Parametric Polymorphism in Haskell

Ben Deane

30th October 2014

Disclaimer

I suspect there are several people in the audience who know more about this than I do!

This is what I think I know. (Broadly on the topic of Parametric Polymorphism.)

Two kinds of polymorphism

Parametric

- Type variables (a, b, etc)

Ad-hoc

- Type classes (Eq, Num, etc)

Two kinds of polymorphism

Parametric

- Type variables (a, b, etc)
- Universal

Ad-hoc

- Type classes (Eq, Num, etc)
- Existential?

Two kinds of polymorphism

Parametric

- Type variables (a, b, etc)
- Universal
- Compile-time

Ad-hoc

- Type classes (Eq, Num, etc)
- Existential?
- Runtime (also)

Two kinds of polymorphism

Parametric

- Type variables (a, b, etc)
- Universal
- Compile-time
- C++ templates, Java generics

Ad-hoc

- Type classes (Eq, Num, etc)
- Existential?
- Runtime (also)
- Classical ("normal" OO)

Polymorphic Datatypes

For example. . .

```
data Maybe a = Nothing | Just a

data List a = Nil | Cons a (List a)

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

Polymorphic Functions

For example...

```
reverse :: [a] -> [a]

fst :: (a,b) -> a

id :: a -> a
```

Universally quantified

- Work over all types
- Assume nothing behaviour-wise
- Parametricity
  - Intuitively, all instances act the same way
  - Theorems for free
  - eg. `reverse . map f` ⇔ `map f . reverse`

Totality

Partial functions

```haskell
head :: [a] -> a

tail :: [a] -> [a]
```

What happens when `a` is `[]`?

These are not *total functions*.
They are undefined for some inputs.

Type Inference and Unification

Unification: the process of solving a system of equations in type variables.

and

Inference: Why you thought you meant `Int -> Int` but the compiler knows you really meant `Num a => a -> a`.

An Odd Thought

- What can we say about `a -> a`?

An Odd Thought

- What can we say about `a -> a`?
  - We know it has to be `id`.

An Odd Thought

- What can we say about `a -> a`?
  - We know it has to be `id`.
- What can we say about `Int -> Int`?

An Odd Thought

- What can we say about `a -> a`?
  - We know it has to be `id`.
- What can we say about `Int -> Int`?
  - Almost nothing!

An Odd Thought

- What can we say about `a -> a`?
  - We know it has to be `id`.
- What can we say about `Int -> Int`?
  - Almost nothing!
- Even though we know more about `Int` than about `a`!

Types as Sets

- The set of representable values
- How many values inhabit the type
- Characterised by cardinality of the set

For example

0	? (in some languages, `void`)
1	`a -> a`
2	`Bool`
2^{64}	`Int`

Sum Types and Product Types

- Sum types represent alternation

For example

$a + b$ ⇔ `Either a b`

$1 + a$ ⇔ `Maybe a`

Sum Types and Product Types

- Product types represent composition

For example

a * b ⇔ `(a,b)`

a * 2 ⇔ `(a,Bool)`

What about function types?

- They equate to a power function
  - $b^a \Leftrightarrow$ `a -> b`
- (For concrete types, not type variables)

From CIS194 2014

Week 4, Exercise 5

```
-- How many distinct functions inhabit this type?
ex5 :: Bool -> Bool
-- Answer: 4
ex5   = const True
ex5_2 = const False
ex5_3 = id
ex5_4 = complement
-- Using type algebra:
-- Bool -> Bool
-- => 2 -> 2
-- => 2^2 = 4
```

Further Reading/Watching

- Theorems for Free - Philip Wadler
- The Algebra of Algebraic Data Types - Chris Taylor
  - London HUG video
- Adventures with Types in Haskell - Simon Peyton-Jones