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

⇒ length :: [T] -> Int for all types T
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 \( T[U/a] = \) "\( T \) with \( a \) replaced by \( U \)"

Example: \((a \to a)[\text{Bool}/a] = \text{Bool} \to \text{Bool}\)

(Often called *ML-style polymorphism*)

---

Defining polymorphic functions

\[
\begin{align*}
\text{id} & : : a \to a \\
\text{id} \ x & = \ x \\
\text{fst} \ (x,y) & = \ x
\end{align*}
\]
Defining polymorphic functions

\[
\begin{align*}
\text{id} &: \ a \to a \\
\text{id} \ x &= x \\
\text{fst} &: (a,b) \to a \\
\text{fst} \ (x,y) &= x \\
\text{swap} &: (a,b) \to (b,a) \\
\text{swap} \ (x,y) &= (y,x) \\
\text{silly} &: \text{Bool} \to a \to \text{Char} \\
\text{silly} \ x \ y &= \text{if} \ x \ \text{then} \ 'c' \ \text{else} \ 'd' \\
\text{silly2} &: \text{Bool} \to \text{Bool} \to \text{Bool} \\
\text{silly2} \ x \ y &= \text{if} \ x \ \text{then} \ x \ \text{else} \ y
\end{align*}
\]
Polymorphic list functions from the Prelude

```haskell
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

```haskell
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

```haskell
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

```haskell
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]
replicate 3 'c' = "ccc"

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]

custom

custom :: [Int] -> [Int]
custom (x : xs) = if x > 0 then tail xs else init xs

-- A property:
prop_custom :: [Int] -> Bool
prop_custom xs = custom xs == tail (init xs)

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

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

custom :: [Int] -> [Int]
custom (x : xs) = if x > 0 then tail xs else init xs

-- A property:
prop_custom :: [Int] -> Bool
prop_custom xs = custom xs == tail (init xs)
Polymorphic list functions from the *Prelude*

\[
\text{concat} :: [[a]] \to [a] \\
\text{concat} [[1, 2], [3, 4], [0]] = [1, 2, 3, 4, 0] \\
\]

\[
\text{zip} :: [a] \to [b] \to [(a,b)] \\
\text{zip} [1,2] "ab" = [(1, 'a'), (2, 'b')] \\
\]

\[
\text{concat} :: [[a]] \to [a] \\
\text{concat} [[1, 2], [3, 4], [0]] = [1, 2, 3, 4, 0] \\
\]

\[
\text{zip} :: [a] \to [b] \to [(a,b)] \\
\text{zip} [1,2] "ab" = [(1, 'a'), (2, 'b')] \\
\]

\[
\text{unzip} :: \\
\text{unzip} [(1, 'a'), (2, 'b')] = ([1,2], "ab") \\
\]
Polymorphic list functions from the Prelude

```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) == (xs, ys)

Polymorphic list functions from the Prelude

```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) == (xs, ys)
Haskell libraries

- Prelude and much more

Hoogle — searching the Haskell libraries

Hoogle is a Haskell API search engine, which allows you to search many standard Haskell libraries by either function name, or by approximate type signature.

Example searches:
- map
- (a -> b) -> (a -> b)
- Ord a => a -> [a]
- Data.Map.insert

Enter your own search at the top of the page.

The Hoogle manual contains more details, including further details on search queries, how to install Hoogle as a command line application and how to integrate Hoogle with Firefox/Emacs/Vim etc.

I am very interested in any feedback you may have. Please email me, or add an entry to my bug tracker.

© Neil Mitchell 2004-2012, version 4.2.16
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
- **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
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
```
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**

```haskell
(+) :: 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

- 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 possess
  (==) :: Eq a => a -> a -> Bool
Other important type classes

- The class `Eq` of *equality types*, i.e. types that possess
  
  \[ (==) : \text{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]

  \[ (/=) : \text{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]

  Most types are of class `Eq`. Exception: functions

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

  \[ (<=) : \text{Ord}\ a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]

  More on type classes later. Don't confuse with OO classes.
null :: Eq a => [a] -> Bool
null xs = xs == []

Why?
== on [a] may call == on a
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

Conditional properties have result type Property

Example
QuickCheck does not find a counterexample to
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

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]

uarr :: Picture
uarr =
  [" # ",
   " ### ",
   "####",
   " # ",
   " # "]
4.3 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]
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 = [ 11 ++ 12 | (11,12) <- zip pic1 pic2]
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 = putStrLn (render pic)
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)

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

Loading package base ... linking ... done.
[1 of 1] Compiling Main
  ( Pictures.hs, interpreted )
OK, modules loaded: 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-1.2.1.0 ... 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>
Exercise

Ensure that the lower left square of the chessboard $n$ is always black.

4.4 Pattern matching

Every list can be constructed from $[]$
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>2 : 3 : []</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 \((::) : a \rightarrow [a] \rightarrow [a]\)

<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>
</tbody>
</table>

Note: \(x : y : zs = x : (y : zs)\)
\((::)\) associates to the right
Every list is either
\[ \begin{array}{c}
\emptyset \quad \text{or of the form} \\
x : \ x_\infty \quad \text{where} \\
\begin{align*}
\quad x \quad &\text{is the } \textit{head} \quad \text{(first element, Kopf), and} \\
\quad x_\infty \quad &\text{is the } \textit{tail} \quad \text{(rest list, Rumpf)}
\end{align*}
\end{array} \quad \begin{array}{c}
\emptyset \quad \text{or of the form} \\
x : \ x_\infty \quad \text{where} \\
\begin{align*}
\quad x \quad &\text{is the } \textit{head} \quad \text{(first element, Kopf), and} \\
\quad x_\infty \quad &\text{is the } \textit{tail} \quad \text{(rest list, Rumpf)}
\end{align*}
\end{array} \]

\[ \begin{array}{c}
\emptyset \quad \text{and } (\cdot) \quad \text{are called } \textit{constructors} \\
because every list can be \textit{constructed uniquely} \quad \text{from them.}
\end{array} \quad \begin{array}{c}
\emptyset \quad \text{and } (\cdot) \quad \text{are called } \textit{constructors} \\
because every list can be \textit{constructed uniquely} \quad \text{from them.}
\end{array} \]

\[ \begin{array}{c}
\text{Every non-empty list can be decomposed uniquely into head and tail.}
\end{array} \quad \begin{array}{c}
\text{Every non-empty list can be decomposed uniquely into head and tail.}
\end{array} \]

Therefore these definitions make sense:
\[ \begin{align*}
\text{head} \ (x : \ x_\infty) \ &= \ x \\
\text{tail} \ (x : \ x_\infty) \ &= \ x_\infty
\end{align*} \]
(++) 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:
\text{nonsense (xs ++ ys) = length xs - length ys}

---

Patterns

Patterns are expressions consisting only of constructors and variables.

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.

\[\Rightarrow\]
Every non-empty list can be decomposed uniquely into head and tail.

Therefore these definitions make sense:
\[
\text{head (x : xs) = x}
\]
\[
\text{tail (x : xs) = xs}
\]
Patterns

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

→ 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 \((p_1, ..., p_n)\) 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 \((p_1, ..., p_n)\) where each \( p_i \) is a pattern
- a constructor pattern \( C \ p_1 \ ... \ p_n \)
  where \( C \) is a constructor and each \( p_i \) is a pattern

(++) is not a constructor:

\[ [1,2,3] \text{ is not uniquely constructable with } (++) : \]
\[ [1,2,3] = [1] ++ [2,3] = [1,2] ++ [3] \]
Patterns

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

→ 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

\[
\begin{align*}
\text{head} &: [a] \to a \\
\text{head} (x : _) &= x \\
\text{tail} &: [a] \to [a] \\
\text{tail} (_ : xs) &= xs \\
\text{null} &: [a] \to \text{Bool} \\
\text{null} [] &= \text{True} \\
\text{null} (_ : _) &= \text{False}
\end{align*}
\]

Function definitions by pattern matching

\[
\begin{align*}
f \ pat_1 &= e_1 \\
\vdots \\
f \ pat_n &= e_n
\end{align*}
\]
Function definitions by pattern matching

\[
\begin{align*}
  f \ pat_1 &= e_1 \\
  \vdots \\
  f \ pat_n &= e_n \\
\end{align*}
\]

If \( f \) has multiple arguments:

\[
\begin{align*}
  f \ pat_{11} \ldots \ pat_{1k} &= e_1 \\
  \vdots \\
\end{align*}
\]

Conditional equations:

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

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

Example (contrived)

\[
\begin{align*}
  \text{true12 \ (True : True : _) &= True} \\
  \text{true12 \ _ &= False} \\
\end{align*}
\]
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

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

Example

\[
\text{length } [] = 0 \\
\text{length } (_ : xs) = \text{length } xs + 1
\]
4.5 Recursion over lists

Example

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (_:xs) &= \text{length } xs + 1 \\
\text{reverse } [] &= [] \\
\text{reverse } (x:x) &= \text{reverse } xs ++ [x]
\end{align*}
\]

\[
\begin{align*}
\text{sum} :: \text{Num } a => [a] \rightarrow a \\
\text{sum } [] &= 0 \\
\text{sum } (x:xs) &= x + \text{sum } xs
\end{align*}
\]

---

**Primitive recursion** on lists:

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

*Primitive recursion* on lists:

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

- \textit{base}: no call of \( f \)
- \textit{rec}: only call(s) \( f \; xs \)

**Example**

\[
\text{concat} :: [[a]] \to [a]
\]

\[
\begin{align*}
\text{concat} \; [] &= [] \\
\text{concat} \; (xs : xss) &= \text{concat} \; [xs : \text{concat} \; xss]
\end{align*}
\]
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)

Finding primitive recursive definitions

Example

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

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

Insertion sort

Example

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

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

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

Example

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

ins :: a -> [a] -> [a]
Example

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

\[\text{ins} :: a \to [a] \to [a]\]
\[\text{ins } x \ [] = [x]\]
\[\text{ins } x \ (y:ys) | x \leq y = x : y : ys\]
\[\text{otherwise} = y : \text{ins } x \ ys\]

Example

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

\[\text{ins} :: a \to [a] \to [a]\]
\[\text{ins } x \ [] = [x]\]
\[\text{ins } x \ (y:ys) | x \leq y = x : y : ys\]
\[\text{otherwise} = y : \text{ins } x \ ys\]

Example

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

\[\text{ins} :: \text{Ord } a \Rightarrow a \to [a] \to [a]\]
\[\text{ins } x \ [] = [x]\]
\[\text{ins } x \ (y:ys) | x \leq y = x : y : ys\]
\[\text{otherwise} = y : \text{ins } x \ ys\]

Beyond primitive recursion: Complex patterns

Example

\[\text{ascending} :: \text{Ord } a \Rightarrow [a] \to \text{bool}\]
Beyond primitive recursion: Complex patterns

Example

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

Beyond primitive recursion: Complex patterns

Example

```haskell
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

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

The most general type:

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

where `a` is a type variable

Aber dir fehlt Fachwissen Kontakte Die Idee
Bitbanana

Wir wollen, dass die Welt eure Ideen sieht!

Wir bieten dir...

Mitmacher

Wir bieten dir...

Support

Wir bieten dir...

Publishing
Und DU machst den meisten Gewinn.
Egal wie es ausgeht, du siehst als erstes Geld (70% der Umsätze) und trägst 0% Risiko.

Be part of the team!

www.bitbanana.com