Part 4: Polymorphism and overloading

Introduction to Haskell

Andres Löh

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:

```
fst (x, y) = x
```

Possible type signatures (all would work):

```
fst :: (a, a) -> a
fst :: (Int, a) -> Int
fst :: (Int, Int) -> Int
fst :: (a, b) -> a
fst :: (Int, Char) -> 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

```
fst :: (a, b) -> a
```

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

```
fst :: (a, a) -> a
fst :: (Int, Char) -> Int
fst :: (a -> Int -> b, c) -> a -> Int -> b
```

are instantiations of the most general type.

Instantiating types

The type signature

```
fst :: (a, b) -> a
```

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

```
fst :: (a, a) -> a
fst :: (Int, Char) -> Int
fst :: (a -> Int -> b, c) -> a -> Int -> b
```

are instantiations of the most general type.

Type inference will always infer the most general type!

Instantiating types

The type signature

```haskell
fst :: (a, b) -> a
```

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

```haskell
fst :: (a, a) -> a
fst :: (Int, Char) -> Int
fst :: (a -> Int -> b, c) -> a -> Int -> 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:

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

restrictedFst :: (Int, Int) -> Int
restrictedFst = fst  -- ok
```

```
newFst :: (a, b) -> a
newFst = 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:

```
(Int, Int) -> (Int, Int)
```

Example

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

```
(Int, Int) -> (Int, Int)
```

And of this one?

```
(a, a) -> (a, a)
```

Example

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

```
(Int, Int) -> (Int, Int)
```

And of this one?

```
(a, a) -> (a, a)
```

And of this one?

```
(a, b) -> (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".)

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.

A common pitfall: who gets to choose

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

```
parse :: String -> a
parse "False" = False
parse "0"     = 0
...
```

What is wrong here?

A common pitfall: who gets to choose

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

```
parse :: String -> 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 -> 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)
```

Well-Typed

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:

```
(==) :: Eq a => a -> a -> Bool
```

Read: “Given that `a` is an instance of `Eq`, the function has the type `a -> a -> Bool`.”

Overloading and inference

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

```
allEqual :: Eq a => [a] -> Bool
allEqual []             = True
allEqual [x]            = True
allEqual (x : y : ys) = x == y && allEqual (y : ys)
```

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 ::
  ... => (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:

```
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

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

Now:

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

And `(/=)` 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 -> Eq -> Bool
f :: Eq a -> Eq a -> Bool
```

are both invalid. Classes appear in constraints!

Also note that the type

```
Eq a => a -> a -> 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) => a -> b -> 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.

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<)     :: a -> a -> Bool
  (<=)    :: a -> a -> Bool
  (>)     :: a -> a -> Bool
  (>=)    :: a -> a -> Bool
  max     :: a -> a -> a
  min     :: a -> a -> a
```

Several default definitions – you'd typically define just `compare` or `(<=)`.

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

Superclasses

```haskell
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 => ...`.

Overloading vs. parameterization

Consider:

```
sort   :: Ord a                     => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Overloading vs. parameterization

Consider:

```
sort   :: Ord a                      => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [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.

Overloading vs. parameterization

Consider:

```
sort   :: Ord a                  => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [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.

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

```haskell
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
```

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

```haskell
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:

```haskell
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:

```
strange :: String -> String
strange x = show (read x :: Bool)
```

Or this:

```
strange :: String -> String
strange x = show (read x :: Int)
```

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

Manually resolving overloading

This works:

```
strange :: String -> String
strange x = show (read x :: Bool)
```

Or this:

```
strange :: String -> String
strange x = show (read x :: Int)
```

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.

Type classes on type constructors

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 `elem` traverses a data structure and checks whether it contains a particular element;
- 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.

For some of these concepts, Haskell therefore offers more type classes.

Foldable

A class for data structures that can be viewed as a list, i.e., that have elements in some natural order.

```
class Foldable t where
  foldr   :: (a -> b -> b) -> b -> t a -> b
  foldl'  :: (b -> b -> b) -> b -> t a -> b
  toList  :: t a -> [a]
  null    :: t a -> Bool
  length  :: t a -> Int
  elem    :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  product :: Num a => t a -> a
  ...
```

Some of these are only available via `Data.Foldable` .

Note that `Foldable` abstracts over a **parameterized** type `t` .

Other foldable types

The `Maybe` type is a container with 0 or 1 elements:

```
GHCi> null (Just 3)
False
GHCi> null Nothing
True
GHCi> product Nothing
1
```

Possible pitfall: foldable tuples and `Either`

A pair is a container containing exactly 1 element (its **second** component). (Tagged value.)

```
GHCi> toList (3, 4)
[4]
GHCi> toList ("foo", True)
[True]
GHCi> sum (3, 4)
4
```

Possible pitfall: foldable tuples and `Either`

A pair is a container containing exactly 1 element (its **second** component). (Tagged value.)

```
GHCi> toList (3, 4)
[4]
GHCi> toList ("foo", True)
[True]
GHCi> sum (3, 4)
4
```

An `Either` is like `Maybe` where `Nothing` is replaced by `Left` .
So `Right` injects an element, `Left` does not.

```
GHCi> length (Right 3)
1
GHCi> length (Left 3)
0
```

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)
```

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

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Like `Foldable`, the class `Functor` abstracts over a parameterized type.

The `Functor` class

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Like `Foldable`, the class `Functor` abstracts over a parameterized type.

```
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree

instance Functor Maybe where
  fmap = mapMaybe
```

The `Functor` class

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Like `Foldable`, the class `Functor` abstracts over a parameterized type.

```haskell
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree

instance Functor Maybe where
  fmap = mapMaybe
```

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

Deriving `Functor` and `Foldable`

Class instances for `Functor` and `Foldable` (and a few other classes) can be derived via language extensions:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}
```

Language pragmas have to appear at the top of the module.

Deriving `Functor` and `Foldable`

Class instances for `Functor` and `Foldable` (and a few other classes) can be derived via language extensions:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}
```

Language pragmas have to appear at the top of the module.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Show, Eq, Functor, Foldable)
```

Deriving `Functor` and `Foldable`

Class instances for `Functor` and `Foldable` (and a few other classes) can be derived via language extensions:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}
```

Language pragmas have to appear at the top of the module.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Show, Eq, Functor, Foldable)
```

```
GHCi> length (Node (Leaf 3) (Leaf 4))
2
GHCi> (+ 1) <$> Node (Leaf 3) (Leaf 4)
Node (Leaf 4) (Leaf 5)
```

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**:

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

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
```

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:

```
fromIntegral :: (Integral a, Num b) => a -> b
```

From a fractional type to an integral:

```
round   :: (RealFrac a, Integral b) => a -> b
floor   :: (RealFrac a, Integral b) => a -> b
ceiling :: (RealFrac a, Integral b) => a -> b
```

Here, `round` rounds to the nearest even number.