: T' \to U
\]
(remember: horizontal line = implication)

Parametric polymorphism as in Haskell:
Types may contain type variables (“parameters”)
\[
f :: T
\]
\[
f :: T[U/a]
\]
where \( T[U/a] = \text{“} T \text{ with } a \text{ replaced by } U \text{“} \)
Example: \((a \to a)[\text{Bool}/a]\)

Two kinds of polymorphism

Subtype polymorphism as in Java:
\[
f :: T \to U \quad T' \leq T
\]
\[
f :: T' \to U
\]
(remember: horizontal line = implication)

Parametric polymorphism as in Haskell:
Types may contain type variables (“parameters”)
\[
f :: T
\]
\[
f :: T[U/a]
\]
where \( T[U/a] = \text{“} T \text{ with } a \text{ replaced by } U \text{“} \)
Example: \((a \to a)[\text{Bool}/a] = \text{Bool} \to \text{Bool}\)

Two kinds of polymorphism

Subtype polymorphism as in Java:
\[
f :: T \to U \quad T' \leq T
\]
\[
f :: T' \to U
\]
(remember: horizontal line = implication)

Parametric polymorphism as in Haskell:
Types may contain type variables (“parameters”)
\[
f :: T
\]
\[
f :: T[U/a]
\]
where \( T[U/a] = \text{“} T \text{ with } a \text{ replaced by } U \text{“} \)
Example: \((a \to a)[\text{Bool}/a] = \text{Bool} \to \text{Bool}\)

(Often called ML-style polymorphism)

Defining polymorphic functions

\[
id :: a \to a
\]
\[
id x = x
\]
Defining polymorphic functions

\[
id :: a \rightarrow a \\
id x = x
\]

\[
\text{fst} :: (a,b) \rightarrow a \\
\text{fst} (x,y) = x
\]

\[
id :: a \rightarrow a \\
id x = x
\]

\[
\text{fst} :: (a,b) \rightarrow a \\
\text{fst} (x,y) = x
\]

\[
\text{swap} :: (a,b) \rightarrow (b,a) \\
\text{swap} (x,y) = (y,x)
\]

\[
\text{silly} :: \text{Bool} \rightarrow a \rightarrow \text{Char} \\
\text{silly} x y = \text{if} \ x \ \text{then} \ 'c' \ \text{else} \ 'd'
\]
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 :: 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

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

reverse :: [a] -> [a]
reverse [1, 2, 3] = [3, 2, 1]
Polymorphic list functions from the \textbf{Prelude}

\begin{verbatim}
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"
\end{verbatim}

\begin{verbatim}
head, last :: [a] -> a
head "list" = 'l', last "list" = 't'
\end{verbatim}

\begin{verbatim}
head, last :: [a] -> a
head "list" = 'l', last "list" = 't'
tail, init :: [a] -> [a]
tail "list" = "ist",
\end{verbatim}
Polymorphic list functions from the \texttt{Prelude}

\begin{verbatim}
head, last :: [a] -> a
head "list" = 'l', last "list" = 't'

tail, init :: [a] -> [a]
tail "list" = "ist", init "list" = "lis"
\end{verbatim}

\begin{verbatim}
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"
\end{verbatim}

Polymorphic list functions from the \texttt{Prelude}

\begin{verbatim}
concat ::
concat [[1, 2], [3, 4], [0]] = [1, 2, 3, 4, 0]
\end{verbatim}

\begin{verbatim}
concat :: [[a]] -> [a]
concat [[1, 2], [3, 4], [0]] = [1, 2, 3, 4, 0]
\end{verbatim}
Polymorphic list functions from the Prelude

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

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

```haskell
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"]```

```haskell
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) ==
    unzip(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 = length xs == length ys ==>
    unzip(zip xs ys) == (xs, ys)

Haskell libraries

- Prelude and much more

Hoogle — searching the Haskell libraries


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

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

Further list functions from the **Prelude**

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

Further list functions from the **Prelude**

```haskell
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,
```
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
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

(+) :: 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
  ...
Other important type classes

- The class `Eq` of *equality types*, i.e. types that possess
  
  `(==) :: Eq a => a -> a -> Bool
  
  `(/=)` :: Eq a => a -> a -> Bool
  
  Most types are of class `Eq`. Exception: `functions`
Other important type classes

- The class Eq of equality types, i.e. types that possess
  \[(=) \quad :: \quad \text{Eq} \ a \Rightarrow a \to a \to \text{Bool}\]
  \[(/=) \quad :: \quad \text{Eq} \ a \Rightarrow a \to a \to \text{Bool}\]
  Most types are of class Eq. Exception: functions

- The class Ord of ordered types, i.e. types that possess
  \[(<) \quad :: \quad \text{Ord} \ a \Rightarrow a \to a \to \text{Bool}\]
  \[(\leq) \quad :: \quad \text{Ord} \ a \Rightarrow a \to a \to \text{Bool}\]
Other important type classes

- The class `Eq` of *equality types*, i.e. types that possess
  \[(=) : \ Eq \ a \to a \to \text{Bool}\]
  \[(/=) : \ Eq \ a \to a \to \text{Bool}\]
  Most types are of class `Eq`. Exception: functions

- The class `Ord` of *ordered types*, i.e. types that possess
  \[(<) : \ Ord \ a \to a \to \text{Bool}\]
  \[(<=) : \ Ord \ a \to a \to \text{Bool}\]

More on type classes later. Don’t confuse with OO classes.

```
null \text{xs} \ = \ \text{xs} \ == \ []
```

```null :: [a] \to \text{Bool}
null \text{xs} \ = \ \text{xs} \ == \ []```
Warning: == []

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

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

The solution: specialize the polymorphic property, e.g.
prop_reverse :: [Int] -> 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

Conditional properties have result type Property

Example
QuickCheck does not find a counterexample to
prop.reverse :: [Int] -> Bool
prop.reverse xs = reverse xs == xs

Now QuickCheck works
4.3 Case study: Pictures

type Picture = [String]

Conditional properties have result type Property

Example

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

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

```haskell
type Picture = [String]

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

flipH :: Picture -> Picture
```

```haskell
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 =
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
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 = putStrLn(render pic)

-- Chessboards

pr pic = putStrLn(render pic)

-- Chessboards

lapnippowld:Code nippow$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l Pictures
[1 of 1] Compiling Main           ( Pictures.hs, interpreted )
Ok, modules loaded: Main.
*Main> quickCheck

Chessboards
Chessboards

\[ b\text{Sq} = \text{replicate 5} \left( \text{replicate 5} \ '##' \right) \]

\[ w\text{Sq} = \text{replicate 5} \left( \text{replicate 5} \ ' ' \right) \]

\[ \text{alterH} :: \text{Picture} \to \text{Picture} \to \text{Int} \to \text{Picture} \]

\[ \text{alterH pic1 pic2 1} = \text{pic1} \]
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
Chessboards

\[
\begin{align*}
\text{bSq} &= \text{replicate 5 (replicate 5 '#')} \\
\text{wSq} &= \text{replicate 5 (replicate 5 ' ')} \\
\text{alterH} :: \text{Picture} \to \text{Picture} \to \text{Int} \to \text{Picture} \\
\text{alterH} \text{ pic1 pic2 1} &= \text{pic1} \\
\text{alterH} \text{ pic1 pic2 n} &= \text{pic1 'beside' alterH pic2 pic1 (n-1)} \\
\text{alterV} :: \text{Picture} \to \text{Picture} \to \text{Int} \to \text{Picture} \\
\text{alterV} \text{ pic1 pic2 1} &= \text{pic1} \\
\text{alterV} \text{ pic1 pic2 n} &= \text{pic1 'above' alterV pic2 pic1 (n-1)} \\
\text{chessboard} :: \text{Int} \to \text{Picture} \\
\text{chessboard} \text{ n} &= \text{alterV bw wb n where} \\
&\quad \text{bw} = \text{alterH bSq wSq n} \\
&\quad \text{wb} = \text{alterH wSq bSq n}
\end{align*}
\]

Exercise

Ensure that the lower left square of \text{chessboard n} is always black.

4.4 Pattern matching
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]

<table>
<thead>
<tr>
<th>syntactic sugar</th>
<th>in reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>3 : []</td>
</tr>
<tr>
<td>[2, 3]</td>
<td></td>
</tr>
</tbody>
</table>
4.4 Pattern matching

Every list can be constructed from \([\,]\) by repeatedly adding an element at the front with the "cons" operator \((\cdot) : : a \rightarrow [a] \rightarrow [a]\)

<table>
<thead>
<tr>
<th>syntactic sugar</th>
<th>in reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3]</td>
<td>3 : []</td>
</tr>
<tr>
<td>[2, 3]</td>
<td>2 : 3 : []</td>
</tr>
<tr>
<td>[1, 2, 3]</td>
<td>1 : 2 : 3 : []</td>
</tr>
<tr>
<td>([x_1, \ldots, x_n])</td>
<td>(x_1 : \ldots : x_n : [])</td>
</tr>
</tbody>
</table>

Note: \(x : y : zs = x : (y : zs)\)
\((\cdot)\) associates to the right
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.

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

Therefore these definitions make sense:

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

implies Patterns allow unique decomposition = pattern matching.
Patterns

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

⇒ 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
Patterns

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

⇒ 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

\[ \text{head :: [a] -> a} \]
\[ \text{head \ (x : _) = x} \]
Function definitions by pattern matching

Example

\[
\begin{align*}
\text{head} &: [a] \to a \\
\text{head} (x : _) &= x \\

\text{tail} &: [a] \to [a] \\
\text{tail} (_, : xs) &= xs \\

\text{null} &: [a] \to \text{Bool} \\
\text{null} [] &= \text{True} \\
\text{null} (_, : _) &= \text{False} \\
\end{align*}
\]

Function definitions by pattern matching

\[
\begin{align*}
\text{If } f \text{ has multiple arguments:} \\
& f \text{ pat}_1 = e_1 \\
& \vdots \\
& f \text{ pat}_n = e_n \\
\end{align*}
\]

\[
\begin{align*}
& f \text{ pat}_1 = e_1 \\
& \vdots \\
& f \text{ pat}_n = e_n \\
\end{align*}
\]

Conditional equations:

\[
\begin{align*}
& f \text{ patterns } | \text{ condition } = e \\
\end{align*}
\]

When \( f \) is called, the equations are tried in the given order.
Function definitions by pattern matching

Example (contrived)

\[
\begin{align*}
\text{true12} & :: [\text{Bool}] \to \text{Bool} \\
\text{true12} \;&:: \; \text{True} \; : \; \text{True} \; : \; \_ \; = \; \text{True} \\
\text{true12} \;&:: \; \_ \; = \; \text{False} \\
\text{same12} & :: \; \text{Eq} \; a \; \rightarrow \; [a] \; \rightarrow \; [a] \; \rightarrow \; \text{Bool} \\
\text{same12} \;&:: \; \text{x} \; : \; \_ \; \rightarrow \; \_ \; : \; \text{y} \; : \; \_ \; = \; \text{x} \; \&\& \; \text{y} \\
\text{asc3} & :: \; \text{x} \; : \; \text{y} \; : \; \text{z} \; : \; \_ \; = \; \text{x} \; < \; \text{y} \; \&\& \; \text{y} \; < \; \text{z}
\end{align*}
\]
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
asc3 (x : y : _) = x < y

4.5 Recursion over lists

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
asc3 (x : y : _) = x < y
asc3 _ = True
4.5 Recursion over lists

Example

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

Primitive recursion on lists:

\[
\begin{align*}
    f [] &= base & \text{-- base case} \\
    f (x : xs) &= rec & \text{-- recursive case}
\end{align*}
\]

- base: no call of \( f \)
**Primitive recursion** on lists:

\[
\begin{align*}
f \; \text{[]} & \; = \; \text{base} \quad -- \quad \text{base case} \\
f \; (x : \; xs) & \; = \; \text{rec} \quad -- \quad \text{recursive case}
\end{align*}
\]

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

\( f \) may have additional parameters.

---

**Finding primitive recursive definitions**

**Example**

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

```haskell
concat (xs : xs) = xs ++ concat xs
```
Finding primitive recursive definitions

Example

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

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

Insertion sort

Example

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

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

\[ \text{inSort} :: [a] \rightarrow [a] \]
\[ \text{inSort} \; [] = [] \]
\[ \text{inSort} \; (x:xs) = \text{ins} \; x \; (\text{inSort} \; xs) \]

\[ \text{ins} :: a \rightarrow [a] \rightarrow [a] \]
\[ \text{ins} \; x \; [] = [x] \]
\[ \text{ins} \; x \; (y:ys) \mid x \leq y = x : y : ys \]
\[ \mid \text{otherwise} = \]

Example

\[ \text{ascending} :: \text{Ord} \; a \Rightarrow [a] \rightarrow \text{bool} \]
\[ \text{ascending} \; [] = \text{True} \]
\[ \text{ascending} \; [\_] = \text{True} \]
\[ \text{ascending} \; (x : y : zs) = \]
Beyond primitive recursion: Complex patterns

Example

ascending :: Ord a => [a] -> bool
ascending [] = True
ascending [_] = True
ascending (a : b : zs) = a <= b && ascending (b : zs)

Beyond primitive recursion: Multiple arguments

Example

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

General recursion: Quicksort

Example

quicksort :: Ord a => [a] -> [a]
pr pic = putStrLn(render pic)

-- Chessboards

lapnkp0d1d:Code nlpkows $ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :,? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :l Pictures
[1 of 1] Compiling Main
         ( Pictures.hs, interpreted )
Ok, modules loaded: Main.
(Main) quickCheck prop_aboveFlipV
Loading package array-0.4.0.1 ... linking ... done.
Loading package deepseq-1.3.0.1 ... linking ... done.
Loading package old-locale-1.0.0.5 ... linking ... done.
Loading package time-1.4.0.1 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
Loading package containers-0.5.0.0 ... linking ... done.
Loading package pretty-1.1.1.0 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package QuickCheck-2.6 ... linking ... done.
+++ Ok, passed 100 tests.
(Main)