BRYN MAWR COLLEGE

tweag.io

Kind Inference for Datatypes

Ningning Xie [1] Richard A. Eisenberg [2,3] Bruno C. d. S. Oliveira [1]

POPL 2020

[1]The University of Hong Kong

[2]Bryn Mawr College

[3]Tweag I/O

Datatypes 101

- A datatype declaration defines a type, and the shape of each of the elements.

```
data List a = Nil | Cons a (List a)
  -- List :: ⋆ → ⋆                          type constructor
  -- Nil :: ∀a. List a                       data constructor
  -- Cons :: ∀a. a → List a → List a         data constructor
```

Datatypes 101

- A datatype declaration defines a type, and the shape of each of the elements.

```
data List a = Nil | Cons a (List a)
  -- List :: ⋆ → ⋆                         type constructor
  -- Nil :: ∀a. List a                     data constructor
  -- Cons :: ∀a. a → List a → List a       data constructor
```

- Kind inference.

```
List Int :: ⋆       -- accepted
List Maybe :: ?     -- rejected
  -- Maybe :: ⋆ → ⋆
```

Many Extensions

Haskell, Idris, Coq, Agda, ...

- Polymorphic Kinds
- Dependent Types
- ...

Many Extensions, But...

Haskell, Idris, Coq, Agda, ...

- Polymorphic Kinds
- Dependent Types
- ...

$$\textbf{data}\ X :: \forall a\ (b :: \star \to \star).\ a\ b \to \star$$

$$\textbf{data}\ Y :: \forall (c :: \mathit{Maybe\ Bool}).\ X\ c \to \star$$

Many Extensions, But...

Haskell, Idris, Coq, Agda, ...

- Polymorphic Kinds
- Dependent Types
- ...

$$\textbf{data}\ X :: \forall a\ (b :: \star \to \star).\ a\ b \to \star$$

$$\textbf{data}\ Y :: \forall (c :: \mathit{Maybe\ Bool}).\ X\ c \to \star$$

error: Couldn't match kind ...

Problem

- Which datatype declarations should be accepted?

Problem

- Which datatype declarations should be accepted?
- What kinds do accepted datatypes have?

Why It Is Interesting

- A **declarative specification** of datatypes guides programmers.

Why It Is Interesting

- A **declarative specification** of datatypes guides programmers.
- An **algorithm** helps inform the design of principled inference algorithms for datatypes.

Contributions

- **Kind inference for Haskell98**

  The first formalization of Haskell98's datatype declarations.

- **Kind inference for modern Haskell**

  A type and kind language that is unified and dependently typed.

- **Technical advances**

  Technical innovations important in the implementation of type-inference for dependent types.

Kind Inference for Haskell98

Language Features

kind	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2$

Language Features

$$\text{kind} \quad \kappa \quad ::= \quad \star \mid \kappa_1 \to \kappa_2$$

- Haskell98 supports mutual recursion.

```
data P a = MkP (Q a)
data Q a = MkQ (P a)
```

Language Features

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆?
  -- Tree :: (⋆ → ⋆) → ⋆?
  -- ... (infinitely many incomparable kinds)
```

Language Features

$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$

-- $\mathit{Tree} :: \star \to \star$?

-- $\mathit{Tree} :: (\star \to \star) \to \star$?

-- ... *(infinitely many incomparable kinds)*

```
Prelude> :kind Tree
Tree :: ?
```

It is possible that some parts of an inferred kind may not be fully determined by the corresponding definitions; in such cases, **a default of $\star$ is assumed**.

Haskell98 Report Sec.4.6

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆    winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆        winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆
```

winner!

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)

      mutual recursion

  -- a :: ⋆ → ⋆
  -- P1 :: (⋆ → ⋆) → ⋆
  -- P2 :: ⋆
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆      winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
-- a :: ⋆ → ⋆
-- P1 :: (⋆ → ⋆) → ⋆
-- P2 :: ⋆
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆        winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
-- a :: ⋆ → ⋆
-- P1 :: (⋆ → ⋆) → ⋆
-- P2 :: ⋆
```

```
data P1 a = MkP1 P2 ✗
data P2   = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆    winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)

      mutual recursion

 -- a :: ⋆ → ⋆
 -- P1 :: (⋆ → ⋆) → ⋆
 -- P2 :: ⋆
```

```
data P1 a = MkP1
data P2   = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆
```

winner!

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
-- a :: ⋆ → ⋆
-- P1 :: (⋆ → ⋆) → ⋆
-- P2 :: ⋆
```

```
data P1 a = MkP1
data P2   = MkP2 (P1 Maybe)
```

no mutual recursion

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)

      mutual recursion

 -- a :: ⋆ → ⋆
 -- P1 :: (⋆ → ⋆) → ⋆
 -- P2 :: ⋆
```

```
data P1 a = MkP1

      no mutual recursion

data P2 = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆
```

winner!

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
-- a :: ⋆ → ⋆
-- P1 :: (⋆ → ⋆) → ⋆
-- P2 :: ⋆
```

```
data P1 a = MkP1
```

no mutual recursion

```
data P2 = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆
```

winner!

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
-- a :: ⋆ → ⋆
-- P1 :: (⋆ → ⋆) → ⋆
-- P2 :: ⋆
```

```
data P1 a = MkP1
```

no mutual recursion

defaulting a :: ⋆

```
data P2 = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
-- a :: ⋆ → ⋆
-- P1 :: (⋆ → ⋆) → ⋆
-- P2 :: ⋆
```

```
data P1 a = MkP1
  -- P1 :: ⋆ → ⋆
```

no mutual recursion

defaulting a :: ⋆

```
data P2 = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆
```

winner!

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)
```

mutual recursion

```
  -- a :: ⋆ → ⋆
  -- P1 :: (⋆ → ⋆) → ⋆
  -- P2 :: ⋆
```

```
data P1 a = MkP1
  -- P1 :: ⋆ → ⋆
```

no mutual recursion

defaulting a :: ⋆

```
data P2 = MkP2 (P1 Maybe)
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)

      mutual recursion

 -- a :: ⋆ → ⋆
 -- P1 :: (⋆ → ⋆) → ⋆
 -- P2 :: ⋆
```

```
data P1 a = MkP1
  -- P1 :: ⋆ → ⋆

     no mutual recursion
       defaulting a :: ⋆

data P2 = MkP2 (P1 Maybe)
     -- rejected
```

Defaulting

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ⋆ → ⋆     winner!
```

```
data P1 a = MkP1 P2
data P2   = MkP2 (P1 Maybe)

        mutual recursion

  -- a :: ⋆ → ⋆
  -- P1 :: (⋆ → ⋆) → ⋆
  -- P2 :: ⋆
```

```
data P1 a = MkP1
  -- P1 :: ⋆ → ⋆

     no mutual recursion
       defaulting a :: ⋆

data P2 = MkP2 (P1 Maybe)
     -- rejected
```

More about Haskell98

- **A declarative specification** of the datatype language.
- **A sound algorithmic system** using *context extension* approaches (Gundry et al., 2010; Dunfield and Krishnaswami, 2013).
  - Defaulting fits naturally in the approach.
  - Consequences and design alternatives of defaulting.
- **Restore completeness and principality** by endowing the declarative system with *kind parameters* (Garcia and Cimini, 2015).

Kind Inference for Modern Haskell

The Kind Language

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\, \tau_2 \mid \tau_1\ @\tau_2\ \mid\!\rightarrow$

The Kind Language

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Kind polymorphism

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
```

The Kind Language

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2\ \mid \to$

- Kind polymorphism

```
data Tree a = Leaf | Fork (Tree a) (Tree a)

  -- Tree :: ∀k. k → ⋆
```

The Kind Language

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2\ \mid \rightarrow$

- Kind polymorphism

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ∀k. k → ⋆
```

- The type and kind language is unified, dependently typed

```
data D :: ∀(k :: ⋆) (a :: k). Tree a → ⋆
```

The Kind Language

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Kind polymorphism

```
data Tree a = Leaf | Fork (Tree a) (Tree a)
  -- Tree :: ∀k. k → ⋆
```

- The type and kind language is unified, dependently typed

```
data D :: ∀(k :: ⋆) (a :: k). Tree a → ⋆
```

- Visible kind application, lifted from visible type application (Eisenberg et al., 2016)

Language Features

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

Language Features

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Open kind signature: a kind signature can contain free variables.

```
data D :: Tree @ k  a → ⋆
```

Language Features

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Open kind signature: a kind signature can contain free variables.

```
data D :: Tree @k a → ⋆
  -- D :: ∀(k :: ⋆) (a :: k). Tree @k a → ⋆
```

Language Features

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Open kind signature: a kind signature can contain free variables.

```
data D :: Tree @ k  a → ⋆
  -- D :: ∀(k :: ⋆) (a :: k). Tree @k a → ⋆
```

- Polymorphic recursion.

Language Features

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Open kind signature: a kind signature can contain free variables.

```
data D :: Tree @k a → ⋆
  -- D :: ∀(k :: ⋆) (a :: k). Tree @k a → ⋆
```

- Polymorphic recursion.
- Existential quantification for data constructors.

Language Features

kind, type	σ, K	$::=$	$\forall a.\ \sigma \mid \forall a : \kappa.\ \sigma \mid \tau$
mono-	τ, κ	$::=$	$\star \mid \mathsf{Int} \mid a \mid T \mid \tau_1\ \tau_2 \mid \tau_1\ @\tau_2 \mid \rightarrow$

- Open kind signature: a k ... ee variables.

```
data D :: Tree @ k ...
  -- D :: ∀(k :: ⋆) (a :: k). Tree @k a → ⋆
```

Challenges!

- Polymorphic recursion.
- Existential quantification for data constructors.

Challenge #1

In which order do we put free variables in the ordered type context before kinding?

$$\textbf{data}\ Q :: \forall (a :: (f\ \ b))\ (c :: k).\ f\ c \to \star$$

Challenge #1

In which order do we put free variables in the ordered type context before kinding?

```
data Q :: ∀(a :: (f b)) (c :: k). f c → ⋆

-- well-scoped:
    f, b, k ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
```

Challenge #1

In which order do we put free variables in the ordered type context before kinding?

```
data Q :: ∀(a :: (f b)) (c :: k). f c → ⋆

-- well-scoped:
    f, b, k ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    f, k, b ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    k, b, f ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    ...
```

Challenge #1

In which order do we put free variables in the ordered type context before kinding?

```
data Q :: ∀(a :: (f b)) (c :: k). f c → ⋆

-- well-scoped:
    f, b, k ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    f, k, b ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    k, b, f ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    ...
```

$f :: k \to \star$

$b :: k$

$k :: \star$

Challenge #1

In which order do we put free variables in the ordered type context before kinding?

```
data Q :: ∀(a :: (f b)) (c :: k). f c → ⋆

-- well-scoped:
    f, b, k ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    f, k, b ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    k, b, f ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆
    ...
```

$f :: k \to \star$

$b :: k$

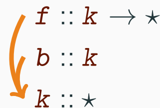
$k :: \star$

Challenge #1

In which order do we put free variables in the ordered type context before kinding?

```
data Q :: ∀(a :: (f b)) (c :: k). f c → ⋆

-- well-scoped:
    f, b, k ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆ ✗
    f, k, b ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆ ✗
    k, b, f ⊢ᵏ ∀(a :: (f b)) (c :: k). f c → ⋆ ✓
    ...
```



Challenge #1

In which order do we put free variables in the ordered type context before kinding?

$$\texttt{data}\ Q :: \forall (a :: (f\ \ b))\ (c :: k).\ f\ c \to \star$$

`-- well-scoped:`

$$f, b, k \vdash^{k} \forall (a :: (f\ b))\ (c :: k).\ f\ c \to \star\ ✗$$

$$f, k, b \vdash^{k} \forall (a :: (f\ b))\ (c :: k).\ f\ c \to \star\ ✗$$

$$k, b, f \vdash^{k} \forall (a :: (f\ b))\ (c :: k).\ f\ c \to \star\ ✓$$

$\ldots$

$$f :: k \to \star$$

$$b :: k$$

$$k :: \star$$

We *cannot* know the dependency order before kinding.

Our Solution

Local scopes, sets of variables that may be reordered.

$$\Delta, \Theta \quad ::= \quad \bullet \mid \Delta, a : \kappa \mid \Delta, T : K \mid \Delta, \{\Delta'\} \mid \ldots$$

Moving judgment reorders local scopes.

$$\boxed{\Delta_1 \mathbin{+\!\!+}^{\mathsf{mv}} \Delta_2 \rightsquigarrow \Theta}$$

Challenge #2

How to generalize unification variables?

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\ k \to k \to \star$$

Challenge #2

How to generalize unification variables? Given

$$SameKind :: \forall (k :: \star).\, k \to k \to \star$$

Challenge #2

How to generalize unification variables? Given

$$SameKind :: \forall (k :: \star).\, k \to k \to \star$$

```
data Q :: ∀a. SameKind a a
```

Challenge #2

How to generalize unification variables? Given

$$SameKind :: \forall (k :: \star).\, k \to k \to \star$$

$\mathbf{data}\ Q :: \forall (a :: \widehat{\beta}).\, SameKind\ a\ a$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall(k :: \star).\, k \to k \to \star$$

$\mathbf{data}\ Q :: \forall(a :: \widehat{\beta}).\, \mathit{SameKind}\ a\ a$

$k \mapsto \widehat{\beta}$

Challenge #2

How to generalize unification variables? Given

$$SameKind :: \forall(k :: \star).\, k \to k \to \star$$

$\mathbf{data}\ Q :: \forall(a :: \widehat{\beta}).\, SameKind\ a\ a$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

Challenge #2

How to generalize unification variables? Given

$$SameKind :: \forall (k :: \star).\, k \to k \to \star$$

$\mathbf{data}\ Q :: \forall (a :: \widehat{\beta}).\ SameKind\ a\ a$

$k \mapsto \widehat{\beta}$
$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$\mathbf{data}\ Q :: \forall (b :: \star)\ (a :: b).$
$SameKind\ @b\ a\ a$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\, k \to k \to \star$$

$$\textbf{data}\ Q :: \forall (a :: \widehat{\beta}).\ \mathit{SameKind}\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$$\textbf{data}\ Q :: \forall (b :: \star)\ (a :: b).\ \mathit{SameKind}\ @b\ a\ a$$

$$\textbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$

$$\textbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ T1\ d \to \star$$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\, k \to k \to \star$$

$$\mathbf{data}\ Q :: \forall (a :: \widehat{\beta}).\, \mathit{SameKind}\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$$\mathbf{data}\ Q :: \forall (b :: \star)\ (a :: b).\ \mathit{SameKind}\ @b\ a\ a$$

$$\mathbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$

$$\mathbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ T1\ d \to \star$$

$k \mapsto (c \to \star)$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\, k \to k \to \star$$

$$\texttt{data}\ Q :: \forall (a :: \widehat{\beta}).\ \mathit{SameKind}\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$$\texttt{data}\ Q :: \forall (b :: \star)\ (a :: b).\ \mathit{SameKind}\ @b\ a\ a$$

$$\texttt{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$

$$\texttt{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ T1\ d \to \star$$

$k \mapsto (c \to \star) \qquad f \mapsto c$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\, k \to k \to \star$$

$$\mathbf{data}\ Q :: \forall (a :: \widehat{\beta}).\, \mathit{SameKind}\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$$\mathbf{data}\ Q :: \forall (b :: \star)\ (a :: b).$$
$$\mathit{SameKind}\ @b\ a\ a$$

$$\mathbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$
$$\mathbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).$$
$$\mathit{SameKind}\ T1\ d \to \star$$

$k \mapsto (c \to \star)$ $\quad f \mapsto c$

$a \mapsto \widehat{\beta}$ $\quad \widehat{\beta} :: c$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\, k \to k \to \star$$

$$\textbf{data}\ Q :: \forall (a :: \widehat{\beta}).\, \mathit{SameKind}\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$$\textbf{data}\ Q :: \forall (b :: \star)\ (a :: b).\ \mathit{SameKind}\ @b\ a\ a$$

$$\textbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$

$$\textbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ T1\ d \to \star$$

$k \mapsto (c \to \star)$ $\quad f \mapsto c$

$a \mapsto \widehat{\beta}$ $\quad \widehat{\beta} :: c$

$\Downarrow$ generalizes

$$\textbf{data}\ T2 :: \forall (b :: c)\ (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ @(c \to \star)\ (T1\ @c\ @b)\ a$$

Challenge #2

How to generalize unification variables? Given

$$\mathit{SameKind} :: \forall (k :: \star).\, k \to k \to \star$$

$$\textbf{data}\ Q :: \forall (a :: \widehat{\beta}).\ \mathit{SameKind}\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

$\Downarrow$ generalizes

$$\textbf{data}\ Q :: \forall (b :: \star)\ (a :: b).\ \mathit{SameKind}\ @b\ a\ a$$

$$\textbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$

$$\textbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ T1\ d \to \star$$

$k \mapsto (c \to \star)$ $\quad f \mapsto c$

$a \mapsto \widehat{\beta}$ $\quad \widehat{\beta} :: c$

$\Downarrow$ generalizes

$$\textbf{data}\ T2 :: \forall (b :: c)\ (c :: \star)\ (d :: c \to \star).\ \mathit{SameKind}\ @(c \to \star)\ (T1\ @c\ @b)\ a$$

```
-- (b::c) ill-typed, c out of scope
```

Challenge #2

How to generalize unification variables? Given

$$SameKind :: \forall (k :: \star).\, k \to k \to \star$$

$$\textbf{data}\ Q :: \forall (a :: \widehat{\beta}).\, SameKind\ a\ a$$

$k \mapsto \widehat{\beta}$

$\widehat{\beta} :: \star$

⇓ generalizes

$$\textbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\, f \to \star$$

$$\textbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ SameKind\ T1\ d \to \star$$

$k \mapsto (c \to \star)$ $\quad f \mapsto c$

$a \mapsto \widehat{\beta}$ $\quad \widehat{\beta} :: c$

⇓ generalizes

☞ How to handle unsolved unification variables with dependency on local variables?

(c :: ⋆) (d :: c → ⋆).

SameKind @(c → ⋆) (T1 @c @b) a

-- (b::c) ill-typed, c out of scope

Our Solution

Reject it!

Our Solution

Reject it! via **Quantification Check**.

Our Solution

Reject it! via **Quantification Check**.

$\forall(a :: k).\, \sigma$	after kinding σ, no unsolved unification variables can depend on a.

Our Solution

Reject it! via **Quantification Check**.

$\forall(a :: k).\ \sigma$ after kinding σ, no unsolved unification variables can depend on a.

$$\textbf{data}\ T1 :: \forall(f :: \star)\ (a :: f).\ f \to \star$$
$$\textbf{data}\ T2 :: \forall(c :: \star)\ (d :: c \to \star).\ SameKind\ T1\ d \to \star$$

$k \mapsto (c \to \star)$

$f \mapsto c$

$a \mapsto \widehat{\beta}$

$\widehat{\beta} :: c$

Our Solution

Reject it! via **Quantification Check**.

$\forall (a :: k).\, \sigma$ after kinding σ, no unsolved unification variables can depend on a.

$$\mathbf{data}\ T1 :: \forall (f :: \star)\ (a :: f).\ f \to \star$$
$$\mathbf{data}\ T2 :: \forall (c :: \star)\ (d :: c \to \star).\ SameKind\ T1\ d \to \star$$

$k \mapsto (c \to \star)$

$f \mapsto c$

$a \mapsto \widehat{\beta}$

$\widehat{\beta} :: c$ ✗

Our Solution

Reject it! via **Quantification Check**.

$\forall(a :: k).\,\sigma$ after kinding σ, no unsolved unification variables can depend on a.

$\mathbf{data}\ T1 :: \forall(f :: \star)\ (a :: f).\ f \to \star$
$\mathbf{data}\ T2 :: \forall(c :: \star)\ (d :: c \to \star).\ SameKind\ T1\ d \to \star$

$k \mapsto (c \to \star)$

$f \mapsto c$

$a \mapsto \widehat{\beta}$

$\widehat{\beta} :: c$ ✗

An unification variable, after kinding σ, is either

- solved; or
- since it doesn't depend on a, generalized outside a.

Challenge #3

Unification for a dependently typed language.

Challenge #3

Unification for a dependently typed language.

- **First-order unification**, for the algorithm to be *guessing-free* and *predicative* (Vytiniotis et al., 2011).

Challenge #3

Unification for a dependently typed language.

- **First-order unification**, for the algorithm to be *guessing-free* and *predicative* (Vytiniotis et al., 2011).
- Coen (2004) considers first-order unification, but only soundness is proved.

Challenge #3

Unification for a dependently typed language.

- **First-order unification**, for the algorithm to be *guessing-free* and *predicative* (Vytiniotis et al., 2011).
- Coen (2004) considers first-order unification, but only soundness is proved.
- A higher-order unification algorithm for Coq (Ziliani and Sozeau, 2015) favors syntactic equality by trying first-order unification. To our knowledge, they lack a correctness proof.

Unification

$$\widehat{\alpha} : \kappa_1, \widehat{\beta} : \kappa_2 \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta}$$

Unification

$$\widehat{\alpha} : \kappa_1, \widehat{\beta} : \kappa_2 \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta}$$

① Try solving $\widehat{\beta} = \widehat{\alpha}$

Unification

$$\widehat{\alpha} : \kappa_1, \widehat{\beta} : \kappa_2 \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta}$$

① Try solving $\widehat{\beta} = \widehat{\alpha}$

② May *not* be well-formed: κ_1 may *not* match κ_2.

Unification

$$\widehat{\alpha} : \kappa_1, \widehat{\beta} : \kappa_2 \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta}$$

① Try solving $\widehat{\beta} = \widehat{\alpha}$

② May *not* be well-formed: κ_1 may *not* match κ_2.

③ Try unifying $\kappa_1 \approx \kappa_2$ first.

Unification

$$\widehat{\alpha} : \kappa_1, \widehat{\beta} : \kappa_2 \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta}$$

① Try solving $\widehat{\beta} = \widehat{\alpha}$

② May *not* be well-formed: κ_1 may *not* match κ_2.

③ Try unifying $\kappa_1 \approx \kappa_2$ first.

④ Unifying a type recurs to its kind, how does it terminate?

Unification

$$\widehat{\alpha} : \kappa_1, \widehat{\beta} : \kappa_2 \Vdash^{\mathsf{u}} \widehat{\alpha} \approx \widehat{\beta}$$

① Try solving $\widehat{\beta} = \widehat{\alpha}$

② May *not* be well-formed: κ_1 may *not* match κ_2.

③ Try unifying $\kappa_1 \approx \kappa_2$ first.

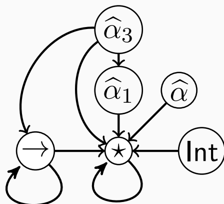
④ Unifying a type recurs to its kind, how does it terminate?

⑤ $\star : \star$ axiom.

Our Solution

Dependency graph visualizes the dependency information in an ordered context.

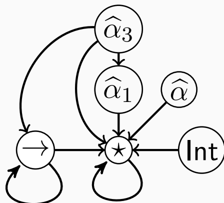
$\Delta = \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \to \widehat{\alpha}_2$



Our Solution

Dependency graph visualizes the dependency information in an ordered context.

$\Delta = \widehat{\alpha} : \star, \widehat{\alpha}_1 : \star, \widehat{\alpha}_2 : \star = \widehat{\alpha}_1, \widehat{\alpha}_3 : \star \to \widehat{\alpha}_2$



✓ We proved unification terminates.

More about modern Haskell

- **A declarative specification** of the datatype language.
- **A sound algorithmic system**.

  Principality: our algorithm does not infer every kind acceptable by the declarative system, but the kinds it does infer are always the best (principal) one (Vytiniotis et al., 2011).
- **Discussion of language extensions**.

  Higher-rank polymorphism, GADTs, type families, visible dependent quantification, datatype promotion, partial type signature.

Summary

Conclusion

We have presented the first known, detailed account of kind inference for datatypes, both for Haskell98 and the Haskell of today.

Haskell and GHC

① Goal: produce a sound and complete pair of specification and algorithm.

Haskell and GHC

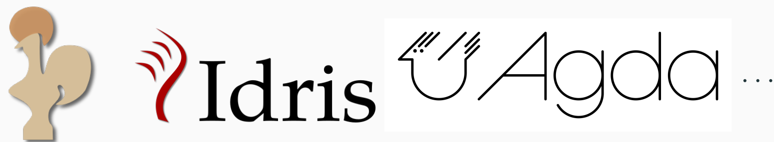
② Detailed comparison with GHC included in our technical supplement (Xie et al., 2019).

③ Identified ways GHC can improve.

③ Identified ways GHC can improve.

unexpected semantics $\Longrightarrow$ quantification check

Beyond Haskell



Inform the design of principled inference algorithms for languages beyond Haskell.

BRYN MAWR
COLLEGE

tweag.io

Kind Inference for Datatypes

Ningning Xie [1] Richard A. Eisenberg [2,3] Bruno C. d. S. Oliveira [1]

POPL 2020

[1]The University of Hong Kong

[2]Bryn Mawr College

[3]Tweag I/O

References

Claudio Sacerdoti Coen. 2004. *Mathematical knowledge management and interactive theorem proving*. Ph.D. Dissertation. University of Bologna, 2004. Technical Report UBLCS 2004-5.

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP'13*. 14.

Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *ESOP'16*.

Ronald Garcia and Matteo Cimini. 2015. Principal type schemes for gradual programs. In *POPL'15*. 13.

Adam Gundry, Conor McBride, and James McKinna. 2010. Type inference in context. In *MSFP*.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *JFP* 21, 4-5 (2011), 333–412.

Ningning Xie, Richard A. Eisenberg, and Bruno C. d. S. Oliveira. 2019. Kind Inference for Datatypes: Technical Supplement. (Nov. 2019). arXiv:1911.06153.

Beta Ziliani and Matthieu Sozeau. 2015. A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading. In *ICFP'15*.