Microsoft Research

PENN PL

Visible Type Application

Richard A. Eisenberg Stephanie Weirich
Hamidhasan Ahmed

University of Pennsylvania
Philadelphia, PA, USA

European Symposium on Programming
Eindhoven, Netherlands
4 April 2016

The Problem

```
id :: ∀ a. a → a
id x = x
```

What if I want to specialize `id` to `Int`?

Type signature: `id :: Int → Int`

Visible type application: `id @Int`

And it gets worse…

Without type application

```
bind = coerce ((>>=)
  :: Identity a
  → (a -> Identity b)
  → Identity b)
```

```
compile (SCond (se0 :: Sing e0)
  (se1 :: Sing e1) (se2 :: Sing e2))
  = case (fact ’(sEval se0)
              (P :: P (Eval e1))
              (P :: P (Eval e2)) ::
        ((If (Eval e0) (Eval e1)
              (Eval e2)) ’: s) :~:
         (If (Eval e0)
              ((Eval e1) ’: s)
              ((Eval e2) ’: s))) of
  Refl → compile se0 ++
    IFPOP (compile se1)
          (compile se2) ::: Nil
```

With type application

```
bind = coerce
  ((>>=) @Identity @a @b)
```

```
compile (SCond se0
  (se1 :: Sing e1) (se2 :: Sing e2))
  = case fact @(Eval e1) @(Eval e2)
              @s (sEval se0) of
  Refl → compile se0 ++
    IFPOP (compile se1)
          (compile se2) ::: Nil
```

[McBride’s 2012 ICFP keynote]

But, surely this is easy!

Other languages have this.
(Coq, Agda, Idris, Java, F#, ...)

no formal analysis

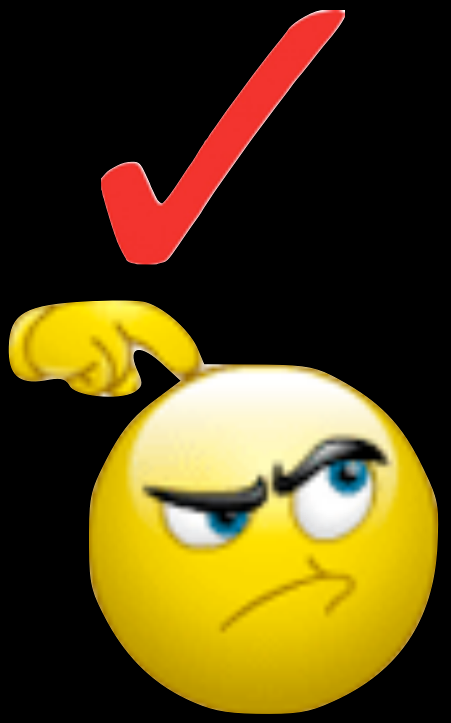
Naive approach:
Allow type application
only after function names.

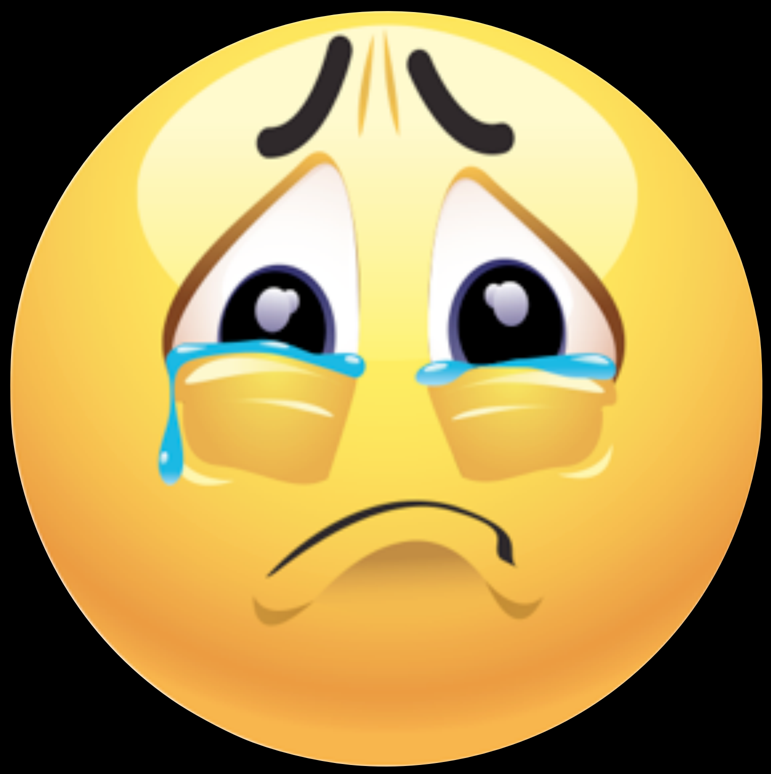
Types after function names

```
id @Int

3 @Int

pair :: ∀ a. a
     → ∀ b. b
     → (a, b)
pair @Int 4
     @Bool True
```



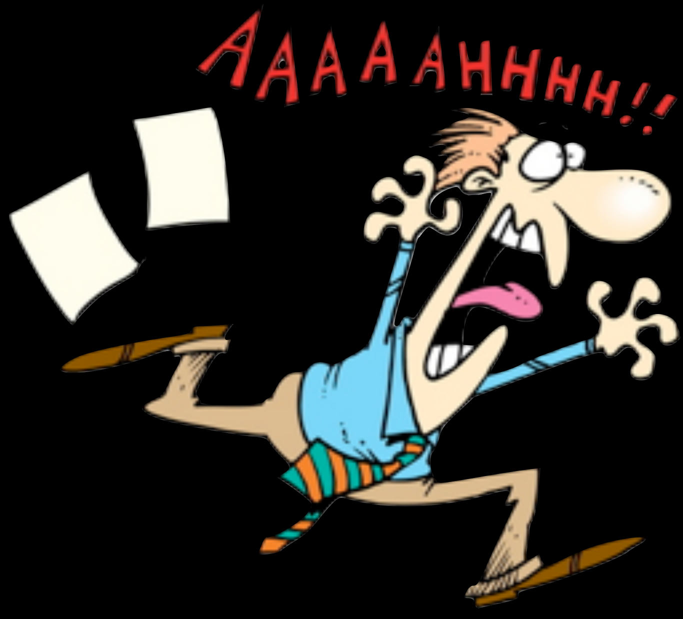


Problems inferring types

```
swap (a, b) = (b, a)
```

1) `∀ a b. (a, b) → (b, a)`
2) `∀ b a. (a, b) → (b, a)`

Choice of type matters with type application



Hindley-Milner to the rescue

A declarative system,
independent of inference.

Declarative systems are
easier to reason about.

Hindley-Milner to the rescue

Principal types property:
Every expression has a
most general type.

Corollary: Let-expansion and
-contraction are reasonable.

Two key ideas

Specified vs. generalized type variables

Lazy instantiation

A type inference problem

```
const :: ∀ a b. a → b → a
id :: ∀ c. c → c

let x = const id
in x @Int
```

Inferred type:

`x :: b → (c → c)`

What is the type of `x @Int`?

1) `b → (Int → Int)`
2) `Int → (c → c)`

A type inference problem

```
const :: ∀ a b. a → b → a
id :: ∀ c. c → c

let x :: ∀ b c. b → (c → c)
    x = const id
in x @Int
```

What is the type of `x @Int`?

1) `b → (Int → Int)`
2) `Int → (c → c)`

What just happened?

```
x :: ∀ b c. b → (c → c)
```

x's type is specified.

User-written types control the order of type instantiation.

What just happened?

```
x ::          b → (c → c)
```

x's type is specified.

User-written types control the order of type instantiation.

Types without ∀ use left-to-right ordering.

Specified vs. generalized

Specified variables are user-written.

(with or without ∀)

Generalized variables are invented by the compiler.

Visible type application works with only specified variables.

A type inference problem

```
const :: ∀ a b. a → b → a
id :: ∀ c. c → c

let x = const id
in x @Int
```

Inferred type:

`x :: b → (c → c)`

What is the type of `x @Int`?

1) `b → (Int → Int)`
2) `Int → (c → c)`
3) ill-typed

The Hindley-Milner Type System

Declarative system:

Easier to reason about

$\vdash$

Syntax-directed system:

Easier to implement

$\models$

The Hindley-Milner Type System

monotype $\tau ::= a \mid \tau_1 \rightarrow \tau_2 \mid \mathrm{Int}$

polytype $\sigma ::= \forall a.\sigma \mid \tau$

The Hindley-Milner Type System

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \text{Int}$$

$$\sigma ::= \forall a.\sigma \mid \tau$$

assigns a polytype

$$\frac{a \notin \mathit{ftv}(\Gamma) \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall a.\sigma}\ \text{GEN} \qquad \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

The Hindley-Milner Type System

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \text{Int}$$

$$\sigma ::= \forall a.\sigma \mid \tau$$

$$\frac{a \notin \mathit{ftv}(\Gamma) \quad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall a.\sigma}\ \text{GEN}$$

$$\frac{\Gamma \vdash e : \sigma_1 \quad \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

can happen anywhere, anytime

The Hindley-Milner Type System

$\tau ::= a \mid \tau_1 \to \tau_2 \mid \text{Int}$

$\sigma ::= \forall a.\sigma \mid \tau$

$$\frac{a \notin \mathit{ftv}(\Gamma) \quad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall a.\sigma}\ \text{GEN}$$

$$\frac{\Gamma \vdash e : \sigma_1 \quad \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

$\forall a.\ a \to a \qquad \sigma_1 \leq \sigma_2 \qquad \text{Int} \to \text{Int}$

"σ_1 is more general than σ_2"

The Hindley-Milner Type System

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \text{Int}$$

$$\sigma ::= \forall\{a\}.\sigma \mid \tau$$

generalized

$$\frac{a \notin \mathit{ftv}(\Gamma) \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\{a\}.\sigma}\ \text{GEN} \qquad \frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \le \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

$$\sigma_1 \le \sigma_2$$

"σ_1 is more general than σ_2"

System V

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \mathrm{Int}$$

$$\upsilon ::= \forall a.\upsilon \mid \tau$$

$$\sigma ::= \forall\{a\}.\sigma \mid \upsilon$$

generalized before specified

$$\frac{a \notin ftv(\Gamma) \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\{a\}.\sigma}\ \text{GEN}$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

$$\frac{\Gamma \vdash e : \forall a.\upsilon}{\Gamma \vdash e\,@\tau : \upsilon[\tau/a]}\ \text{TAPP}$$

System V

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \mathrm{Int}$$

$$\upsilon ::= \forall a.\upsilon \mid \tau$$

$$\sigma ::= \forall \{a\}.\sigma \mid \upsilon$$

$$\frac{a \notin \mathit{ftv}(\Gamma) \quad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\{a\}.\sigma}\ \text{GEN}$$

$$\frac{\Gamma \vdash e : \sigma_1 \quad \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

generalized

specified

$$\frac{\Gamma \vdash e : \forall a.\upsilon}{\Gamma \vdash e\,@\tau : \upsilon[\tau/a]}\ \text{TAPP}$$

Subsumption

$$\sigma_1 \le \sigma_2$$

"σ_1 is more general than σ_2"

$$\forall\{a,b\}.a \to b \le \forall\{a\}.a \to Int$$

$$\forall a,b.a \to b \le Int \to Int$$

$$\forall a,b.a \to b \le \forall a.a \to Int$$

$$\forall a,b.a \to b \not\le \forall b.Int \to b$$

System V

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \mathrm{Int}$$

$$\upsilon ::= \forall a.\upsilon \mid \tau$$

$$\sigma ::= \forall \{a\}.\sigma \mid \upsilon$$

$$\frac{a \notin \mathit{ftv}(\Gamma) \quad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \{a\}.\sigma}\ \text{GEN}$$

$$\frac{\Gamma \vdash e : \sigma_1 \quad \sigma_1 \leq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \text{SUB}$$

$$\frac{\Gamma \vdash e : \forall a.\upsilon}{\Gamma \vdash e\ @\tau : \upsilon[\tau/a]}\ \text{TAPP}$$

Two key ideas

Specified vs. generalized type variables ✓

How to implement?

Lazy instantiation

Syntax-directed version of HM

$$\frac{x{:}\forall\{\bar{a}\}.\tau \in \Gamma}{\Gamma \vDash x : \tau[\overline{\tau'/a}]}\ \text{VAR}$$

Variables are immediately fully instantiated.

Syntax-directed System V

$$\boxed{\Gamma \vDash e : \tau}$$

$$\boxed{\Gamma \vDash^{*} e : \upsilon}$$

$$\frac{\Gamma \vDash^{*} e : \forall \bar{a}.\tau}{\Gamma \vDash e : \tau[\overline{\tau'/a}]}\ \text{INST}$$

$$\frac{x{:}\forall\{\bar{a}\}.\upsilon \in \Gamma}{\Gamma \vDash^{*} x : \upsilon[\overline{\tau'/a}]}\ \text{VAR}$$

might still have specified variables

$$\frac{\Gamma \vDash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vDash e_2 : \tau_1}{\Gamma \vDash e_1\ e_2 : \tau_2}\ \text{APP}$$

Syntax-directed System V

$$\boxed{\Gamma \vDash e : \tau} \qquad \boxed{\Gamma \vDash^{*} e : \upsilon}$$

$$\frac{\Gamma \vDash^{*} e : \forall \bar{a}.\tau}{\Gamma \vDash e : \tau[\overline{\tau'/a}]}\ \text{INST} \qquad \frac{x{:}\forall\{\bar{a}\}.\upsilon \in \Gamma}{\Gamma \vDash^{*} x : \upsilon[\overline{\tau'/a}]}\ \text{VAR}$$

might still have specified variables

$$\frac{\Gamma \vDash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vDash e_2 : \tau_1}{\Gamma \vDash e_1\ e_2 : \tau_2}\ \text{APP}$$

Specified variable instantiation happens only when necessary.

Two key ideas

Specified vs. generalized type variables ✓

Lazy instantiation ✓

Analysis

Theorem: System V has principal types.

Theorem: System V is a conservative extension of Hindley-Milner.

Analysis

specified variables

lazy instantiation

Approach is general and extensible.

Proof: extension to a bidirectional system with higher-rank types.

Related Work

Peyton Jones, Vytiniotis, Weirich, Shields. *Practical type inference for arbitrary-rank types*. JFP, 2007.

Dunfield, Krishnaswami. *Complete and easy bidirectional typechecking for higher-rank polymorphism*. ICFP, 2013.

Summary

Design for visible
type application that:

- retains principal types
- extends Hindley-Milner
- is implemented for Haskell in GHC 8.0

Visible Type Application

Richard A. Eisenberg Stephanie Weirich
Hamidhasan Ahmed

University of Pennsylvania
Philadelphia, PA, USA

European Symposium on Programming
Eindhoven, Netherlands
4 April 2016