How do we bind type variables?
How should we bind type variables?
prefix :: a → [[a]] → [[a]]
prefix x yss = map xcons yss
  where xcons ys = x : ys
prefix :: a → [[a]] → [[a]]

prefix x yss = map xconss yss

  where xconss :: [a] → [a]
xconss ys = x : ys
prefix :: a \to [[a]] \to [[a]]

prefix x yss = map xcons yss

where xcons :: [a] \to [a]
xcons ys = x : ys

but I want a, not a1!

Couldn't match ‘a1’ with ‘a’
‘a1’ is bound in
xcons :: \forall a1. [a1] \to [a1]
prefix :: \( \forall a. a \rightarrow [[a]] \rightarrow [[a]] \)
prefix \( x \) \( yss \) = map \( xcons \) \( yss \)

where \( xcons :: [a] \rightarrow [a] \)
\( xcons \) \( ys \) = \( x \) : \( ys \)

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

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

Type signature is necessary

```haskell
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

\[ (\forall a. \text{Data } a \Rightarrow a \to a) \]

\to \quad (\forall a. \text{Data } a \Rightarrow a \to a)

*type signature is necessary*
Type signatures are useful

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

```haskell
data G a where
  MkInt :: G Int
  MkFun :: G (Int -> Int)

matchG :: G a a -> a
matchG MkInt = 5
matchG MkFun = (10+)
```
Type signatures are useful

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

```haskell
data G a where
  MkInt :: G Int
  MkFun :: G (Int \rightarrow Int)
```

Type signature is necessary

```haskell
matchG :: G a \rightarrow 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 a → Int

test :: Char → Int

no way to infer a
Type signatures are useful

Goal:
Allow a type signature on any expression
Solution:
ScopedTypeVariables
ScopedTypeVariables

prefix :: \( \forall a. a \rightarrow [[a]] \rightarrow [[a]] \)

prefix x yss = map xcons yss

where xcons :: [a] \rightarrow [a]

xcons ys = x : ys

or

prefix (x::a) yss = map xcons yss

where xcons :: [a] \rightarrow [a]

xcons ys = x : ys
ScopedTypeVariables

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

Ok, one module loaded.

```haskell
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 \rightarrow (a \rightarrow a)

existential 

\rightarrow (a \rightarrow \text{Int}) \rightarrow \text{Ticker}

tick :: Ticker \rightarrow Ticker

tick (MkT \text{val} \text{upd} \text{toInt})

= MkT \text{newVal} \text{upd} \text{toInt}

where \text{newVal} = \text{upd} \text{val}
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?
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 \Rightarrow F a \rightarrow 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
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

:: UnivEx τ

case ue of

MkUE @a @b x y → ...

we always know τ here. Why bind it to a?
Universals vs Existentials

we always know $\tau$ here. Why bind it to $a$?

Uniformity

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

what is existential?  \_\_(ツ)_\_/\
...

\[ K : \forall a_{1..m}. \ Q \Rightarrow \eta_{1..n} \rightarrow T \varphi_{1..j} \]
\[ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \vdash \tau_{1..m} \sim a_{1..m} \]
\[ \Gamma \vdash K \, \@ \tau_{1..m} \, \rho_{1..n} : T \, \sigma_{1..j} \]
Universals & Existentials

... 

\[ K : \forall a_{1..m}. Q \Rightarrow \eta_{1..n} \rightarrow T \varphi_{1..j} \]

\[ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \models \tau_{1..m} \sim a_{1..m} \]

\[ \Gamma \vdash K @ \tau_{1..m} \, p_{1..n} : T \sigma_{1..j} \]

type applications in a pattern
Universals & Existentials

\[ \Gamma \vdash K \forall a_{1..m}. Q \Rightarrow \eta_{1..n} \rightarrow T \varphi_{1..j} \]
\[ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \vdash \tau_{1..m} \sim a_{1..m} \]

\[ \Gamma \vdash K \@ \tau_{1..m} \rho_{1..n} : T \sigma_{1..j} \]

expected result type arguments
Universals & Existentials

quantified type variables

\[ K : \forall a_{1..m}. \ Q \Rightarrow \eta_{1..n} \rightarrow T \varphi_{1..j} \]
\[ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \vdash \tau_{1..m} \sim a_{1..m} \]
\[ \Gamma \vdash K @ \tau_{1..m} p_{1..n} : T \sigma_{1..j} \]
Universals & Existentials

constructor constraint

\[ \Gamma, Q, \phi_{1..j} \sim \sigma_{1..j} \vdash \tau_{1..m} \sim a_{1..m} \]

\[ \Gamma \vdash K \at \tau_{1..m} \ p_{1..n} : T \ \sigma_{1..j} \]
Universals & Existentials

result type arguments

\[ K : \forall a_{1..m}. Q \Rightarrow \eta_{1..n} \rightarrow T \varphi_{1..j} \]
\[ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \vdash \tau_{1..m} \sim a_{1..m} \]
\[ \Gamma \vdash K \@ \tau_{1..m} \rho_{1..n} : T \sigma_{1..j} \]
Universals & Existentials

"assuming the GADT equalities..."

\[ K : \forall a_{1..m}. \ Q \Rightarrow \eta_{1..n} \rightarrow T \varphi_{1..j} \]
\[ \Gamma, Q, \varphi_{1..j} \sim \sigma_{1..j} \models \tau_{1..m} \sim a_{1..m} \]
\[ \Gamma \vdash K @ \tau_{1..m} \rho_{1..n} : T \sigma_{1..j} \]

"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 \(\lambda\)-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