How do we bind type variables?

How should we bind type variables?

```haskell
prefix :: a → [[a]] → [[a]]
prefix x yss = map xcons yss
  where xcons ys = x : ys
```

```
prefix :: a → [[a]] → [[a]]
prefix x yss = map xcons yss
  where xcons :: [a] → [a]
        xcons ys = x : ys
```

```
prefix :: a → [[a]] → [[a]]
prefix x yss = map xcons yss
  where xcons :: [a] → [a]
        xcons ys = x : ys
```

but I want a, not a1!

```
Couldn't match ‘a1’ with ‘a’
‘a1’ is bound in
  xcons :: ∀ a1. [a1] -> [a1]
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
  prefix :: ∀ a. a → [[a]] → [[a]]
  prefix x yss = map xcons yss
    where xcons :: [a] → [a]
          xcons ys = x : ys
```

```
Ok, one module loaded.
```

Type signatures are useful

Goal:
Allow a type signature on any expression

Type signatures are useful

- type-class ambiguity

```
show :: Show a ⇒ a → String
read :: Read a ⇒ String → a


normalize :: String → String
normalize = show . read
```

what type to parse into?

Type signatures are useful

- type-class ambiguity
- polymorphic recursion

```
data T a = Leaf a
         | Node (T [a]) (T [a])
```

type signature is necessary

```
leaves :: T a → [a]
leaves (Leaf x) = [x]
leaves (Node t1 t2)
  = concat (leaves t1 ++ leaves t2)
```

Type signatures are useful

- type-class ambiguity
- polymorphic recursion
- higher-rank types

Scrap Your Boilerplate [TLDI '03]:

```
everywhere
  :: (∀ a. Data a ⇒ a → a)
  →   ∀ a. Data a ⇒ a → a
```

type signature is necessary

Type signatures are useful

- type-class ambiguity
- polymorphic recursion
- higher-rank types
- GADTs

```
data G a where
  MkInt :: G Int
  MkFun :: G (Int → Int)

matchG :: G a → a
matchG MkInt = 5
matchG MkFun = (10+)
```

Type signatures are useful

- type-class ambiguity
- polymorphic recursion
- higher-rank types
- GADTs

```
data G a where
  MkInt :: G Int
  MkFun :: G (Int → Int)
```

type signature is necessary

```
matchG :: G a → a
matchG MkInt = 5
matchG MkFun = (10+)
```

Type signatures are useful

- type-class ambiguity
- polymorphic recursion
- higher-rank types
- GADTs
- inherent ambiguity

```
type family F a
ambig :: Typeable a ⇒ F a → Int

test :: Char → Int
test x = ambig x
```

no way to infer a

Type signatures are useful

Goal:
Allow a type signature on any expression

Solution:
ScopedTypeVariables

ScopedTypeVariables

```
prefix :: ∀ a. a → [[a]] → [[a]]
prefix x yss = map xcons yss
  where xcons :: [a] → [a]
        xcons ys = x : ys
```

or

```
prefix (x::a) yss = map xcons yss
  where xcons :: [a] → [a]
        xcons ys = x : ys
```

Pattern Signature

ScopedTypeVariables

```
prefix (x::a) yss = map xcons yss
  where xcons :: [a] → [a]
        xcons ys = 1 : x : ys
```

```
Ok, one module loaded.
λ> :t prefix
prefix ::
  Num a ⇒ a → [[a]] → [[a]]
```

ScopedTypeVariables

```
prefix (x::a) yss = map xcons yss
  where xcons :: [a] → [a]
        xcons ys = True : x : ys
```

```
Couldn't match a with Bool
```

Arbitrary Rule: type variables
?
must be *variables*

What is the specification of ScopedTypeVariables anyway?

Contribution: Typing rules!

Existentials

```
data Ticker where
  MkT :: ∀ a. a → (a → a)
            → (a → Int) → Ticker

    tick :: Ticker → Ticker
    tick (MkT val upd toInt)
      = MkT newVal upd toInt
      where newVal = upd val
```

existential

Existentials

```
data Ticker where
  MkT :: ∀ a. a → (a → a)
             → (a → Int) → Ticker

    tick :: Ticker → Ticker
    tick (MkT val upd toInt)
      = MkT newVal upd toInt
      where newVal :: a
            newVal = upd val
```

what is this?

Existentials

```
data Ticker where
  MkT :: ∀ a. a → (a → a)
                → (a → Int) → Ticker

    tick :: Ticker → Ticker
    tick (MkT (val::a) upd toInt)
      = MkT newVal upd toInt
      where newVal :: a
            newVal = upd val
```

no other way to bind a

Existentials

```
data Elab where
  MkE :: Show a
      ⇒ [Maybe (Tree (a, Int))]
      → Elab
```

a pattern signature to
bind a would be long

Existentials

```
type family F a
data ExF where
  MkF :: Typeable a ⇒ F a → ExF
```

a pattern signature to bind a would be ~~long~~ impossible

Type signatures are useful

Goal:
Allow a type signature on any expression

Solution:
ScopedTypeVariables

Partial Solution:

ScopedTypeVariables

Contribution:

Pattern type applications

Pattern type applications

```
data Ticker where
  MkT :: ∀ a. a → (a → a)
             → (a → Int) → Ticker

    tick :: Ticker → Ticker
    tick (MkT @a val upd toInt)
      = MkT newVal upd toInt
      where newVal :: a
            newVal = upd val
```

Pattern type applications

Explicit binding of
type variables
always works

Universals vs Existentials

```
data UnivEx a where
  MkUE :: a → b → UnivEx a
```

universal existential

```
case ue of
  MkUE @a @b x y → ...
```

:: UnivEx τ

We always know τ here.
Why bind it to a?

Universals vs Existentials

we always know τ here.
why bind it to a?

Uniformity

```
data Confused a where
  MkC :: a ~ b ⇒ b → Confused a
```

what is existential? ¯_(ツ)_/¯

Universals & Existentials

…

$$\frac{\begin{array}{l} K : \forall\, a_{1..m}.\ Q \Rightarrow \eta_{1..n} \rightarrow T\ \varphi_{1..j} \\ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m} \end{array}}{\Gamma \vdash K\ @\tau_{1..m}\ p_{1..n} : T\ \sigma_{1..j}}$$

Universals & Existentials

…

$$\frac{K : \forall\, a_{1..m}.\ Q \Rightarrow \eta_{1..n} \to T\ \varphi_{1..j} \qquad \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m}}{\Gamma \vdash K\ @\tau_{1..m}\ p_{1..n} : T\ \sigma_{1..j}}$$

type applications in a pattern

Universals & Existentials

...

$$\frac{K : \forall\, a_{1..m}.\ Q \Rightarrow \eta_{1..n} \to T\ \varphi_{1..j} \qquad \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m}}{\Gamma \vdash K\ @\tau_{1..m}\ p_{1..n} : T\ \sigma_{1..j}}$$

expected result type arguments

Universals & Existentials

quantified type variables

…

$$\frac{K : \forall\, a_{1..m}.\; Q \Rightarrow \eta_{1..n} \rightarrow T\, \varphi_{1..j} \qquad \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m}}{\Gamma \vdash K\, @\tau_{1..m}\; p_{1..n} : T\, \sigma_{1..j}}$$

Universals & Existentials

constructor constraint

…

$$K : \forall a_{1..m}.\ Q \Rightarrow \eta_{1..n} \to T\ \varphi_{1..j}$$

$$\frac{\Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m}}{\Gamma \vdash K\ @\tau_{1..m}\ p_{1..n} : T\ \sigma_{1..j}}$$

Universals & Existentials

result type arguments

...

$$K : \forall\, a_{1..m}.\; Q \Rightarrow \eta_{1..n} \rightarrow T\, \varphi_{1..j}$$

$$\frac{\Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m}}{\Gamma \vdash K\; @\tau_{1..m}\; p_{1..n} : T\, \sigma_{1..j}}$$

Universals & Existentials

"assuming the GADT equalities..."

...

$$K : \forall\, a_{1..m}.\ Q \Rightarrow \eta_{1..n} \to T\ \varphi_{1..j}$$

$$\frac{\Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \Vdash \tau_{1..m} \sim a_{1..m}}{\Gamma \vdash K\ @\tau_{1..m}\ p_{1..n} : T\ \sigma_{1..j}}$$

"we know the form of the type applications"

Example

```
data Example where
  MkEx :: ∀ a b.
     (a ~ Maybe b) ⇒ Example
case x :: Example of
✓  MkEx @a @b                 → ...
✓  MkEx @(Maybe b) @b         → ...
✓  MkEx @(Maybe b)            → ...
✗  MkEx @a @(Maybe b)         → ...
```

Why this behavior?

It's exactly how pattern signatures would work.

In the paper:
full specification
with typing rules

Upshot: we can easily drop
the variable restriction

Next Steps

Implementation:
My Nguyen

Binding type variables
in λ-expressions
(in paper appendix)

Type Variables in Patterns

Richard A. Eisenberg
Bryn Mawr College
rae@cs.brynmawr.edu

Joachim Breitner
DFINITY Foundation
joachim@dfinity.org

Simon Peyton Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Friday, September 28, 2018
Haskell Symposium
St. Louis, MO, USA