Polymorphism and higher-order functions

Fast Track to Haskell

Andres Löh, Edsko de Vries

8–9 April 2013 — Copyright © 2013 Well-Typed LLP

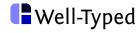




Type inference

- ► The compiler will infer types for expressions, and for constant and function declarations automatically. Type annotations are rarely required.
- ► Type annotations can always be provided and will be checked for correctness by the compiler.
- ► Type signatures for top-level declarations are considered good style. They serve as invaluable machine-checked interface documentation.
- ➤ You can use GHC(i) to obtain inferred types. Use : t often, but also try to train your own type inference capabilities over time it will help you to understand errors with less effort.





One function, several types

Some Haskell expressions and functions can have more than one type.

Example:

$$fst(x,y) = x$$

Possible type signatures (all would work):

```
\begin{array}{l} \text{fst} :: (a,a) \to a \\ \text{fst} :: (\text{Int},a) \to \text{Int} \\ \text{fst} :: (\text{Int},\text{Int}) \to \text{Int} \\ \text{fst} :: (a,b) \to a \\ \text{fst} :: (\text{Int},\text{Char}) \to \text{Int} \end{array}
```

Is one of these clearly the "best" choice?



3 - Parametric polymorphism - Polymorphism and higher-order functions

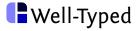


Most general type

Haskell's type system is designed such that (ignoring some language extensions) each term has a most general type:

- ▶ the most general type allows the most flexible use;
- all other types the term has can be obtained by instantiating the most general type, i.e., by substituting type variables with type expressions.





Instantiating types

The type signature

$$fst::(a,b)\to a$$

declares the most general type for fst. Types like

```
 \begin{split} \text{fst} &:: (a, a) \to a \\ \text{fst} &:: (\text{Int}, \text{Char}) \to \text{Int} \\ \text{fst} &:: (a \to \text{Int} \to b, c) \to a \to \text{Int} \to b \end{split}
```

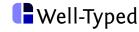
are instantiations of the most general type.

Type inference will always infer the most general type!

(So sometimes it's worth asking GHC about the inferred type of a function, even if you started by providing a type signature, and you might be surprised that the inferred type is more general than what you had specified.)



5 - Parametric polymorphism - Polymorphism and higher-order functions



No run-time type information

Haskell terms carry no type information at run-time.

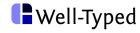
Remember

You can only ever use a term in the ways its type dictates.

Example:

$$\begin{split} &\text{fst} :: (a,b) \to a \\ &\text{fst} \ (x,y) = x \\ &\text{restrictedFst} :: (Int,Int) \to Int \\ &\text{restrictedFst} = fst & -- ok \\ &\text{newFst} :: (a,b) \to a \\ &\text{newFst} = restrictedFst & -- type error! \end{split}$$





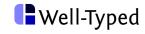
Parametric polymorphism

- ► A type with type variables (but no class constraints) is called (parametrically) polymorphic.
- ► Type variables can be instantiated to any type expression, but several occurrences of the same variable have to be the same type.
- ► If a function argument has polymorphic type, then you know nothing about it. No pattern matching is possible. You can only pass it on.
- ► If a function result has polymorphic type, then (except for undefined and error) you can only try to build one from the function arguments.

Let us look at examples.



7 – Parametric polymorphism – Polymorphism and higher-order functions



Example

How many functions can you think of that have this type:

$$(Int, Int) \rightarrow (Int, Int)$$

And of this one?

$$(a,a) \rightarrow (a,a)$$

And of this one?

$$(a,b) \rightarrow (b,a)$$

(Thanks to Doaitse Swierstra for the example.)



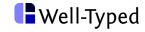


Parametricity

- In general, parametric polymorphism severely restricts how a function can be implemented.
- ➤ So if the functionality you're trying to implement is quite general, this is a good thing, because it really prevents you from making errors.
- ► Conversely, if you see a function with parametrically polymorphic type, you know that it cannot look at the polymorphic values.
- ▶ By looking at polymorphic types alone, one can obtain non-trivial properties of the functions. (This is sometimes called "parametricity".)
- For example, map :: (a → b) → [a] → [b] must produce a list in which all elements are obtained by applying the given function to elements of the original list but we don't know how long the resulting list is, or in which order the elements occur.



9 - Parametric polymorphism - Polymorphism and higher-order functions



A common pitfall: who gets to choose

Sometimes, it may be tempting to write a program like the following:

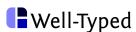
```
parse :: String \rightarrow a parse "False" = False parse "0" = 0 ...
```

What is wrong here?

For polymorphic types, it is always the caller who gets to choose at which type the function should be used.

A function with polymorphic result type (but no polymorphic arguments) is impossible to write without either looping or causing an exception: we'd have to produce a value that belongs to every type imaginable!





What if we need to return values of different types?

Option 1: use Either:

```
data Either a b = Left a | Right b

parse :: String \rightarrow Either Bool Int

parse "False" = Left False

parse "0" = Right 0
```

Option 2: define your own datatype.

```
data Value = VBool Bool | VInt Int

parse :: String → Value

parse "False" = VBool False

parse "0" = VInt 0
```

The second option is quite common in libraries that interface with dynamically typed languages (SQL, JSON, ...).



11 - Parametric polymorphism - Polymorphism and higher-order functions



Reusing code, reusing names

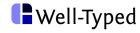
Parametric polymorphism

Allows you to use the same implementation in as many contexts as possible.

Overloading (ad-hoc polymorphism)

Allows you to use the same function name in different contexts, but with different implementations for different types.





Type classes

A type class defines an interface that can be implemented by potentially many different types.

Example:

```
class Eq a where

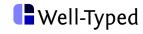
(==) :: a \rightarrow a \rightarrow Bool

(\neq) :: a \rightarrow a \rightarrow Bool
```

Using **instance** declarations, we can explain how a certain type (or types of a certain shape) implement the interface.



13 - Overloading - Polymorphism and higher-order functions



Instances

```
instance Eq Bool where

False == False = True

True == True = True

_- == _- = False

x \neq y = not (x == y)
```

```
instance Eq a \Rightarrow Eq [a] where

[] == [] = True

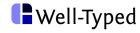
(x:xs) == (y:ys) = x == y && xs == ys

== = = False

xs \neq ys = not (xs == ys)
```

We use equality on a while defining equality on [a].





Class constraints

All the instances of a given type class specify a subset of all the Haskell types, namely the subset that implements the class interface.

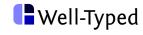
In type signatures, class constraints specify that a type variable can only be instantiated to types belonging to a certain class:

```
(==) :: Eq \ a \Rightarrow a \rightarrow a \rightarrow Bool
```

Read: "Given that a is an instance of Eq, the function has the type $a \rightarrow a \rightarrow Bool$."



15 - Overloading - Polymorphism and higher-order functions



Overloading and inference

Not only class methods, but also functions that directly or indirectly use class methods can have types with constraints. Example:

```
\begin{array}{ll} \text{allEqual} :: \mathsf{Eq} \ \mathsf{a} \Rightarrow [\mathsf{a}] \to \mathsf{Bool} \\ \text{allEqual} \ [] &= \mathsf{True} \\ \text{allEqual} \ [\mathsf{x}] &= \mathsf{True} \\ \text{allEqual} \ (\mathsf{x} : \mathsf{y} : \mathsf{ys}) = \mathsf{x} = \mathsf{y} \ \&\& \ \mathsf{allEqual} \ (\mathsf{y} : \mathsf{ys}) \end{array}
```

Also recall elem or lookup.

Class constraints will be automatically inferred by the compiler.





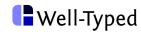
Several class constraints

There can be multiple constraints on a function, and they can apply to several variables:

```
example :: (Eq a, Eq b, Show a) \Rightarrow (a, b) \rightarrow (a, b) \rightarrow String
example (x1, y1) (x2, y2)
   | x1 = x2 \&\& y1 = y2 = show x1
   otherwise = "different"
```



17 - Overloading - Polymorphism and higher-order functions



Default definitions

class Eq a where

$$(==) :: a \rightarrow a \rightarrow Bool$$

 $(\neq) :: a \rightarrow a \rightarrow Bool$

$$x = y = not (x \neq y)$$

$$x \neq y = not (x == y)$$

Now:

instance Eq Bool where

And (\neq) will work automatically.

Careful: if you provide neither (==) nor (\neq) , you won't get a ກວດmplaint, but both functions will loop. ■ Well-Typed

18 - Overloading - Polymorphism and higher-order functions

Classes are not types!

Note that

$$\begin{array}{l} f:: Eq \rightarrow Eq \rightarrow Bool \\ f:: Eq \ a \rightarrow Eq \ a \rightarrow Bool \end{array}$$

are both invalid. Classes appear in constraints! Also note that the type

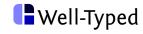
$$\mathsf{Eq}\; a \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}$$

forces both arguments to be of the same type. You cannot pass two different types that are both an instance of Eq – that would require a function of type

$$(\mathsf{Eq}\; a, \mathsf{Eq}\; b) \Rightarrow a \to b \to \mathsf{Bool}$$



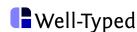
19 - Overloading - Polymorphism and higher-order functions





- ► For equality and inequality.
- ► Note that equality in Haskell is structural equality. There is no "object identity", and no pointer equality.
- Supported by most datatypes, such as numbers, characters, tuples, lists, Maybe, Either, ...
- Not supported for function types.







For comparisons between values of the same type.

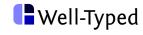
```
class Eq a \Rightarrow Ord\ a\ where
compare :: a \rightarrow a \rightarrow Ordering
(<) :: a \rightarrow a \rightarrow Bool
(\leqslant) :: a \rightarrow a \rightarrow Bool
(>) :: a \rightarrow a \rightarrow Bool
(\geqslant) :: a \rightarrow a \rightarrow Bool
(\geqslant) :: a \rightarrow a \rightarrow Bool
max :: a \rightarrow a \rightarrow a
min :: a \rightarrow a \rightarrow a
```

Several default definitions – you'd typically define just compare or (\leq) .

```
data Ordering = LT | EQ | GT
```



21 - Important classes - Polymorphism and higher-order functions



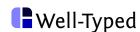
Superclasses

```
class Eq a \Rightarrow Ord a where ...
```

The condition indicates that Eq is a superclass of Ord:

- ► You cannot give an instance for Ord without first providing an instance to Eq.
- ► Conversely, a constraint Ord $a \Rightarrow ...$ on a function implies Eq a. In other words, $(\text{Ord } a, \text{Eq } a) \Rightarrow ...$ is equivalent to Ord $a \Rightarrow ...$.





Show

class Show a where

show :: $a \rightarrow String$

showsPrec :: Int \rightarrow a \rightarrow ShowS

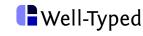
showList $:: [a] \rightarrow ShowS$

The most important method is **show**:

- used to produce a human-readable String -representation of a value;
- ▶ it is sufficient to define show in new instances, as the others have default definitions;
- the other two functions can be used to more efficiently and beautifully implement show internally (for example, remove unnecessary parentheses);
- also used by GHCi to print result values of evaluated terms;
- once again, function types are not an instance.



23 - Important classes - Polymorphism and higher-order functions



Read

class Read a where

readsPrec :: Int \rightarrow ReadS a

readList :: ReadS [a]

Most often, the derived function read is used:

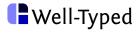
read :: Read $a \Rightarrow String \rightarrow a$

Tries to interpret a given String (such as produced by show) as a value of a type.

How the value is interpreted is statically determined by the context:

read "1" + 2 -- used as a number, parsed as a number not (read "False") -- used as a Bool, parsed as a Bool





Unresolved overloading

The following function produces an error (not in GHCi, but if placed in a file):

```
strange x = \text{show (read } x)
```

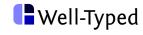
The error will say something about an "ambiguous type variable" and mention constraints for Read and Show.

Can you imagine what the problem is?

The x is a String which is then parsed into something by read. But what type should it be parsed at? The context does not tell, because the result is passed to Show, which is also overloaded.



25 - Important classes - Polymorphism and higher-order functions



Manually resolving overloading

This works:

```
\begin{array}{l} \text{strange} :: \mathsf{String} \to \mathsf{String} \\ \text{strange} \ x = \mathsf{show} \ (\mathsf{read} \ x :: \mathsf{Bool}) \end{array}
```

Or this:

```
\begin{array}{l} \text{strange} :: String \rightarrow String \\ \text{strange} \ x = \text{show (read } x :: Int) \end{array}
```

But note that the choice of intermediate type does make a difference!

In general, if several overloaded functions are combined such that the resulting type does not mention any overloaded variables anymore, you have to specify the intermediate types manually to help the type checker resolve the overloading.





deriving

For a limited number of type classes (but in particular Eq, Ord, Show, Read), the Haskell compiler has a built-in algorithm to derive an instance for nearly any datatype.

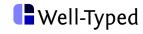
```
data Tree a = Leaf a | Node (Tree a) (Tree a)
deriving (Eq, Ord, Show, Read)
```

Defines the Tree datatype of binary trees together with suitable instances:

- equality is always deep and structural;
- ordering depends on the order of constructors;
- Show and Read assume the natural human-readable Haskell string representation.



27 - Important classes - Polymorphism and higher-order functions



Numeric types and classes

There are several numeric types and classes in Haskell:

type instance of
Int Num Integral
Integer Num Integral

Float Num Fractional Floating RealFrac
Double Num Fractional Floating RealFrac

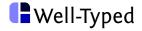
Rational Num Fractional

The class Num is a superclass of Integral.

The class Fractional is a superclass of Floating.

- Whereas Int is bounded, Integer is unbounded (bounded by memory only).
- ► A Double is usually of higher precision than a Float.
- The datatype Rational is for fractions.





Operations on numbers

Most operations on numbers and even numeric literals are overloaded:

```
(+) :: (Num \ a) \Rightarrow a \rightarrow a \rightarrow a

(-) :: (Num \ a) \Rightarrow a \rightarrow a \rightarrow a

(*) :: (Num \ a) \Rightarrow a \rightarrow a \rightarrow a
```

```
1 :: (Num a) \Rightarrow a -- overloaded literals
1.2 :: (Fractional a) \Rightarrow a -- overloaded literals
```

```
(/) :: (Fractional a) \Rightarrow a \rightarrow a \rightarrow a mod :: (Integral a) \Rightarrow a \rightarrow a \rightarrow a div :: (Integral a) \Rightarrow a \rightarrow a \rightarrow a \rightarrow a sin :: (Floating a) \Rightarrow a \rightarrow a log :: (Floating a) \Rightarrow a \rightarrow a
```



29 - Numbers - Polymorphism and higher-order functions



No automatic coercion

We can use overloaded functions at different types:

But there is no implicit coercion:

Question

Why is 3.2 * 2 ok, but not 3.2 * (5 'div' 2)?

Because $2 :: (Num a) \Rightarrow a$, but $(5 'div' 2) :: (Integral a) \Rightarrow a$.





Converting between numeric types

From an integral type to another:

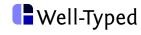
```
fromIntegral :: (Integral a, Num b) \Rightarrow a \rightarrow b
```

From a fractional type to an integral:

```
round :: (RealFrac a, Integral b) \Rightarrow a \rightarrow b -- nearest even floor :: (RealFrac a, Integral b) \Rightarrow a \rightarrow b ceiling :: (RealFrac a, Integral b) \Rightarrow a \rightarrow b
```



31 – Numbers – Polymorphism and higher-order functions



Recap: Haskell types

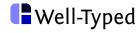
Question

Which Haskell types have we seen so far?

Note that there are:

- ▶ truly built-in types such as Int, Char or functions (\rightarrow) ;
- types that could be defined by data, but support special syntax, such as tuples and lists;
- ► types that are defined in the basic libraries, but could just as well have been defined by you (Bool, Maybe, Either);
- types that are actually just synonyms for other types (String).





Use GHCi for information

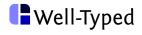
A quick way to remind yourself of the definition of a datatype or class is by using :i or :info in GHCi:

- Shows the data declaration and class instances for datatypes.
- ► Shows the **class** declaration and instances for classes.
- ► Shows a (partial) data declaration for constructors.
- ► Shows a (partial) **class** declaration for methods.
- Shows the type signature for functions and constants.

Sometimes, GHCi lets you glimpse at internal implementation details that are – at this point – difficult to understand (such as for : i Int).



33 - Numbers - Polymorphism and higher-order functions



Functions, functions

A function parameterized by another function or returning a function is called a higher-order function.

Currying

Strictly speaking, every curried function in Haskell is a function returning another function:

```
elem :: Eq a \Rightarrow a \rightarrow ([a] \rightarrow Bool)
elem 3 :: (Eq a, Num a) \Rightarrow [a] \rightarrow Bool
```





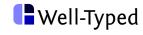
Filtering and mapping

Two of the most useful list functions are higher-order, as they each take a function as an argument:

The use of a function $a \to Bool$ to express a predicate is generally common. And mapping a function over a data structure is an operation that isn't limited to lists.



35 - Higher-order functions - Polymorphism and higher-order functions



Overloading vs. parameterization

Consider:

```
\begin{array}{ll} \text{sort} & :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\ \text{sortBy} :: (a \rightarrow a \rightarrow \text{Ordering}) \rightarrow [a] \rightarrow [a] \end{array}
```

Both functions are rather similar:

- the first takes the comparison function to use from the instance declaration for the element type of the list;
- the second is passed an explicit comparison function.

Using an overloaded function is a bit more convenient, but using sortBy is a bit more flexible.

Interestingly, GHC implements overloaded functions by passing type class "dictionaries" as additional arguments.



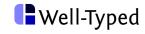


Excursion: performance impact of overloading

- ► You pay no price whatsoever for parametric polymorphism.
- Overloaded functions get extra arguments at runtime.
 There is a slight performance penalty for that.
- ► Only overloaded functions get extra arguments remember that there is no general run-time type information!
- ▶ It is possible to instruct GHC to generate specialized versions for overloaded functions at particular types, thereby eliminating the run-time overhead.
- ► GHC also has a relatively aggressive inliner. Inlining overloaded functions can also remove the overhead, much like specialization.



37 - Higher-order functions - Polymorphism and higher-order functions



Function composition

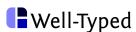
One of the most ubiquitous higher-order functions is function composition:

$$(\circ) :: \dots$$

 $(f \circ g) x = f (g x)$

For once – rather than starting from a type – let's infer the type from the code.





One of the most ubiquitous higher-order functions is function composition:

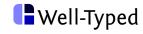
$$(\circ) :: \ldots \to \ldots \to \ldots \to \ldots$$

 $(f \circ g) x = f (g x)$

It's apparently a curried function taking three arguments f, g and x.



39 – Higher-order functions – Polymorphism and higher-order functions



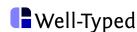
Function composition

One of the most ubiquitous higher-order functions is function composition:

$$\begin{array}{l} (\circ) :: (\ldots \to \ldots) \to (\ldots \to \ldots) \to \ldots \\ (f \circ g) \ x = f \ (g \ x) \end{array}$$

Both f and g are applied to something, so they must be functions.





One of the most ubiquitous higher-order functions is function composition:

$$\begin{array}{l} (\circ) :: (\ldots \to \ldots) \to (\ldots \to \ldots) \to a \to \ldots \\ (f \circ g) \ x = f \ (g \ x) \end{array}$$

No requirements seem to be made about the type of x, except that its passed to g, so let's assume a type variable here . . .



41 – Higher-order functions – Polymorphism and higher-order functions



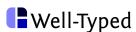
Function composition

One of the most ubiquitous higher-order functions is function composition:

$$(\circ)::(\ldots\to\ldots)\to(a\to\ldots)\to a\to\ldots$$
 $(f\circ g)\ x=f\ (g\ x)$

... which then should be the source type of g as well.





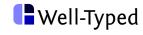
One of the most ubiquitous higher-order functions is function composition:

$$\begin{array}{l} (\circ) :: (b \rightarrow \ldots) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow \ldots \\ (f \circ g) \ x = f \ (g \ x) \end{array}$$

The target type of g should match the source type of f.



43 - Higher-order functions - Polymorphism and higher-order functions



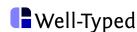
Function composition

One of the most ubiquitous higher-order functions is function composition:

$$\begin{array}{l} (\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\ (f \circ g) \ x = f \ (g \ x) \end{array}$$

The target type of f is also the type of the overall result.





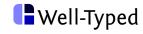
One of the most ubiquitous higher-order functions is function composition:

$$\begin{array}{l} (\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ (f \circ g) \ x = f \ (g \ x) \end{array}$$

Putting extra parentheses in the type may make it more obvious that we are indeed composing two matching functions.



45 - Higher-order functions - Polymorphism and higher-order functions



Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

We start by generating all numbers (lazy evaluation in action).





Composing functions

We can often build functions from existing functions simply by composing them.

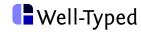
Example: Computing the first 100 odd square numbers.

```
example :: [Int]  \text{example} = \qquad \qquad \text{map } (\lambda x \to x * x) \quad [1 \ldots]
```

We use map to compute the square numbers. Note that map and filter are often used with anonymous functions.



47 - Higher-order functions - Polymorphism and higher-order functions



Composing functions

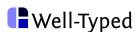
We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int] 
example = ( filter odd \circ map (\lambda x \rightarrow x * x)) [1..]
```

We use function composition composition (and partial application) to subsequently filter the odd square numbers.





Composing functions

We can often build functions from existing functions simply by composing them.

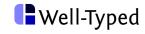
Example: Computing the first 100 odd square numbers.

```
example :: [Int] example = (take 100 \circ filter odd \circ map (\lambda x \rightarrow x * x)) [1 . .]
```

Finally, we use composition again to take the first 100 elements of this list.



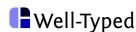
49 – Higher-order functions – Polymorphism and higher-order functions



Composition as a design pattern

- Function composition gives you a way to split one programming problem into several, possibly smaller, programming problems.
- ► In general, higher-order functions are part of your toolbox for attacking programming problems. Recognizing something as a map or filter is also useful.
- Of course, you should never forget the standard design principle of following the datatype structure as a good way of defining most functions, if applying a higher-order function fails.



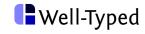


Lessons

- ► Function composition is a bit like the functional semicolon. It allows us to decompose larger tasks into smaller ones.
- ► Lazy evaluation allows us to separate the generation of possible results from selecting interesting results. This allows more modular programs in many situations.
- Partial application and anonymous functions help to keep such composition chains concise.



51 - Higher-order functions - Polymorphism and higher-order functions



Flipping a function

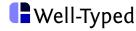
If you want to change the order of arguments of a two-argument curried function, you can use

$$\begin{aligned} & \text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c \\ & \text{flip } f \ x \ y = f \ y \ x \end{aligned}$$

Example use:

foreach = flip map
example = foreach [1,2,3] (
$$\lambda x \rightarrow x * x$$
)





Currying and uncurrying

Sometimes, you end up with a pair and want to apply a function to it that typically (in Haskell) is in curried form. Fortunately, we can convert between curried and uncurried form easily:

```
uncurry :: (a \rightarrow b \rightarrow c) \rightarrow (a,b) \rightarrow c

uncurry f(x,y) = f x y

curry :: ((a,b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c

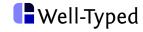
curry f(x,y) = f(x,y)
```

Example:

```
map (uncurry (*)) (zip [1..3] [4..6])
```



53 - Operating on functions - Polymorphism and higher-order functions



Abstraction

One of the strengths of Haskell's flexibility with functions is that they really allow to abstract from reoccuring patterns and thereby save code.

The standard design principle for lists we've been using all the time works as follows:

```
\begin{array}{ll} \text{function} :: [\text{someType}] \rightarrow \text{someResult} \\ \text{function} \: [] &= \dots &-- \text{code} \\ \text{function} \: (x:xs) = \dots &-- \text{code that can use } x \text{ and function } xs \end{array}
```



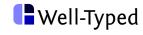


```
\begin{array}{ll} \text{function} :: [\text{someType}] \rightarrow \text{someResult} \\ \text{function} \: [] &= \dots & \text{--code} \\ \text{function} \: (x:xs) = \dots & \text{--code that can use } x \: \text{ and function } xs \end{array}
```

We have two interesting positions where we have to fill in situation-specific code. Let's abstract!



55 - Capturing design patterns - Polymorphism and higher-order functions

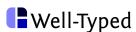


From an informal pattern to a function

```
\begin{array}{ll} \text{function} :: [\text{someType}] \rightarrow \text{someResult} \\ \text{function} \: [] &= \text{nil} \\ \text{function} \: (x:xs) = \text{cons} \: x \: (\text{function} \: xs) \end{array}
```

- We give names to the cases that correspond to the constructors.
- ► The case cons can use x and function xs, so we turn it into a function.
- ► At the moment, this is not a valid function, because nil and cons come out of nowhere but we can turn them into parameters of function!





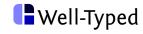
```
\begin{array}{ll} \text{function} :: \ldots \to \ldots \to [\text{someType}] \to \text{someResult} \\ \text{function cons nil } [] &= \text{nil} \\ \text{function cons nil } (x:xs) = \text{cons } x \text{ (function cons nil } xs) \end{array}
```

We now have to look at the types of cons and nil:

- nil is used as a result, so nil :: someResult ;
- cons takes a list element and a result to a result, so cons :: someType → someResult → someResult.



57 - Capturing design patterns - Polymorphism and higher-order functions

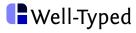


From an informal pattern to a function

```
\begin{array}{ll} \text{function} :: & (\text{someType} \rightarrow \text{someResult} \rightarrow \text{someResult}) \\ & \rightarrow \text{someResult} \\ & \rightarrow [\text{someType}] \rightarrow \text{someResult} \\ \text{function cons nil } [] & = \text{nil} \\ \text{function cons nil } (x:xs) = \text{cons } x \text{ (function cons nil } xs) \\ \end{array}
```

We can give shorter names to someType and someResult





```
function :: (a \rightarrow r \rightarrow r) \rightarrow r \rightarrow [a] \rightarrow r
function cons nil [] = nil
function cons nil (x : xs) = cons x (function cons nil xs)
```

This function is called foldr ...



59 - Capturing design patterns - Polymorphism and higher-order functions

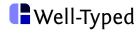


From an informal pattern to a function

```
foldr :: (a \rightarrow r \rightarrow r) \rightarrow r \rightarrow [a] \rightarrow r
foldr cons nil [] = nil
foldr cons nil (x : xs) = cons x (foldr cons nil xs)
```

We could equivalently define it using where ...

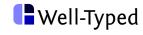




The arguments cons and nil never change while traversing the list, so we can just refer to them in the local definition go, without explicitly passing them around.



61 - Capturing design patterns - Polymorphism and higher-order functions



Using foldr

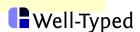
```
\begin{array}{ll} \text{length} :: [a] \rightarrow \text{Int} \\ \text{length} \ [] &= 0 \\ \text{length} \ (x:xs) = 1 + \text{length} \ xs \end{array}
```

length = foldr (
$$\lambda x r \rightarrow 1 + r$$
) 0

or (using const and an operator section)

$$length = foldr (const (1 +)) 0$$





Examples of using foldr

```
(++) :: [a] \rightarrow [a] \rightarrow [a]
(++) xs ys = foldr (:) ys xs
filter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]
filter p = foldr (\lambda x r \rightarrow \textbf{if } p x \textbf{then } x : r \textbf{else } r) []
map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
map f = foldr (\lambda x r \rightarrow f x : r) []
```

```
and :: [Bool] \rightarrow Bool
and = foldr (&&) True
any :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool
any p = foldr (\lambda x r \rightarrow p x || r) False
```

And many more.



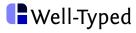
63 - Capturing design patterns - Polymorphism and higher-order functions



The role of foldr

- When a list function is easy to express using foldr, then you should.
- Makes it immediately recognizable for the reader that it follows the standard design principle.
- ► Some functions can be expressed using foldr, but that does not necessarily make them any clearer. In such cases, aim for clarity.





Accumulating parameter pattern

```
reverse :: [a] \rightarrow [a]

reverse = go[]

where

go :: [a] \rightarrow [a] \rightarrow [a]

go acc[] = acc

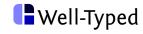
go acc(x : xs) = go[(x : acc)]xs
```

```
\begin{array}{l} \text{sum} :: \text{Num } a \Rightarrow [a] \rightarrow a \\ \text{sum} = \text{go} \boxed{0} \\ \textbf{where} \\ \text{go} :: \text{Num } a \Rightarrow a \rightarrow [a] \rightarrow a \\ \text{go acc} \left[\right] &= \text{acc} \\ \text{go acc} \left(x : xs\right) = \text{go} \boxed{(x + \text{acc})} xs \end{array}
```

See something to abstract here?



65 - Capturing design patterns - Polymorphism and higher-order functions

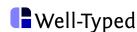


Abstracting

```
\begin{array}{l} \text{function} :: [a] \rightarrow r \\ \text{function} = \text{go } \dots \\ \textbf{where} \\ \text{go acc} \left[ \right] &= \text{acc} \\ \text{go acc} \left( x : xs \right) = \text{go} \left( \dots \text{ acc } \dots \text{ } x \dots \right) xs \end{array}
```

We apply the same tactics as before: let's abstract from the interesting positions and introduce names.





Abstracting

```
function :: [a] \rightarrow r

function = go e

where

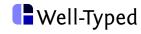
go acc [] = acc

go acc (x : xs) = go (op acc x) xs
```

Now we need to introduce e and op as parameters.



67 - Capturing design patterns - Polymorphism and higher-order functions



Abstracting

```
function :: ... \rightarrow ... \rightarrow [a] \rightarrow r
function op e = go e

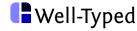
where

go acc [] = acc

go acc (x:xs) = go (op acc x) xs
```

And we have to figure out the types (or let the compiler infer them).





Abstracting

```
function :: (r \rightarrow a \rightarrow r) \rightarrow r \rightarrow [a] \rightarrow r
function op e = go e
where
go acc [] = acc
go acc (x : xs) = go (op acc x) xs
```

This function is called fold.



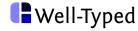
69 - Capturing design patterns - Polymorphism and higher-order functions



Abstracting

This function is called fold.





$$\begin{aligned} &\text{foldr } (\oplus) \text{ e } [x,y,z] = x \oplus (y \oplus (z \oplus e)) \\ &\text{foldl } (\oplus) \text{ e } [x,y,z] = ((e \oplus x) \oplus y) \oplus z \end{aligned}$$

Performance advice

There's a function foldl' with the same type as foldl. It forces evaluation of the accumulating argument and is often more efficient than foldl.



71 – Capturing design patterns – Polymorphism and higher-order functions



Generic concepts

Some of the concepts we have seen are not specific to lists; for example:

- ► the function foldr replaces data constructors by suitable functions and follows the structure of the datatype, just like the standard design principle;
- the function <u>filter</u> traverses a data structure and produces a substructure containing just the elements with a certain property;
- the function map traverses a data structure and produces a new structure of the same shape, but with modified elements.

The final case is so important that Haskell captures this abstraction by means of a type class.





Mapping over other types

```
 \begin{tabular}{ll} \textbf{data} \end{tabular} \begin{tabular}{ll} \textbf{data} \end{tabular} \begin{tabular}{ll} \textbf{Tree } a = Leaf \ a \mid Node \ (Tree \ a) \ (Tree \ a) \ \\ mapTree \ :: (a \to b) \to Tree \ a \to Tree \ b \\ mapTree \ f \ (Leaf \ x) &= Leaf \ (f \ x) \ \\ mapTree \ f \ (Node \ I \ r) &= Node \ (mapTree \ f \ I) \ (mapTree \ f \ r) \ \\ \end{tabular}
```

```
data Maybe a = Nothing \mid Just \ a

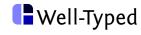
mapMaybe :: (a \rightarrow b) \rightarrow Maybe \ a \rightarrow Maybe \ b

mapMaybe f Nothing = Nothing

mapMaybe f (Just x) = Just (f x)
```



73 - Beyond lists - Polymorphism and higher-order functions



The Functor class

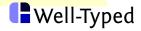
```
class Functor f where fmap :: (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
```

Note that unlike the Eq, Ord, Show and Read typeclasses, the Functor class abstracts over a parameterized type f.

```
instance Functor [] where
  fmap = map
instance Functor Tree where
  fmap = mapTree
instance Functor Maybe where
  fmap = mapMaybe
```

(
$$<$$
\$>) :: Functor f \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b f $<$ \$> x = fmap f x -- just a different name





Haskell files and modules

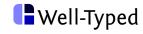
Haskell programs are structured into modules:

module MyModule where import Data.List import Data.Maybe
... -- rest of declarations

- ➤ One module per file.
- ► The module header is optional (default name Main).
- Except for the main module, module names should match file names.
- Modules can import other modules via import.
- ▶ Module imports have to be in the beginning of the module.
- Module names are uppercase, and can have several period-separated components.



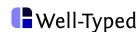
75 - On modules and libraries - Polymorphism and higher-order functions



The Prelude

- ► The Haskell Prelude is a module that is automatically imported into every module.
- It defines many of the functions we've been using. Say :bro Prelude in GHCi to get a list. (You can use :bro for other modules too.)
- ► It is possible to explicitly **import** the **Prelude** module in order to hide some names.





Hiding names, selected imports, qualified imports

Haskell's module system is deliberately kept simple and only allows basic namespace management. A few examples:

```
import Data.List (splitAt, scanl) -- only import these functionsimport qualified Data.Maybe -- everything must be qualifiedimport Prelude hiding (sum) -- do not import this function
```

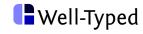
If you import a module qualified, then all references to names from the module have to be prefixed with the module name.

Double imports are possible:

```
import qualified Data.List
import Data.List (sortBy)
```



77 - On modules and libraries - Polymorphism and higher-order functions



A few interesting modules

Prelude -- the most basic functions

Data.List -- extra functions for list processing
Data.Char -- character classification functions

Data.Maybe -- extra functions for dealing with Maybe

Data.Map -- efficient finite maps (dictionaries)

Data.Set -- efficient sets

Tomorrow:

System.IO -- all sorts of input / output functions

Control.Monad -- overloaded monad functions





Finite maps

Finite maps are a more efficient implementation of association lists:

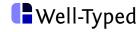
```
type AssocList key val = [(key, val)]
data Map key val -- abstract
```

Most important functions:

```
empty :: Map k v \rightarrow Int keys :: Map k v \rightarrow [k] elems :: Map k v \rightarrow [v] member :: Ord k \Rightarrow k \rightarrow Map k v \rightarrow Bool insert :: Ord k \Rightarrow k \rightarrow V \rightarrow Map k v \rightarrow Map k v delete :: Ord k \Rightarrow k \rightarrow Map k v \rightarrow Map k v lookup :: Ord k \Rightarrow k \rightarrow Map k v \rightarrow Maybe v
```



79 - On modules and libraries - Polymorphism and higher-order functions

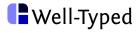


Implementation of finite maps

Internally, finite maps are implemented as balanced search trees:

- ► Therefore, the key type must be an instance of the Ord class.
- Operations that operate on single elements typically have logarithmic complexity.
- Operations that operate on the entire data structure typically have linear complexity.
- Note that in purely functional style, all "modifying" operations create a new Map from an old Map – they do not update in-place.





Finite maps are an instance of Functor

Specialized type:

fmap ::
$$(a \rightarrow b) \rightarrow Map k a \rightarrow Map k b$$

In fact, finite maps support several forms of mapping:

```
\begin{split} \text{map} &:: (a \to b) \to \text{Map k a} \to \text{Map k b} \quad \text{-- like fmap} \\ \text{mapKeys} &:: (k \to l) \to \text{Map k a} \to \text{Map l a} \\ \text{mapWithKey} &:: (k \to a \to b) \to \text{Map k a} \to \text{Map k b} \end{split}
```



81 - On modules and libraries - Polymorphism and higher-order functions



Finite maps and sets

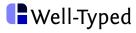
A finite map associates a finite number of keys with values.

In a set, we're only interested in membership, not in a value. One implementation option:

```
type Set key = Map key ()
```

The Data.Set module provides a specialized and optimized implementation of this idea. The interface of Data.Set is very similar to that of Data.Map.



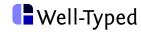


Sets

```
\begin{array}{lll} \textbf{data} \; \textbf{Set} \; \textbf{a} & \textbf{-- abstract} \\ \textbf{empty} & :: \; \textbf{Set} \; \textbf{a} \\ \textbf{size} & :: \; \textbf{Set} \; \textbf{a} \to \textbf{Int} \\ \textbf{elems} & :: \; \textbf{Set} \; \textbf{a} \to \textbf{Int} \\ \textbf{elems} & :: \; \textbf{Set} \; \textbf{a} \to \textbf{[a]} \\ \textbf{member} :: \; \textbf{Ord} \; \textbf{a} \Rightarrow \textbf{a} \to \textbf{Set} \; \textbf{a} \to \textbf{Bool} \\ \textbf{insert} & :: \; \textbf{Ord} \; \textbf{a} \Rightarrow \textbf{a} \to \textbf{Set} \; \textbf{a} \to \textbf{Set} \; \textbf{a} \\ \textbf{delete} & :: \; \textbf{Ord} \; \textbf{a} \Rightarrow \textbf{a} \to \textbf{Set} \; \textbf{a} \to \textbf{Set} \; \textbf{a} \\ \textbf{map} & :: \; (\textbf{Ord} \; \textbf{a}, \textbf{Ord} \; \textbf{b}) \Rightarrow (\textbf{a} \to \textbf{b}) \to \textbf{Set} \; \textbf{a} \to \textbf{Set} \; \textbf{b} \\ \end{array}
```



83 - On modules and libraries - Polymorphism and higher-order functions



Lessons

Try to pick good data structures:

- Lists are great for everything small, and everything that requires access from one side;
- Finite maps and sets are much better for large amounts of data where random access it required.
- Of course, there are lots of other data structures around, and many of them are quite easy to use.



