How does Haskell Infer Types?

OutsideIn(X): Modular Type Inference with Local Assumptions

S.P. Jones et al. 2011

August 27, 2024

Non-HM Features of Haskell

Constrained polymorphism:

```
member :: ∀a. Eq a => a -> [a] -> Bool
```

Indexed types:

```
data R (a :: *) where
  RInt  :: Int  -> R Int
  RBool :: Bool -> R Bool
```

Type families:

```
type family F :: * -> *
type instance F [a]  = F a
type instance F Bool = Int
```

Type Classes are Ambiguous

```
show :: forall a. Show a => a -> String
read :: forall a. Show a => String -> a

flop :: String -> String
flop s = show (read s)
```

- There's clearly a `Show` α constraint of some kind
- but there's no way to pick α!
- We **must** reject such programs.

Type Families Mess With Type Equality

```
type family Elem c :: *
class HasElem c where
  first :: c -> Elem c

type instance Elem [a] = a
instance HasElem [a] where
  first = head

type instance Elem (a,b) = a
instance HasElem (a,b) where
  first = fst

exf :: (Elem a ~ Elem b, Eq a) => a -> b -> Bool
exf x y = first x == first y

-- exf [1] (1, False) is safe!
```

GADTs

GADTs in Haskell can

- have *local assumptions*
- be *existential*
- be *indexed*

```
data Ex a where
  ExEq :: forall b. Eq b  => b -> Ex Bool
  ExPl :: forall b. Num b => b -> Ex Int

ex :: Ex a -> a
ex (ExEq x) = x == x   -- returns a Bool
ex (ExPl y) = y + 1    -- returns an Int
```

Indexing is a Local Assumption

Natural representation of indexing via local assumption:

```
data R a where
  RInt  :: Int  -> R Int
  RBool :: Bool -> R Bool
```

```
data R a where
  RInt  :: (a ∼ Int)  => a -> R a
  RBool :: (a ∼ Bool) => a -> R a
```

GADTs and Inconsistency

Should we accept this program?

```haskell
data R a where
  R1 :: (a ~ Int)  => a -> R a
  R2 :: (a ~ Bool) => a -> R a

foo :: R Int -> Int
foo (R1 y) = y
foo (R2 y) = False
```

Problems

Type Classes: tractable. [See HM(X)]

Type Families: type equality is *non-structural*!

GADTs: *local assumptions* – constraints are scoped

Many examples will follow!

Indexing Loses Principal Types

```
data T a where
  T1 :: Int -> T Bool
  T2 :: T a

test (T1 n) _ = n > 0
test T2     r = r
```

Indexing Loses Principal Types

```
data T a where
  T1 :: Int -> T Bool
  T2 :: T a

test (T1 n) _ = n > 0
test T2     r = r
```

$$\texttt{test :: } \forall \texttt{a. T a -> Bool -> Bool}$$

or

$$\texttt{test :: } \forall \texttt{a. T a -> a -> a}$$

Indexing Loses Principal Types

```
data T a where
  T1 :: Int -> T Bool
  T2 :: T a

test (T1 n) _ = n > 0
test T2     r = r
```

test :: ∀a. T a -> Bool -> Bool

or

test :: ∀a. T a -> a -> a

```
test2 (T1 n) _ = n > 0
test2 T2     r = not r
```

Classes + Annotations Lose Prinicipality

```
class Foo a b where foo :: a -> b -> Int
instance Foo Int b
instance Foo a b => Foo [a] b

g y = let h :: forall c. c -> Int
          h x = foo y x
      in h True
```

g can have any of these types, but has no prinicpal type:

- `g :: Int -> Int`
- `g :: [Int] -> Int`
- `g :: [[Int]] -> Int`
- `g :: ...`

Existential data types can replace `h` in the example.

TCs + Constrained Data also a Problem

```
class C a
class B a b where op :: a -> b
instance C a => B a [a]

data R a where
  MkR :: C a => a -> T a    -- not indexed

-- k :: ∀ab. B a b => R a -> b
-- k :: ∀a .          R a -> [a]
k (MkR x) = op x
```

Failed Attempt

Quantification over constraints can fix the type system to restore principal types:

- `test :: ∀ab. (a ∼ Bool ⊃ b ∼ Bool) => T a -> b -> b`
- `g :: ∀b . (∀c. Foo b c) => b -> Int`
- `k :: ∀ab. (C a ⊃ B a b) => R a -> b`

But this is undesirable:

- Type checking becomes undecidable
- We really did want to reject those programs!

Our Challenge

Determine which definitions should be accepted, *independently of the constraint domain.*

Natural Type System with Local Assumptions (Syntax)

Term variables:	x, y, z, f, g, h
Type variables:	a, b, c
Unification variables:	$\alpha, \beta, \gamma, \ldots$
Data constructors:	K
Expressions:	e
Monotypes:	$\tau, \upsilon ::= a \mid \texttt{Int} \mid \texttt{T}\ \overline{\tau} \mid \texttt{F}\ \overline{\tau} \mid \ldots$
Constraints:	$Q ::= \varepsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid Q(X)$
Type schemes:	$\sigma ::= \forall \overline{a}.Q \Rightarrow \tau$
Top-level axioms:	$\mathcal{Q} ::= \varepsilon \mid \mathcal{Q} \wedge \mathcal{Q} \mid \forall \overline{a}.Q \Rightarrow Q$
Generated constraints:	$C ::= Q \mid C_1 \wedge C_2 \mid \exists \overline{\alpha}.(Q \supset C)$

Note: Data Constructor Types

Γ_0 initially contains data constructor types. Shape:

$$K : \forall \overline{a}\overline{b}.Q \Rightarrow \overline{v} \rightarrow \mathtt{T}\ \overline{a}$$

Note $\overline{b}$ (existentials) and the local assumptions!

Natural Type System with Local Assumptions

$$\frac{(\nu : \forall \overline{a}.Q_1 \Rightarrow \upsilon) \in \Gamma \qquad Q \Vdash [\overline{a \mapsto \tau}]Q_1}{Q; \Gamma \vdash \nu : [\overline{a \mapsto \tau}]\upsilon} \text{ VarCon}$$

$$\frac{Q; \Gamma \vdash e : \tau_1 \qquad Q \Vdash \tau_1 \sim \tau_2}{Q; \Gamma \vdash e : \tau_2} \text{ Eq}$$

$$\frac{Q; \Gamma \vdash e_1 : \tau_1 \qquad Q; \Gamma, (x : \tau_1) \vdash e_2 : \tau_2}{Q; \Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2} \text{ Let}$$

$$\frac{\begin{array}{c} Q \wedge Q_1; \Gamma \vdash e_1 : \tau_1 \qquad \overline{a} \# ftv(Q, \Gamma) \\ Q; \Gamma, (x : \forall \overline{a}.Q_1 \Rightarrow \tau_1) \vdash e_2 : \tau_2 \end{array}}{Q; \Gamma \vdash \texttt{let } x :: \forall \overline{a}.Q_1 \Rightarrow \tau_1 = e_1 \texttt{ in } e_2 : \tau_2} \text{ LetA}$$

Natural System (Case)

$$\frac{Q;\Gamma \vdash e : \mathtt{T}\ \overline{\tau} \qquad K_i : \forall \overline{a}\overline{b}.Q_i \Rightarrow \overline{\upsilon_i} \rightarrow \mathtt{T}\ \overline{a} \in \Gamma \qquad ftv(Q, \Gamma, \overline{\tau}, \tau_r)\#\overline{b} \qquad \overline{Q \wedge ([\overline{a \mapsto \tau}]Q_i); \Gamma, (\overline{x_i : [\overline{a \mapsto \tau}]\upsilon_i}) \vdash e_i : \tau_r}}{Q;\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ \{\overline{K_i\ \overline{x}_i \rightarrow e_i}\} : \tau_r}\ \text{Case}$$

Natural is Too Permissive

Can't check principality. But it gets worse!

Natural is Too Permissive

Can't check principality. But it gets worse!
Recall:

```
data T a where
  T1 :: Int -> T Bool
  T2 :: T a
```

Consider:

```
fr :: a -> T a -> Bool
fr x y = let gr z = not x
         in case y of
              T1 _ -> gr ()
              T2   -> True
```

Type safe?

Natural is Too Permissive

Can't check principality. But it gets worse!
Recall:

```
data T a where
  T1 :: Int -> T Bool
  T2 :: T a
```

Consider:

```
fr :: a -> T a -> Bool
fr x y = let gr z = not x
         in case y of
           T1 _ -> gr ()
           T2   -> True
```

Type safe?
Yes – but we should reject it! Problem: `let` generalization.

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \rightarrow \texttt{T}\ a$$

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \rightarrow \texttt{T}\ a$$

1. make up α for whole term, β_x for x

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \rightarrow \texttt{T}\ a$$

1. make up α for whole term, β_x for x
2. Learn $\beta_x \sim \texttt{T}\ \gamma$ for some γ

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \to \texttt{T}\ a$$

1. make up α for whole term, β_x for x
2. Learn $\beta_x \sim \texttt{T}\ \gamma$ for some γ
3. In branch (where $\gamma \sim \texttt{Bool}$), $\alpha \sim \texttt{Bool}$

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \to \texttt{T}\ a$$

1. make up α for whole term, β_x for x
2. Learn $\beta_x \sim \texttt{T}\ \gamma$ for some γ
3. In branch (where $\gamma \sim \texttt{Bool}$), $\alpha \sim \texttt{Bool}$
4. emit $(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool})$ as constraint

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a. (\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \rightarrow \texttt{T}\ a$$

1. make up α for whole term, β_x for x
2. Learn $\beta_x \sim \texttt{T}\ \gamma$ for some γ
3. In branch (where $\gamma \sim \texttt{Bool}$), $\alpha \sim \texttt{Bool}$
4. emit $(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool})$ as constraint
5. Solve it?

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \to \texttt{T}\ a$$

1. make up α for whole term, β_x for x
2. Learn $\beta_x \sim \texttt{T}\ \gamma$ for some γ
3. In branch (where $\gamma \sim \texttt{Bool}$), $\alpha \sim \texttt{Bool}$
4. emit $(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool})$ as constraint
5. Solve it?

Type Inference with GADTs, Informally

Consider `\x -> case x of { T1 n -> n > 0 }`, where

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \to \texttt{T}\ a$$

1. make up α for whole term, β_x for x
2. Learn $\beta_x \sim \texttt{T}\ \gamma$ for some γ
3. In branch (where $\gamma \sim \texttt{Bool}$), $\alpha \sim \texttt{Bool}$
4. emit $(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool})$ as constraint
5. Solve it?

Can't solve it – two solutions!

- $[\alpha \mapsto \texttt{Bool}]$
- $[\alpha \mapsto \gamma]$

Type Inference with GADTs, Informally

What about `\x -> case x of { T1 n -> n > 0; T2 -> True }`?
Constraints will be:

$$(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool}) \wedge (\alpha \sim \texttt{Bool})$$

This one is solvable.

Type Inference with GADTs, Informally

What about `\x -> case x of { T1 n -> n > 0; T2 -> True }`?
Constraints will be:

$$(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool}) \wedge (\alpha \sim \texttt{Bool})$$

This one is solvable.

Idea: Treat global tyvars as skolem under implications.

(constraints solved separately!)

Local type variables

What about local tyvars under implications?
Let

$$\texttt{T3} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{[Int]} \rightarrow \texttt{T}\ a$$

$$\texttt{null} : \forall d.[d] \rightarrow \texttt{Bool}$$

and consider `\x -> case x of { T3 n -> null n }`

Local type variables

What about local tyvars under implications?
Let

$$\texttt{T3} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{[Int]} \to \texttt{T}\ a$$

$$\texttt{null} : \forall d.[d] \to \texttt{Bool}$$

and consider `\x -> case x of { T3 n -> null n }`
We conclude

$$\gamma \sim \texttt{Bool} \supset (\alpha \sim \texttt{Bool} \wedge \delta \sim \texttt{Int})$$

But δ is entirely local, so no risk to unify $[\delta \mapsto \texttt{Int}]$!

$$\exists \delta . \gamma \sim \texttt{Bool} \supset (\alpha \sim \texttt{Bool} \wedge \delta \sim \texttt{Int})$$

We say δ is "touchable."

Seems Conservative, Actually Robust

Conservative:

- Approach can't solve $(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Int})$.
- But the solution $[\alpha \mapsto \texttt{Int}]$ is unique!

Seems Conservative, Actually Robust

Conservative:

- Approach can't solve ($\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Int}$).
- But the solution $[\alpha \mapsto \texttt{Int}]$ is unique!

Constraints are all solved in context of $\mathcal{Q}$.

- What if $F\ \texttt{Bool} \sim \texttt{Int} \in \mathcal{Q}$?
- New solution: $[\alpha \mapsto F\ \gamma]$
- So we really should not solve above example.

Seems Conservative, Actually Robust

Conservative:

- Approach can't solve $(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Int})$.
- But the solution $[\alpha \mapsto \texttt{Int}]$ is unique!

Constraints are all solved in context of $\mathcal{Q}$.

- What if $F\ \texttt{Bool} \sim \texttt{Int} \in \mathcal{Q}$?
- New solution: $[\alpha \mapsto F\ \gamma]$
- So we really should not solve above example.

Really is conservative, though:

- Cannot solve $(\varepsilon \supset \alpha \sim \texttt{Int})$...
- but even in an open world there can only be one solution.

Constraint Generation (Let)

$$\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}$$

$$\ldots$$

Constraint Generation (Let)

$$\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}$$

$$\dots$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma, (x : \tau_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \text{Let}$$

Constraint Generation (Let)

$$\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}$$

$$\ldots$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma, (x : \tau_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \text{Let}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau \rightsquigarrow C_1 \qquad \Gamma, (x : \tau) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x :: \tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2 \wedge \tau \sim \tau_1}\ \text{LetA}$$

Constraint Generation (Let)

$$\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}$$

...

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma, (x : \tau_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \text{Let}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau \rightsquigarrow C_1 \qquad \Gamma, (x : \tau) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x :: \tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2 \wedge \tau \sim \tau_1}\ \text{LetA}$$

$$\frac{\begin{array}{c} \sigma_1 = \forall \overline{a}. Q_1 \Rightarrow \tau_1 \qquad Q_1 \neq \varepsilon \text{ or } \overline{a} \neq \varepsilon \\ \Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau \rightsquigarrow C \qquad \overline{\beta} = fuv(\tau, C) - fuv(\Gamma) \\ C_1 = \exists \overline{\beta}.(Q_1 \supset C \wedge \tau \sim \tau_1) \\ \Gamma, (x : \sigma_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2 \end{array}}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x :: \sigma_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \text{GLetA}$$

Constraint Generation (Case)

$$\frac{\begin{array}{c}\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C \qquad \beta, \overline{\gamma} \text{ fresh} \\ K_i : \forall \overline{a}\overline{b}_i . Q_i \Rightarrow \overline{\upsilon_i} \to \mathtt{T}\ \overline{a} \qquad \overline{b}_i \text{ fresh} \\ \Gamma, (\overline{x_i : [\overline{a \mapsto \gamma}]\upsilon_i}) \mathrel{|\!\!\blacktriangleright} e_i : \tau_i \rightsquigarrow C_i \\ \overline{\delta}_i = fuv(\tau_i, C_i) - fuv(\Gamma, \overline{\gamma}) \\ C_i' = \begin{cases} C_i \wedge \tau_i \sim \beta & \text{if } \overline{b}_i = \varepsilon \text{ and } Q_i = \varepsilon \\ \exists \overline{\delta}_i . ([\overline{a \mapsto \gamma}]Q_i \supset C_i \wedge \tau_i \sim \beta) & \text{otherwise} \end{cases}\end{array}}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathtt{case}\ e\ \mathtt{of}\ \{\overline{K_i\ \overline{x}_i \to e_i}\} : \beta \rightsquigarrow C \wedge (\mathtt{T}\ \overline{\gamma} \sim \tau) \wedge (\bigwedge C_i')}\ \textsc{Case}$$

Solver Signature

$$\boxed{\mathcal{Q}; Q_{given}; \overline{a}_{tch} \vdash\!\blacktriangleright^{solv} C_{wanted} \rightsquigarrow Q_{residual}; \theta}$$

$\mathcal{Q}$	global axioms
Q_{given}	given constraints (no $\exists$)
$\overline{a}_{tch}$	touchable variables
C_{wanted}	constraints to solve
$Q_{residual}$	constraints failed to solve
θ	substitution witness ($dom(\theta) \subseteq \overline{\alpha}_{tch}$)

Top-level Inference Rules

$$\boxed{\mathcal{Q}; \Gamma \mathrel{|\!\!\blacktriangleright} prog}$$

$$\frac{\begin{array}{c}\Gamma \mathrel{|\!\!\blacktriangleright} e : \upsilon \rightsquigarrow C \\ \mathcal{Q}; Q; fuv(\upsilon, C) \mathrel{|\!\!\blacktriangleright}^{solv} C \wedge \upsilon \sim \tau \rightsquigarrow \varepsilon; \theta \\ \mathcal{Q}; \Gamma, (f : \forall \overline{a}.Q \Rightarrow \tau) \mathrel{|\!\!\blacktriangleright} prog\end{array}}{\mathcal{Q}; \Gamma \mathrel{|\!\!\blacktriangleright} f :: (\forall \overline{a}.Q \Rightarrow \tau) = e, prog} \textsc{BindA}$$

Top-level Inference Rules

$$\boxed{\mathcal{Q}; \Gamma \mapsto prog}$$

$$\frac{\begin{array}{c}\Gamma \mapsto e : \upsilon \rightsquigarrow C \\ \mathcal{Q}; Q; fuv(\upsilon, C) \mapsto^{solv} C \wedge \upsilon \sim \tau \rightsquigarrow \varepsilon; \theta \\ \mathcal{Q}; \Gamma, (f : \forall \overline{a}.Q \Rightarrow \tau) \mapsto prog\end{array}}{\mathcal{Q}; \Gamma \mapsto f :: (\forall \overline{a}.Q \Rightarrow \tau) = e, prog} \textsc{BindA}$$

$$\frac{\begin{array}{c}\Gamma \mapsto e : \tau \rightsquigarrow C \qquad \mathcal{Q}; \varepsilon; fuv(\tau, C) \mapsto^{solv} C \rightsquigarrow Q; \theta \\ \overline{a} \text{ fresh} \qquad \overline{\alpha} = fuv(\theta\tau, Q) \\ \mathcal{Q}; \Gamma, (f : \forall \overline{a}.[\overline{\alpha \mapsto a}](Q \Rightarrow \theta\tau)) \mapsto prog\end{array}}{\mathcal{Q}; \Gamma \mapsto f = e, prog} \textsc{Bind}$$

Parameterized Constraint Solver

$$\boxed{\mathcal{Q}; Q_{given}; \overline{a}_{tch} \mapsto^{simp} \text{Q}_{wanted} \rightsquigarrow Q_{residual}; \theta}$$

(constraint solver for X)

$$\boxed{\mathcal{Q}; Q_{given}; \overline{a}_{tch} \mapsto^{solv} C_{wanted} \rightsquigarrow Q_{residual}; \theta}$$

$$\frac{\begin{array}{c}\mathcal{Q}; Q_g; \overline{\alpha} \mapsto^{simp} \textbf{simple}[C] \rightsquigarrow Q_r; \theta \\ \forall (\exists \overline{\alpha}_i.(Q_i \supset C_i) \in \textbf{implic}[\theta C]), \\ \mathcal{Q}; Q_g \wedge Q_r \wedge Q_i; \overline{\alpha}_i \mapsto^{solv} C_i \rightsquigarrow \varepsilon; \theta_i\end{array}}{\mathcal{Q}; Q_g; \overline{\alpha} \mapsto^{solv} C \rightsquigarrow Q_r; \theta} \text{ SOLVE}$$

Aside: Skolem Escape Checks

Should this typecheck?

```
data Ex where
  Ex :: forall b. b -> Ex

f = case (Ex 3) of Ex _ -> False
```

Aside: Skolem Escape Checks

Should this typecheck?

```
data Ex where
  Ex :: forall b. b -> Ex

f = case (Ex 3) of Ex _ -> False
```

- Only constraint will be $\varepsilon \supset \alpha \sim \texttt{Bool}$ ...
- but we can't solve it, because α is skolem under implication
- Since LHS doesn't entail equalities, we could float RHS ...
- as long as we make sure that `b` doesn't escape.

What have we Achieved? (stopping point)

- Seen why Haskell type inference is hard.
- Established "natural" type system – accepts too many programs.
- Seen *implication constraints* and *touchable variables*
- Observed *OutsideIn(X)* is incomplete.
- Studied constraint generation + algorithm for discharging implications.

Still to do:

- Solve *non*-implication constraints!
- "evidence": solving constraints effects object code, so witnesses of solution are required.
- Interaction with other type system features.

Syntactic Extensions

$$
\begin{aligned}
\tau \quad &::= \ldots \mid F\ \overline{\tau} \\
Q \quad &::= \ldots \mid \mathtt{D}\ \overline{\tau} \\
\mathcal{Q} \quad &::= Q \mid \mathcal{Q} \wedge \mathcal{Q} \mid \forall \overline{a}.Q \Rightarrow \mathtt{D}\ \overline{\tau} \mid \forall \overline{a}.F\ \overline{\xi} \sim \tau
\end{aligned}
$$

$$\zeta, \xi \in \{\tau \mid \tau \text{ contains no type families}\}$$

$$
\begin{aligned}
\mathbb{T} \quad &::= \mathtt{T}\ \overline{\mathbb{T}} \mid F\ \overline{\mathbb{T}} \mid \mathbb{T} \rightarrow \mathbb{T} \mid tv \mid \bullet \\
\mathbb{F} \quad &::= F\ \overline{\mathbb{T}} \\
\mathbb{D} \quad &::= \mathtt{D}\ \overline{\mathbb{T}}
\end{aligned}
$$

Solving Equality Constraints is Tricky!

Consider:

```
type instance F [Int] = Int
type instance G [a]   = Bool

-- Assume g :: forall b. b -> G b

f :: forall a. (a ~ [F a]) => a -> Bool
f x = g x
```

Solving Equality Constraints is Tricky!

Consider:

```
type instance F [Int] = Int
type instance G [a]   = Bool

-- Assume g :: forall b. b -> G b

f :: forall a. (a ~ [F a]) => a -> Bool
f x = g x
```

1. $G\ a \sim \texttt{Bool}$
2. $a \sim [F\ a]$ given $\implies G\ [F\ a] \sim \texttt{Bool}$
3. Discharge with axiom for G.

But step 2 could repeated forever!
Even worse: how to use givens like $F\ (G\ a) \sim G\ a$?

Our Goal

- See importance of order insensitivity
- *intuitively* understand the simplifier

Full details of the solver are the subject of nearly half the paper. I'd encourage you to read it if you're interested, but it's way more than I want to cover here.

What are we simplifying?

Recall the signature:

$$\boxed{\mathcal{Q}; Q_{given}; \overline{a}_{tch} \mapsto^{simp} Q_{wanted} \rightsquigarrow Q_{residual}; \theta}$$

- Form "state quadruple" $\langle \overline{\alpha}, \varphi, Q_g, Q_w \rangle$.
- Apply rules to rewrite this to equivalent tuples.
- Attempt to discharge non-subst equalities in Q_w.
- Any remaining non-subst equalities in φQ_w will be Q_r.
- Form θ from remaining subst equalities in φQ_w.

What are we simplifying?

Recall the signature:

$$\boxed{\mathcal{Q}; Q_{given}; \overline{a}_{tch} \mapsto^{simp} Q_{wanted} \rightsquigarrow Q_{residual}; \theta}$$

- Form "state quadruple" $\langle \overline{\alpha}, \varphi, Q_g, Q_w \rangle$.
- Apply rules to rewrite this to equivalent tuples.
- Attempt to discharge non-subst equalities in Q_w.
- Any remaining non-subst equalities in φQ_w will be Q_r.
- Form θ from remaining subst equalities in φQ_w.

Why is Q_g in the state? Consider

$$F\ \overline{\xi} \sim \zeta_1 \wedge F\ \overline{\xi} \sim \zeta_2$$

Rewriting Q_g is only way to extract $\zeta_1 \sim \zeta_2$. Other reasons too.

Canonical Form (CF)

Core concept, especially for reasoning about termination!
Primarily concerned with eliminating type families.

- $tv \sim \xi$
- $F\ \overline{\xi} \sim \zeta$
- $\texttt{D}\ \overline{\xi}$

Property: type family may **only** be at head.
Property: type family may **only** be on the left.

Types of Rewrites

1. Canonicalize: rewrite g or w constraint towards CF (16)
2. Interact: simplify a g or w constraint using another (12)
3. Simplify: rewrite w constraint using g constraint (6)
4. Top-level reaction: rewrite w constraint using axiom (4)

Order-insensitivity means any of the *38* rules can be applied at any time without changing the final solution.

Canonicalization

$$\boxed{canon[\ell]\ (\ Q_1\ ) = \{\overline{\beta}, \varphi, Q_2\}_\perp}$$

Contains all rules from original HM system except Elim:

- $canon[\ell](\tau \sim \tau) = \{\varepsilon, \varepsilon, \varepsilon\}$
- $canon[\ell](\mathtt{T}\ \overline{\tau_1} \sim \mathtt{T}\ \overline{\tau_2}) = \{\varepsilon, \varepsilon, \bigwedge \overline{\tau_1 \sim \tau_2}\}$
- An orient rule (exact ordering not important)
- Two error cases (mismatched constructor, occurs check)

Canonicalization

$$\boxed{canon[\ell]\ (\ Q_1\ ) = \{\overline{\beta}, \varphi, Q_2\}_\perp}$$

Contains all rules from original HM system except Elim:

- $canon[\ell](\tau \sim \tau) = \{\varepsilon, \varepsilon, \varepsilon\}$
- $canon[\ell](\mathtt{T}\ \overline{\tau}_1 \sim \mathtt{T}\ \overline{\tau_2}) = \{\varepsilon, \varepsilon, \bigwedge \overline{\tau_1 \sim \tau_2}\}$
- An orient rule (exact ordering not important)
- Two error cases (mismatched constructor, occurs check)

Also contains 6 "flattening" rules, such as

$$canon[w](\mathbb{F}[G\ \overline{\xi}] \sim \tau) = \{\beta, \varepsilon, \mathbb{F}[\beta] \sim \tau \wedge G\ \overline{\xi} \sim \beta\}$$

One per $\mathbb{T}, \mathbb{F}, \mathbb{D}$ and per g, w.
In the g case, we must *justify* the flattening by adding it to φ.

Binary Interaction

Transform two **canonical** g or w constraints to simpler pair.
Example: $tv \sim \xi_1 \wedge tv \sim \xi_2$ are not "subst equalities" because the LHS is the same type variable. Rewrite to

$$tv \sim \xi_1 \wedge \xi_1 \sim \xi_2$$

Now they are subst equalities (in suitable Q).

Binary Interaction

Transform two **canonical** g or w constraints to simpler pair.
Example: $tv \sim \xi_1 \wedge tv \sim \xi_2$ are not "subst equalities" because the LHS is the same type variable. Rewrite to

$$tv \sim \xi_1 \wedge \xi_1 \sim \xi_2$$

Now they are subst equalities (in suitable Q).
Other rules:

- use $tv \sim \xi$ to rewrite tv in second constraint (3)
- Pair of equalities for $F\ \overline{\xi}$ (seen before)
- Delete one of $\mathtt{D}\ \overline{\xi} \wedge \mathtt{D}\ \overline{\xi}$

Simplification

Like the binary interaction rules, except directional.

Two key differences:

1. Simplifying wanted $tv \sim \xi$. What could be simpler?
   - If given is also $tv' \sim \zeta$, can apply it as subst ...
   - But what if given is type family equality or dict constraint?
   - Should not create new "flattening wanted" – more work!
   - **Must not** create new givens – no justification!
2. Pair of matching type class constraints discharges the wanted one.

Top-level Reactions

Straightforward: apply matching type family / instance axioms. One catch: if reacting given type class constraint, and have matching instance axiom, this should be an *error*. Consider:

```
class D a where
  d :: a -> Bool
instance C a => D [a] where ...

f :: forall a. D [a] => [a] -> Bool
f x = d x
```

Discharge `d` with local evidence or global?

Top-level Reactions

Straightforward: apply matching type family / instance axioms. One catch: if reacting given type class constraint, and have matching instance axiom, this should be an *error*. Consider:

```
class D a where
  d :: a -> Bool
instance C a => D [a] where ...

f :: forall a. D [a] => [a] -> Bool
f x = d x
```

Discharge `d` with local evidence or global?

- Simplest answer: don't answer!
- GHC answer: very complicated instance resolution
  - Applied at simplifying class constraints + here
  - >12 pages of docs to describe

Conclusion

Additional achievement: solve non-implication constraints!

Takeaways:

- Type inference is hard
- Complete type inference is (literally) impossible
  - Type family can encode addition, lead to looping
  - Settle for knowing algorithm is sound and principal
- Consider not generalizing `let` in your systems
- Solving non-structural equalities is hard
- Carefully crafted rewrite systems can do a lot of work!