Quantification in Haskell

John Wiegley

14 Mar 2017

Overview

Universals

Math

$$\forall a, a$$

Universals

Math

$$\forall a, a$$

Haskell

```
forall a. a
```

Meaning

As a universal

```
forall a. a
```

Meaning

As a universal

```
forall a. a
```

In other words…

```
undefined
```

GHC flags

-Wunused-foralls

Emits a warning in the specific case that a user writes explicit forall syntax with unused type variables.

GHC flags

-Wunused-foralls

Emits a warning in the specific case that a user writes explicit forall syntax with unused type variables.

-fprint-explicit-foralls

Makes GHC print explicit forall quantification at the top level of a type; normally this is suppressed.

ExplicitForAll

```haskell
{-# LANGUAGE ExplicitForAll #-}

foo :: forall a b. a -> b
foo = undefined
```

ExplicitForAll

```haskell
{-# LANGUAGE ExplicitForAll #-}

foo :: forall a. a -> forall b. b
foo = undefined
```

RankNTypes

```haskell
{-# LANGUAGE RankNTypes #-}

bar :: (forall r s. r -> s) -> a -> b
bar f a = f a
```

RankNTypes (Lens)

```haskell
type Lens' s a =
  forall f. Functor f
    => (a -> f a) -> s -> f s

_fst :: Lens' (Int, Int) Int
_fst f (x, y) = (,y) <$> f x

_fst :: Functor f
     => (Int -> f Int)
     -> (Int, Int)
     -> f (Int, Int)
```

RankNTypes (Lens)

```
type Lens' s a =
  forall f. Functor f
    => (a -> f a) -> s -> f s

hmm :: Lens' (Int, Int) Int
    -> (Int, Int) -> Int
hmm l p = getConst $ l Const p

hmm :: (forall f. Functor f
          => (Int -> f Int)
          -> (Int, Int)
          -> f (Int, Int))
    -> (Int, Int)
    -> Int
```

ExistentialQuantification

```haskell
{-# LANGUAGE ExistentialQuantification #-}

data Exists = forall a. Exists a
```

ExistentialQuantification

```haskell
{-# LANGUAGE ExistentialQuantification #-}

data Machine i log o = forall s. Machine
  { monitorState :: s
  , monitorFunc  ::
      i -> StateT s (Writer [log]) o
  }
```

RankNTypes

We'll come back to why this works, but we can use the *final encoding* of the universal to represent an existential.

```haskell
{-# LANGUAGE RankNTypes #-}

newtype Exists = Exists {
  getExists ::
    forall r. (forall a. a -> r) -> r
}
```

GADTSyntax or GADTs

GADT syntax can also be used to encode existentials, without needing the full power of GADTs.

```haskell
{-# LANGUAGE GADTSyntax #-}

data Exists where
  Exists :: a -> Exists
```

ScopedTypeVariables

```haskell
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}

baz :: forall s. Reifies s Int
    => Tagged s Int -> Int
baz (Tagged n) =
  n + reflect (Proxy :: Proxy s)
```

ImpredicativeTypes (Avoid!)

```haskell
{-# LANGUAGE ImpredicativeTypes #-}

type T = (Int, forall a. a -> Int)
```

ImpredicativeTypes (Avoid!)

```
{-# LANGUAGE ImpredicativeTypes #-}

type TLens = (Int, Lens' (Int, Int) Int)
```

ImpredicativeTypes (Solution)

```haskell
{-# LANGUAGE RankNTypes #-}

newtype Wrapped r = Wrapped {
  getWrapped :: forall a. a -> r
}

type T = (Int, Wrapped Int)
```

Negation

Math

$$\forall a, \neg a$$

Negation

Math

$$\forall a, \neg a$$

Haskell

```
forall a r. a -> r
```

Existentials

Math

$$\exists a, a$$

Existentials

Math

$$\exists a, a$$

Haskell?

```
exists a. a
```

Existentials

Haskell

```haskell
forall r. (forall a. a -> r) -> r
```

Relationships

$$
\begin{aligned}
\forall a, a &\iff \neg\exists a, \neg a \\
\exists a, a &\iff \neg\forall a, \neg a \\
\neg\forall a, a &\iff \exists a, \neg a \\
\neg\exists a, a &\iff \forall a, \neg a
\end{aligned}
$$

Proofs available upon request…

Derivation

$$\begin{aligned}\exists a, a &= \neg\forall a, \neg a \\ &= \forall r, (\forall a, \neg a) \to r \\ &= \forall r, (\forall a, a \to r) \to r\end{aligned}$$

Another derivation

$$
\begin{aligned}
\texttt{a} &\cong \texttt{Id a} \\
&\cong \texttt{Yoneda Id a} \\
&\cong \texttt{Ran Id Id a} \\
&\cong \texttt{forall r, (a} \to \texttt{Id r)} \to \texttt{Id r} \\
&\cong \texttt{forall r, (a} \to \texttt{r)} \to \texttt{r}
\end{aligned}
$$

Be careful of placement

Not the same as undefined

$$\forall a, a \not\cong \forall r, (\forall a, a \to r) \to r$$

Be careful of placement

Not the same as undefined

$$\forall a, a \ \ncong \ \forall r, (\forall a, a \to r) \to r$$

Haskell

```
works :: forall r. (forall a. a -> r) -> r
works k = k (10 :: Int)
```

Another undefined

undefined, finally encoded

$$\forall a, a \;\cong\; \forall a, \forall r, (a \to r) \to r$$

Another undefined

undefined, finally encoded

$$\forall a, a \;\cong\; \forall a, \forall r, (a \to r) \to r$$

Haskell

```
impossible :: forall a r. (a -> r) -> r
impossible k = k (10 :: Int)
```

Generic programming

Concrete

```
sort :: [Int] -> [Int]
```

Generic programming

Concrete

```haskell
sort :: [Int] -> [Int]
```

General

```haskell
sort :: Ord a => [a] -> [a]
```

Generic programming (C++)

Concrete

```
void stable_sort(
  std::vector<Int>::iterator,
  std::vector<Int>::iterator
);
```

Generic programming (C++)

`RandomIterator` must meet the requirements of `ValueSwappable` and `RandomAccessIterator`.

General

```
template <typename RandomIterator>
void stable_sort(RandomIterator first,
                 RandomIterator last);
```

Generic programming (Java)

Concrete

```
class MySorter {
  public static void sort(List<Int> list);
};
```

Generic programming (Java)

General

```
class MySorter {
  public static
    <T extends Comparable<? super T>>
      void sort(List<T> list);
};
```

Information hiding

Objects (ala OOP) are built on existentials.[1]See the section on *Existential Objects* in TAPL.}

[1]{

Haskell objects

```haskell
data Object = forall a. Real a => Object a

add :: Object -> Object -> Object
add (Object x) (Object y) =
  Object (toRational x + toRational y)

example :: (forall a. Real a => a -> r) -> r
example k =
  case add (Object (10 :: Int))
           (Object (1.0 :: Float)) of
      Object n -> k n
```

But not this. . .

```haskell
bad_example :: forall a. Real a => a
bad_example =
  case add (Object (10 :: Int))
           (Object (1.0 :: Float)) of
     Object n -> n
```

The ST Monad

Over to Emacs…

Parametricity

```haskell
myMap :: forall a b. (a -> b) -> [a] -> [b]

myMap f (x:xs) = f x : myMap f xs
myMap f _ = []
```

Parametricity

Gives rise to the following law, that *no implementation may avoid*:

Free Theorem for myMap

$$\mathit{map}\ f \circ \mathit{myMap}\ g = \mathit{myMap}\ f \circ \mathit{map}\ g$$

Parametricity

The more general a function is, the more it's restricted to information in its own type.