Polymorphism and higher-order functions

Fast Track to Haskell

Andres Löh, Edsko de Vries

8–9 April 2013 —

Well-Typed
The Haskell Consultants

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.

Parametric polymorphism

One function, several types

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

Example:

$$\mathsf{fst}\ (x, y) = x$$

Possible type signatures (all would work):

$$\mathsf{fst} :: (a, a) \rightarrow a$$
$$\mathsf{fst} :: (\mathsf{Int}, a) \rightarrow \mathsf{Int}$$
$$\mathsf{fst} :: (\mathsf{Int}, \mathsf{Int}) \rightarrow \mathsf{Int}$$
$$\mathsf{fst} :: (a, b) \rightarrow a$$
$$\mathsf{fst} :: (\mathsf{Int}, \mathsf{Char}) \rightarrow \mathsf{Int}$$

Is one of these clearly the "best" choice?

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

$$\text{fst} :: (a, b) \to a$$

declares the most general type for `fst`. Types like

$$\text{fst} :: (a, a) \to a$$
$$\text{fst} :: (\text{Int}, \text{Char}) \to \text{Int}$$
$$\text{fst} :: (a \to \text{Int} \to b, c) \to a \to \text{Int} \to b$$

are instantiations of the most general type.

Instantiating types

The type signature

$$\mathsf{fst} :: (a, b) \to a$$

declares the most general type for fst. Types like

$$\mathsf{fst} :: (a, a) \to a$$
$$\mathsf{fst} :: (\mathsf{Int}, \mathsf{Char}) \to \mathsf{Int}$$
$$\mathsf{fst} :: (a \to \mathsf{Int} \to b, c) \to a \to \mathsf{Int} \to b$$

are instantiations of the most general type.

Type inference will always infer the most general type!

Instantiating types

The type signature

$$\mathsf{fst} :: (a, b) \to a$$

declares the most general type for fst. Types like

$$\mathsf{fst} :: (a, a) \to a$$
$$\mathsf{fst} :: (\mathsf{Int}, \mathsf{Char}) \to \mathsf{Int}$$
$$\mathsf{fst} :: (a \to \mathsf{Int} \to b, c) \to a \to \mathsf{Int} \to b$$

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.)

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.

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:

$\mathsf{fst} :: (\mathsf{a}, \mathsf{b}) \to \mathsf{a}$
$\mathsf{fst}\ (\mathsf{x}, \mathsf{y}) = \mathsf{x}$

$\mathsf{restrictedFst} :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}$
$\mathsf{restrictedFst} = \mathsf{fst}$ -- ok

$\mathsf{newFst} :: (\mathsf{a}, \mathsf{b}) \to \mathsf{a}$
$\mathsf{newFst} = \mathsf{restrictedFst}$ -- type error!

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.

Example

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

$$(\mathsf{Int}, \mathsf{Int}) \rightarrow (\mathsf{Int}, \mathsf{Int})$$

Example

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

$$(\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$

And of this one?

$$(\mathsf{a}, \mathsf{a}) \to (\mathsf{a}, \mathsf{a})$$

Example

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

$$(\mathsf{Int}, \mathsf{Int}) \to (\mathsf{Int}, \mathsf{Int})$$

And of this one?

$$(\mathsf{a}, \mathsf{a}) \to (\mathsf{a}, \mathsf{a})$$

And of this one?

$$(\mathsf{a}, \mathsf{b}) \to (\mathsf{b}, \mathsf{a})$$

(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".)

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, $\mathsf{map} :: (\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}]$ 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.

A common pitfall: who gets to choose

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

$$
\begin{aligned}
&\mathsf{parse} :: \mathsf{String} \to \mathsf{a} \\
&\mathsf{parse}\ \texttt{"False"} = \mathsf{False} \\
&\mathsf{parse}\ \texttt{"0"} \qquad\ = 0 \\
&\ldots
\end{aligned}
$$

What is wrong here?

A common pitfall: who gets to choose

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

$$
\begin{aligned}
&\mathsf{parse} :: \mathsf{String} \to \mathsf{a} \\
&\mathsf{parse}\ \texttt{"False"} = \mathsf{False} \\
&\mathsf{parse}\ \texttt{"0"} \qquad = 0 \\
&\ldots
\end{aligned}
$$

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 → Either Bool Int
parse "False" = Left False
parse "0"     = Right 0
```

What if we need to return values of different types?

Option 1: use Either:

```
data Either a b = Left a | Right b
parse :: String → 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, ...).

Overloading

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 → a → Bool
  (≠) :: a → a → Bool
```

Type classes

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

Example:

```
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
```

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

Instances

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

  x ≠ y = not (x == y)
```

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

  xs ≠ ys = not (xs == ys)
```

Instances

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

  x ≠ y = not (x == y)
```

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

  xs ≠ 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.

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:

$$(==) :: \mathsf{Eq}\ a \Rightarrow a \to a \to \mathsf{Bool}$$

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

Overloading and inference

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

$$
\begin{aligned}
&\mathsf{allEqual} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow \mathsf{Bool}\\
&\mathsf{allEqual}\ [\,] \qquad\qquad = \mathsf{True}\\
&\mathsf{allEqual}\ [x] \qquad\quad\ = \mathsf{True}\\
&\mathsf{allEqual}\ (x : y : ys) = x == y\ \&\&\ \mathsf{allEqual}\ (y : ys)
\end{aligned}
$$

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:

```haskell
example :: ... => (a, b) -> (a, b) -> String
example (x1, y1) (x2, y2)
    | x1 == x2 && y1 == y2 = show x1
    | otherwise            = "different"
```

Can you infer the constraints?

Several class constraints

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

```haskell
example :: (Eq a, Eq b, Show a) => (a, b) -> (a, b) -> String
example (x1, y1) (x2, y2)
    | x1 == x2 && y1 == y2 = show x1
    | otherwise            = "different"
```

Default definitions

```
class Eq a where
  (==) :: a → a → Bool
  (≠)  :: a → a → Bool

  x == y = not (x ≠ y)
  x ≠ y  = not (x == y)
```

Now:

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

And $(\neq)$ will work automatically.

Default definitions

```
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
  x == y = not (x ≠ y)
  x ≠ y = not (x == y)
```

Now:

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

And `(≠)` will work automatically.

Careful: if you provide neither `(==)` nor `(≠)`, you won't get a complaint, but both functions will loop.

Classes are not types!

Note that

```
f :: Eq → Eq → Bool
f :: Eq a → Eq a → Bool
```

are both invalid. Classes appear in constraints!

Classes are not types!

Note that

$$f :: Eq \to Eq \to Bool$$
$$f :: Eq\ a \to Eq\ a \to Bool$$

are both invalid. Classes appear in constraints!

Also note that the type

$$Eq\ a \Rightarrow a \to a \to Bool$$

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

$$(Eq\ a, Eq\ b) \Rightarrow a \to b \to Bool$$

Important classes

Eq

- 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.

Ord

For comparisons between values of the same type.

$$
\begin{aligned}
&\textbf{class}\ \mathrm{Eq}\ a \Rightarrow \mathrm{Ord}\ a\ \textbf{where}\\
&\quad \mathrm{compare} :: a \to a \to \mathrm{Ordering}\\
&\quad (<) :: a \to a \to \mathrm{Bool}\\
&\quad (\leqslant) :: a \to a \to \mathrm{Bool}\\
&\quad (>) :: a \to a \to \mathrm{Bool}\\
&\quad (\geqslant) :: a \to a \to \mathrm{Bool}\\
&\quad \mathrm{max} :: a \to a \to a\\
&\quad \mathrm{min} :: a \to a \to a
\end{aligned}
$$

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

$$\textbf{data}\ \mathrm{Ordering} = \mathrm{LT} \mid \mathrm{EQ} \mid \mathrm{GT}$$

Superclasses

```
class Eq a ⇒ 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 ⇒ ...` on a function implies `Eq a`. In other words, `(Ord a, Eq a) ⇒ ...` is equivalent to `Ord a ⇒ ...`.

Show

```
class Show a where
  show      :: a → String
  showsPrec :: Int → a → ShowS
  showList  :: [a] → 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.

Read

```
class Read a where
  readsPrec :: Int → ReadS a
  readList :: ReadS [a]
```

Most often, the derived function `read` is used:

```
read :: Read a ⇒ String → 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 = 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?

Unresolved overloading

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

```
strange x = 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.

Manually resolving overloading

This works:

$$\mathsf{strange} :: \mathsf{String} \rightarrow \mathsf{String}$$
$$\mathsf{strange}\ x = \mathsf{show}\ (\mathsf{read}\ x :: \mathsf{Bool})$$

Or this:

$$\mathsf{strange} :: \mathsf{String} \rightarrow \mathsf{String}$$
$$\mathsf{strange}\ x = \mathsf{show}\ (\mathsf{read}\ x :: \mathsf{Int})$$

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

Manually resolving overloading

This works:

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

Or this:

$$
\begin{array}{l}
\mathsf{strange} :: \mathsf{String} \to \mathsf{String} \\
\mathsf{strange}\ \mathsf{x} = \mathsf{show}\ (\mathsf{read}\ \mathsf{x} :: \mathsf{Int})
\end{array}
$$

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.

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.

Numbers

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

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`.

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:

$$
\begin{aligned}
(+) &:: (\mathsf{Num}\ a) \Rightarrow a \rightarrow a \rightarrow a \\
(-) &:: (\mathsf{Num}\ a) \Rightarrow a \rightarrow a \rightarrow a \\
(*) &:: (\mathsf{Num}\ a) \Rightarrow a \rightarrow a \rightarrow a
\end{aligned}
$$

Operations on numbers

Most operations on numbers and even numeric literals are overloaded:

$$
\begin{aligned}
(+) &:: (\mathsf{Num}\ a) \Rightarrow a \to a \to a \\
(-) &:: (\mathsf{Num}\ a) \Rightarrow a \to a \to a \\
(*) &:: (\mathsf{Num}\ a) \Rightarrow a \to a \to a
\end{aligned}
$$

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

Operations on numbers

Most operations on numbers and even numeric literals are overloaded:

```
(+) :: (Num a) ⇒ a → a → a
(−) :: (Num a) ⇒ a → a → a
(∗) :: (Num a) ⇒ a → a → a
```

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

```
(/) :: (Fractional a) ⇒ a → a → a

mod :: (Integral   a) ⇒ a → a → a
div :: (Integral   a) ⇒ a → a → a

sin :: (Floating   a) ⇒ a → a
log :: (Floating   a) ⇒ a → a
```

No automatic coercion

We can use overloaded functions at different types:

```
3 * 4
3.2 * 4.5
```

But there is no implicit coercion:

```
3.2 * (5 `div` 2)    -- type error
3.2 * fromIntegral (5 `div` 2)
```

No automatic coercion

We can use overloaded functions at different types:

$3 * 4$
$3.2 * 4.5$

But there is no implicit coercion:

```
3.2 * (5 `div` 2)    -- type error
3.2 * fromIntegral (5 `div` 2)
```

Question

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

No automatic coercion

We can use overloaded functions at different types:

```
3 * 4
3.2 * 4.5
```

But there is no implicit coercion:

```
3.2 * (5 `div` 2)    -- type error
3.2 * fromIntegral (5 `div` 2)
```

Question

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

Because `2 :: (Num a) ⇒ a`, but `(5 `div` 2) :: (Integral a) ⇒ a`.

Converting between numeric types

From an integral type to another:

$$\mathsf{fromIntegral} :: (\mathsf{Integral}\ a, \mathsf{Num}\ b) \Rightarrow a \rightarrow b$$

From a fractional type to an integral:

$$\begin{array}{lll}
\mathsf{round} & :: (\mathsf{RealFrac}\ a, \mathsf{Integral}\ b) \Rightarrow a \rightarrow b & \text{-- nearest even} \\
\mathsf{floor} & :: (\mathsf{RealFrac}\ a, \mathsf{Integral}\ b) \Rightarrow a \rightarrow b & \\
\mathsf{ceiling} & :: (\mathsf{RealFrac}\ a, \mathsf{Integral}\ b) \Rightarrow a \rightarrow b &
\end{array}$$

Recap: Haskell types

Question

Which Haskell types have we seen so far?

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.

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`).

Higher-order functions

Functions, functions, functions

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

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:

$$\begin{aligned}
&\mathsf{elem} && :: \mathsf{Eq}\ a \Rightarrow a \to ([a] \to \mathsf{Bool}) \\
&\mathsf{elem}\ 3 && :: (\mathsf{Eq}\ a, \mathsf{Num}\ a) \Rightarrow [a] \to \mathsf{Bool}
\end{aligned}$$

Filtering and mapping

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

$$\text{filter} :: (a \to \text{Bool}) \to [a] \to [a]$$
$$\text{map} :: (a \to b) \to [a] \to [b]$$

Filtering and mapping

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

$$\text{filter} :: (a \to \text{Bool}) \to ([a] \to [a])$$
$$\text{map} :: (a \to b) \to ([a] \to [b])$$

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

Overloading vs. parameterization

Consider:

$$
\begin{aligned}
&\mathrm{sort} &&:: \mathrm{Ord}\ a \Rightarrow [a] \to [a] \\
&\mathrm{sortBy} &&:: (a \to a \to \mathrm{Ordering}) \to [a] \to [a]
\end{aligned}
$$

Overloading vs. parameterization

Consider:

$$\begin{aligned}&\mathsf{sort} && :: \mathsf{Ord}\ a \Rightarrow [a] \rightarrow [a]\\ &\mathsf{sortBy} && :: (a \rightarrow a \rightarrow \mathsf{Ordering}) \rightarrow [a] \rightarrow [a]\end{aligned}$$

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.

Overloading vs. parameterization

Consider:

$$\mathsf{sort} \;::\; \mathsf{Ord}\ a \Rightarrow [a] \to [a]$$
$$\mathsf{sortBy} \;::\; (a \to a \to \mathsf{Ordering}) \to [a] \to [a]$$

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.

Function composition

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

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

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

Function composition

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

$$(\circ) :: \ldots \to \ldots \to \ldots \to \ldots$$
$$(\mathsf{f} \circ \mathsf{g})\ \mathsf{x} = \mathsf{f}\ (\mathsf{g}\ \mathsf{x})$$

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

Function composition

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

$$(\circ) :: (\ldots \to \ldots) \to (\ldots \to \ldots) \to \ldots \to \ldots$$
$$(\mathsf{f} \circ \mathsf{g})\ \mathsf{x} = \mathsf{f}\ (\mathsf{g}\ \mathsf{x})$$

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

Function composition

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

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

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 ...

Function composition

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

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

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

Well-Typed

Function composition

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

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

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

Function composition

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

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

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

Function composition

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

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

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

Composing functions

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

Composing functions

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

Example: Computing the first 100 odd square numbers.

$$\mathit{example} :: [\mathit{Int}]$$
$$\mathit{example} = \qquad [1\,..]$$

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]
example =                    map (λx → x * x)  [1 ..]
```

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

Composing functions

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

Example: Computing the first 100 odd square numbers.

$$\mathsf{example} :: [\mathsf{Int}]$$
$$\mathsf{example} = (\qquad\qquad \mathsf{filter}\ \mathsf{odd} \circ \mathsf{map}\ (\lambda \mathsf{x} \rightarrow \mathsf{x} * \mathsf{x}))\ [1\,..]$$

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.

$$\mathsf{example} :: [\mathsf{Int}]$$
$$\mathsf{example} = (\mathsf{take}\ 100 \circ \mathsf{filter\ odd} \circ \mathsf{map}\ (\lambda \mathsf{x} \rightarrow \mathsf{x} * \mathsf{x}))\ [1\,..]$$

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

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.

Operating on functions

Flipping a function

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

$$\text{flip} :: (a \to b \to c) \to (b \to a \to c)$$
$$\text{flip}\ f\ x\ y = f\ y\ x$$

Flipping a function

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

```
flip :: (a → b → c) → b → a → c
flip f x y = f y x
```

Note once again that the function arrow associates to the right, so `flip` can really be seen as a function with three arguments:

```
f :: a → b → c
x :: b
y :: a
```

Flipping a function

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

$$\mathrm{flip} :: (a \to b \to c) \to b \to a \to c$$
$$\mathrm{flip}\ f\ x\ y = f\ y\ x$$

Example use:

$$\mathrm{foreach} = \mathrm{flip}\ \mathrm{map}$$
$$\mathrm{example} = \mathrm{foreach}\ [1, 2, 3]\ (\lambda x \to x * x)$$

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:

$$\mathsf{uncurry} :: (a \to b \to c) \to (a, b) \to c$$
$$\mathsf{uncurry}\ f\ (x, y) = f\ x\ y$$

$$\mathsf{curry} :: ((a, b) \to c) \to a \to b \to c$$
$$\mathsf{curry}\ f\ x\ y = f\ (x, y)$$

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:

$$\mathsf{uncurry} :: (a \to b \to c) \to (a, b) \to c$$
$$\mathsf{uncurry}\ f\ (x, y) = f\ x\ y$$

$$\mathsf{curry} :: ((a, b) \to c) \to a \to b \to c$$
$$\mathsf{curry}\ f\ x\ y = f\ (x, y)$$

Example:

$$\mathsf{map}\ (\mathsf{uncurry}\ (*))\ (\mathsf{zip}\ [1\,..\,3]\ [4\,..\,6])$$

Capturing design patterns

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.

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:

```
function :: [someType] → someResult
function []       = ...   -- code
function (x : xs) = ...   -- code that can use  x  and  function xs
```

From an informal pattern to a function

```
function :: [someType] → someResult
function []       = ...   -- code
function (x : xs) = ...   -- code that can use  x  and  function xs
```

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

From an informal pattern to a function

$$
\begin{aligned}
&\mathsf{function} :: [\mathsf{someType}] \to \mathsf{someResult} \\
&\mathsf{function}\ [\,] \qquad\quad = \mathsf{nil} \\
&\mathsf{function}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{cons}\ \mathsf{x}\ (\mathsf{function}\ \mathsf{xs})
\end{aligned}
$$

- 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!

From an informal pattern to a function

$$
\begin{aligned}
&\textsf{function} :: \ldots \to \ldots \to [\textsf{someType}] \to \textsf{someResult}\\
&\textsf{function cons nil } [\,] \qquad\quad = \textsf{nil}\\
&\textsf{function cons nil } (\textsf{x} : \textsf{xs}) = \textsf{cons x } (\textsf{function cons nil xs})
\end{aligned}
$$

We now have to look at the types of $\textsf{cons}$ and $\textsf{nil}$:

- $\textsf{nil}$ is used as a result, so $\textsf{nil} :: \textsf{someResult}$;
- $\textsf{cons}$ takes a list element and a result to a result, so $\textsf{cons} :: \textsf{someType} \to \textsf{someResult} \to \textsf{someResult}$.

From an informal pattern to a function

$$
\begin{aligned}
&\mathsf{function} :: (\mathsf{someType} \to \mathsf{someResult} \to \mathsf{someResult})\\
&\qquad \to \mathsf{someResult}\\
&\qquad \to [\mathsf{someType}] \to \mathsf{someResult}\\
&\mathsf{function}\ \mathsf{cons}\ \mathsf{nil}\ [\,] = \mathsf{nil}\\
&\mathsf{function}\ \mathsf{cons}\ \mathsf{nil}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{cons}\ \mathsf{x}\ (\mathsf{function}\ \mathsf{cons}\ \mathsf{nil}\ \mathsf{xs})
\end{aligned}
$$

We can give shorter names to someType and someResult

...

From an informal pattern to a function

$$
\begin{aligned}
&\mathsf{function} :: (\mathsf{a} \to \mathsf{r} \to \mathsf{r}) \to \mathsf{r} \to [\mathsf{a}] \to \mathsf{r} \\
&\mathsf{function}\ \mathsf{cons}\ \mathsf{nil}\ [\,] = \mathsf{nil} \\
&\mathsf{function}\ \mathsf{cons}\ \mathsf{nil}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{cons}\ \mathsf{x}\ (\mathsf{function}\ \mathsf{cons}\ \mathsf{nil}\ \mathsf{xs})
\end{aligned}
$$

This function is called foldr . . .

From an informal pattern to a function

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

We could equivalently define it using **where** …

From an informal pattern to a function

```
foldr :: (a → r → r) → r → [a] → r
foldr cons nil = go
  where
    go []       = nil
    go (x : xs) = cons x (go xs)
```

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.

Using foldr

$$
\begin{aligned}
&\mathrm{length} :: [a] \to \mathrm{Int} \\
&\mathrm{length}\ [\,] \qquad\quad = 0 \\
&\mathrm{length}\ (x : xs) = 1 + \mathrm{length}\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{foldr} :: (a \to r \to r) \to r \to [a] \to r \\
&\mathrm{foldr}\ \mathrm{cons}\ \mathrm{nil} = \mathrm{go}\ [\,] \\
&\quad \mathbf{where} \\
&\qquad \mathrm{go}\ [\,] \qquad\quad = \mathrm{nil} \\
&\qquad \mathrm{go}\ (x : xs) = \mathrm{cons}\ x\ (\mathrm{go}\ xs)
\end{aligned}
$$

Using foldr

$$\begin{aligned}
&\mathsf{length} :: [a] \to \mathsf{Int} \\
&\mathsf{length}\ [\,] \qquad\quad = 0 \\
&\mathsf{length}\ (x : xs) = 1 + \mathsf{length}\ xs
\end{aligned}$$

$$\begin{aligned}
&\mathsf{foldr} :: (a \to r \to r) \to r \to [a] \to r \\
&\mathsf{foldr}\ cons\ nil = go\ [\,] \\
&\quad \mathbf{where} \\
&\qquad go\ [\,] \qquad\quad = nil \\
&\qquad go\ (x : xs) = cons\ x\ (go\ xs)
\end{aligned}$$

$$\mathsf{length} = \mathsf{foldr}\ (\lambda x\ r \to 1 + r)\ 0$$

or (using const and an operator section)

$$\mathsf{length} = \mathsf{foldr}\ (\mathsf{const}\ (1\ +))\ 0$$

Examples of using foldr

$$(\mathbin{+\!\!+}) :: [a] \to [a] \to [a]$$
$$(\mathbin{+\!\!+})\ xs\ ys = \mathrm{foldr}\ (:)\ ys\ xs$$

$$\mathrm{filter} :: (a \to \mathrm{Bool}) \to [a] \to [a]$$
$$\mathrm{filter}\ p = \mathrm{foldr}\ (\lambda x\ r \to \mathbf{if}\ p\ x\ \mathbf{then}\ x : r\ \mathbf{else}\ r)\ [\,]$$

$$\mathrm{map} :: (a \to b) \to [a] \to [b]$$
$$\mathrm{map}\ f = \mathrm{foldr}\ (\lambda x\ r \to f\ x : r)\ [\,]$$

Examples of using foldr

```haskell
(++) :: [a] → [a] → [a]
(++) xs ys = foldr (:) ys xs

filter :: (a → Bool) → [a] → [a]
filter p = foldr (λx r → if p x then x : r else r) []

map :: (a → b) → [a] → [b]
map f = foldr (λx r → f x : r) []
```

```haskell
and :: [Bool] → Bool
and = foldr (&&) True

any :: (a → Bool) → [a] → Bool
any p = foldr (λx r → p x || r) False
```

And many more.

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

```haskell
reverse :: [a] → [a]
reverse = go []
  where
    go :: [a] → [a] → [a]
    go acc []       = acc
    go acc (x : xs) = go (x : acc) xs
```

```haskell
sum :: Num a ⇒ [a] → a
sum = go 0
  where
    go :: Num a ⇒ a → [a] → a
    go acc []       = acc
    go acc (x : xs) = go (x + acc) xs
```

See something to abstract here?

Accumulating parameter pattern

```
reverse :: [a] → [a]
reverse = go []
  where
    go :: [a] → [a] → [a]
    go acc []       = acc
    go acc (x : xs) = go (x : acc) xs
```

```
sum :: Num a ⇒ [a] → a
sum = go 0
  where
    go :: Num a ⇒ a → [a] → a
    go acc []       = acc
    go acc (x : xs) = go (x + acc) xs
```

See something to abstract here?

Abstracting

$$
\begin{aligned}
&\text{function} :: [a] \to r\\
&\text{function} = \text{go}\ \ldots\\
&\quad \textbf{where}\\
&\qquad \text{go acc } [\,] \qquad\quad = \text{acc}\\
&\qquad \text{go acc } (x : xs) = \text{go } (\ldots\ \text{acc}\ \ldots\ x\ \ldots)\ xs
\end{aligned}
$$

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

Abstracting

```
function :: [a] → 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.

Abstracting

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

This function is called `foldl`.

Abstracting

```
foldl :: (r → a → r) → r → [a] → r
foldl op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = go (op acc x) xs
```

This function is called `foldl`.

foldr and foldl

$$\mathsf{foldr}\ (\oplus)\ e\ [x, y, z] = x \oplus (y \oplus (z \oplus e))$$
$$\mathsf{foldl}\ (\oplus)\ e\ [x, y, z] = ((e \oplus x) \oplus y) \oplus z$$

foldr and foldl

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

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.

Beyond lists

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 `filter` 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

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

mapTree :: (a → b) → Tree a → Tree b
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node l r) = Node (mapTree f l) (mapTree f r)
```

Mapping over other types

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node l r) = Node (mapTree f l) (mapTree f r)
```

```haskell
data Maybe a = Nothing | Just a

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

The Functor class

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

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

The Functor class

```
class Functor f where
  fmap :: (a → b) → f a → 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
```

The Functor class

```
class Functor f where
  fmap :: (a → b) → f a → 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
```

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
f <$> x = fmap f x    -- just a different name
```

On modules and libraries

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.

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 functions
import qualified Data.Maybe          -- everything must be qualified
import 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.

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 functions
import qualified Data.Maybe          -- everything must be qualified
import 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)
```

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:

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

Most important functions:

```haskell
empty  :: Map k v
size   :: Map k v → Int

keys   :: Map k v → [k]
elems  :: Map k v → [v]
member :: Ord k ⇒ k → Map k v → Bool

insert :: Ord k ⇒ k → v → Map k v → Map k v
delete :: Ord k ⇒ k → Map k v → Map k v
lookup :: Ord k ⇒ k → Map k v → Maybe v
```

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 → b) → Map k a → Map k b
```

Skills Matter · Well-Typed

Finite maps are an instance of Functor

Specialized type:

```
fmap :: (a → b) → Map k a → Map k b
```

In fact, finite maps support several forms of mapping:

```
map :: (a → b) → Map k a → Map k b    -- like  fmap
mapKeys       :: (k → l) → Map k a → Map l a
mapWithKey :: (k → a → b) → Map k a → Map k b
```

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 ()
```

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

```
data Set a   -- abstract

empty  :: Set a
size   :: Set a → Int

elems  :: Set a → [a]
member :: Ord a ⇒ a → Set a → Bool

insert :: Ord a ⇒ a → Set a → Set a
delete :: Ord a ⇒ a → Set a → Set a
map    :: (Ord a, Ord b) ⇒ (a → b) → Set a → Set b
```

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.