Haskell: type classes and higher-order polymorphism

Julian Grove

September 11, 2023

Computational Semantics

Introduction

Last time

We looked at function literals, pattern matching, and some basic functions on lists.

This time

We'll look at more things which are specific to Haskell:

This time

We'll look at more things which are specific to Haskell:

- Types and type classes

This time

We'll look at more things which are specific to Haskell:

- Types and type classes
- Polymorphism

This time

We'll look at more things which are specific to Haskell:

- Types and type classes
- Polymorphism

This time

We'll look at more things which are specific to Haskell:

- Types and type classes
- Polymorphism

Polymorphism will be a big player in the part of the course when we start looking at functors. (It is a special kind of abstraction.)

Some more types

Maybe

A commonly used data type in Haskell is `Maybe a`.

Maybe

A commonly used data type in Haskell is `Maybe a`. `Maybe a` is like the data type `a` on its own, but with extra information about whether the a computational succeeded for failed.

Key-value pairs

Example: a function for looking up the value associated with a key.

Key-value pairs

Example: a function for looking up the value associated with a key.

No exception was thrown when I tried to look up `"the Stone Roses"`… the function returned a value, `Nothing`, inhabitting the data type `Maybe Integer`.

Key-value pairs

Example: a function for looking up the value associated with a key.

No exception was thrown when I tried to look up `"the Stone Roses"`… the function returned a value, `Nothing`, inhabitting the data type `Maybe Integer`.

One way to think of the `Maybe a` data type is that it represents an action your computer can perform—throwing an error—as data.

Key-value pairs

Example: a function for looking up the value associated with a key.

No exception was thrown when I tried to look up `"the Stone Roses"`… the function returned a value, `Nothing`, inhabitting the data type `Maybe Integer`.

One way to think of the `Maybe a` data type is that it represents an action your computer can perform—throwing an error—as data.

This illustrates what in Haskell is a pretty commonly used technique of bluring the lines between *effects* and data.

`div`

Let's look at the function `div :: Integer -> Integer -> Integer`.

`div`

Let's look at the function `div :: Integer -> Integer -> Integer`.

`div` is a partial function.

`div`

Let’s look at the function `div :: Integer -> Integer -> Integer`.

`div` is a partial function.

Can we write a function `safeDiv` which is a total function, using maybe types?

Either

A generalization of `Maybe a` is the data type `Either a b`.

Either

A generalization of `Maybe a` is the data type `Either a b`.

`Left` means failure; `Right` means success.

Either

A generalization of `Maybe a` is the data type `Either a b`.

`Left` means failure; `Right` means success. Questions:

- How might we represent `Maybe` data types as `Either` data types?
- In what sense are maybe types a generalization of either types?

Type classes and polymorphism

Type classes

One of the most famous distinguishing features of Haskell.

Type classes

One of the most famous distinguishing features of Haskell.

A type class allows you to provide multiple implementations of what looks like the same function on different types.

Type classes

One of the most famous distinguishing features of Haskell.

A type class allows you to provide multiple implementations of what looks like the same function on different types.

These implementations are called *instances*.

Show

We have already seen type classes when we've used `deriving Show` in data type declarations.

Show

We have already seen type classes when we've used `deriving Show` in data type declarations.

But we can actually implement `Show` instances ourselves.

Eq

Another useful type class is `Eq`.

Eq

Another useful type class is `Eq`.

Like `Show` instances, `Eq` instances can be derived.

Declaring type classes

We can declare our own type classes.

Declaring type classes

We can declare our own type classes.

For the three computer brands, you might want to know which of its models' keys is are possible reboot keys…

Kinds of polymorphism

Kinds of polymorphism

Each of the type classes we’ve looked at provide what is known as *ad hoc polymorphism.*

Kinds of polymorphism

Each of the type classes we've looked at provide what is known as *ad hoc polymorphism*.

- Only some data types need provide instances.

Kinds of polymorphism

Each of the type classes we've looked at provide what is known as *ad hoc polymorphism*.

- Only some data types need provide instances.
- Different instances can have fundamentally *different* definitions.

Kinds of polymorphism

Each of the type classes we’ve looked at provide what is known as *ad hoc polymorphism.*

- Only some data types need provide instances.
- Different instances can have fundamentally *different* definitions.

Kinds of polymorphism

Each of the type classes we've looked at provide what is known as *ad hoc polymorphism*.

- Only some data types need provide instances.
- Different instances can have fundamentally *different* definitions.

Ad hoc polymorphism is therefore contrasted with *parametric polymorphism*.

Kinds of polymorphism

Each of the type classes we've looked at provide what is known as *ad hoc polymorphism.*

- Only some data types need provide instances.
- Different instances can have fundamentally *different* definitions.

Ad hoc polymorphism is therefore contrasted with *parametric polymorphism.*

- We can't constrain the instantiating data type ahead of time.

Kinds of polymorphism

Each of the type classes we've looked at provide what is known as *ad hoc polymorphism*.

- Only some data types need provide instances.
- Different instances can have fundamentally *different* definitions.

Ad hoc polymorphism is therefore contrasted with *parametric polymorphism*.

- We can't constrain the instantiating data type ahead of time.
- Parametric polymorphic functions come in families all of whose members act fundamentally *the same*.

Kinds of polymorphism

Each of the type classes we've looked at provide what is known as *ad hoc polymorphism*.

- Only some data types need provide instances.
- Different instances can have fundamentally *different* definitions.

Ad hoc polymorphism is therefore contrasted with *parametric polymorphism*.

- We can't constrain the instantiating data type ahead of time.
- Parametric polymorphic functions come in families all of whose members act fundamentally *the same*.
- Examples…

Foldable

`foldr` and `foldl` are actually methods of a type class `Foldable`.

Foldable

`foldr` and `foldl` are actually methods of a type class `Foldable`.

- Lists are foldable

Foldable

`foldr` and `foldl` are actually methods of a type class `Foldable`.

- Lists are foldable
- Maybe types are foldable. How?

Foldable

`foldr` and `foldl` are actually methods of a type class `Foldable`.

- Lists are foldable
- Maybe types are foldable. How?
- Trees are foldable. How?

Functors

Functors provide another class, with a single method `fmap`.

Functors

Functors provide another class, with a single method `fmap`.

- Lists are functors.

Functors

Functors provide another class, with a single method `fmap`.

- Lists are functors.
- `Maybe` is a functor.

Functors

Functors provide another class, with a single method `fmap`.

- Lists are functors.
- `Maybe` is a functor.
- `Either a` is a functor.

Functors

Functors provide another class, with a single method `fmap`.

- Lists are functors.
- `Maybe` is a functor.
- `Either a` is a functor.
- `Tree` is a functor.