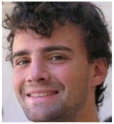
HIW'14 • September 6th 2014

# Partial Type Signatures

Thomas Winant



Dominique Devriese

Frank Piessens

Tom Schrijvers

KU LEUVEN

UNIVERSITEIT GENT

# Partial Type Signature

# Partial Type Signature

$foo\ file = \mathbf{do}\ \ldots\ ?\ \ldots$

# Partial Type Signature

$$foo\ file = \mathbf{do}\ \ldots\ \_\ \ldots$$

# Partial Type Signature

$$foo\ file = \mathbf{do}\ \ldots\ \_\ \ldots$$

```
Found hole ‘_’ with type: …
    Relevant bindings include
      …
```

# Partial Type Signature

$$\textit{foo file} = \textbf{do} \ldots$$

```
Found hole '_' with type: …
    Relevant bindings include
      …
```

# Partial Type Signature

$$foo :: FilePath \rightarrow IO \, ?$$
$$foo \; file = \mathbf{do} \; \ldots$$

```
Found hole ‘_’ with type: …
    Relevant bindings include
      …
```

# Partial Type Signature

$$foo :: FilePath \to IO\ \_$$
$$foo\ file = \mathbf{do}\ \ldots$$

```
Found hole ‘_’ with type: …
    Relevant bindings include
      …
```

# Partial Type Signature

$$foo :: FilePath \to IO\ \_$$
$$foo\ file = \mathbf{do}\ \ldots$$

```
Found hole ‘_’ with type: …
    In the type signature: foo :: FilePath -> IO _
    To use the inferred type,
      enable PartialTypeSignatures
```

# Overview

Motivation

Syntax

Formalisation

Implementation

# Motivation

# Motivation

Dilemma: write the complete type signature or none at all?

# Motivation

Dilemma: write the complete type signature or none at all?

Compromise: partial type signatures

# Motivation

Dilemma: write the complete type signature or none at all?

Compromise: partial type signatures

$\Rightarrow$ Mix annotated with inferred types using wildcards ($\_$).

$$\mathit{foo} :: \_ \rightarrow (\_, \mathit{Bool}) \quad \text{-- Inferred: } \mathit{Bool} \rightarrow (\mathit{Bool}, \mathit{Bool})$$
$$\mathit{foo}\ x = (x, x)$$

# Motivation

Dilemma: write the complete type signature or none at all?

Compromise: partial type signatures

⇒ Mix annotated with inferred types using wildcards (_).

$$foo :: \_ \rightarrow (\_, Bool) \quad \text{-- Inferred: } Bool \rightarrow (Bool, Bool)$$
$$foo\ x = (x, x)$$

⇒ Combine *type checking* with *type inference*.

# Motivation

# Motivation

During development:

- Functions & types change frequently

# Motivation

During development:

- Functions & types change frequently
- Type signatures need to be updated

# Motivation

During development:

- Functions & types change frequently
- Type signatures need to be updated
- Type signatures are omitted

# Motivation

During development:

- Functions & types change frequently
- Type signatures need to be updated
- Type signatures are omitted
- Documentation & type checking against signature lost

# Motivation

During development:

- Functions & types change frequently
- Type signatures need to be updated
- Type signatures are omitted
- Documentation & type checking against signature lost

$\Rightarrow$ Partial type signatures

# Motivation

During development:

- Functions & types change frequently
- Type signatures need to be updated
- Type signatures are omitted
- Documentation & type checking against signature lost

$\Rightarrow$ Partial type signatures

Annotate the fixed parts of the type and replace the variable parts with wildcards.

# Motivation

# Motivation

*The complete type is not yet known.*

# Motivation

*The complete type is not yet known.*
$\Rightarrow$ Agda-style hole-driven development

# Motivation

*The complete type is not yet known.*
$\Rightarrow$ Agda-style hole-driven development

$$bar :: \_ \rightarrow (Char, Int) \rightarrow \_$$
$$bar\, f\, (x, y) = \neg\, (f\, x\, y)$$

# MOTIVATION

*The complete type is not yet known.*

$\Rightarrow$ Agda-style hole-driven development

$$bar :: \_ \rightarrow (Char, Int) \rightarrow \_$$
$$bar\, f\, (x, y) = \neg\ (f\, x\, y)$$

```
Found hole ‘_’ with type: Char -> Int -> Bool
    In the type signature:
      bar :: _ -> (Char, Int) -> _
    ...
```

# Motivation

*The complete type is not yet known.*

$\Rightarrow$ Agda-style hole-driven development

$$bar :: (Char \to Int \to Bool) \to (Char, Int) \to \_$$
$$bar\, f\, (x, y) = \neg\, (f\, x\, y)$$

```
Found hole '_' with type: Char -> Int -> Bool
    In the type signature:
      bar :: _ -> (Char, Int) -> _
    ...
```

# MOTIVATION

*The complete type is not yet known.*
$\Rightarrow$ Agda-style hole-driven development

$$bar :: (Char \to Int \to Bool) \to (Char, Int) \to \_$$
$$bar\, f\, (x, y) = \neg\, (f\, x\, y)$$

```
Found hole '_' with type: Bool
    In the type signature:
      bar :: _ -> (Char, Int) -> _
    ...
```

# Motivation

*The complete type is not yet known.*
$\Rightarrow$ Agda-style hole-driven development

$$bar :: (Char \to Int \to Bool) \to (Char, Int) \to Bool$$
$$bar\, f\, (x, y) = \neg\, (f\, x\, y)$$

```
Found hole '_' with type: Bool
    In the type signature:
      bar :: _ -> (Char, Int) -> _
    ...
```

# Motivation

*The complete type is not yet known.*
⇒ Agda-style hole-driven development

$$bar :: (Char \to Int \to Bool) \to (Char, Int) \to Bool$$
$$bar\, f\, (x, y) = \neg\, (f\, x\, y)$$

Emacs support for TypedHoles thanks to
Alejandro Serrano Mena's GSOC project.
Relatively easy to add support for PartialTypeSignatures.

# Motivation

*The complete type is not yet known.*
$\Rightarrow$ Agda-style hole-driven development

{-# LANGUAGE PartialTypeSignatures #-}

$$bar :: \_ \rightarrow (Char, Int) \rightarrow \_$$
$$bar\,f\,(x, y) = \neg\,(f\,x\,y)$$

No need to fill them in!

# Motivation

$$
\begin{aligned}
&\mathit{replaceLoopsRuleP} :: (\mathit{ProductionRule}\ p,\\
&\quad \mathit{EpsProductionRule}\ p,\\
&\quad \mathit{RecProductionRule}\ p\ \mathit{phi}\ r,\\
&\quad \mathit{TokenProductionRule}\ p\ t,\\
&\quad \mathit{PenaltyProductionRule}\ p) \Rightarrow\\
&\quad \mathit{PenaltyExtendedContextFreeRule}\ \mathit{phi}\ r\ t\ v \rightarrow\\
&\quad (\forall \mathit{ix}.\mathit{phi}\ \mathit{ix} \rightarrow p\ [r\ \mathit{ix}]) \rightarrow (\forall \mathit{ix}.\mathit{phi}\ \mathit{ix} \rightarrow p\ [r\ \mathit{ix}]) \rightarrow p\ v
\end{aligned}
$$

# Motivation

$$\begin{aligned}&\mathit{replaceLoopsRuleP} :: (\mathit{ProductionRule}\ p,\\ &\quad \mathit{EpsProductionRule}\ p,\\ &\quad \mathit{RecProductionRule}\ p\ \mathit{phi}\ r,\\ &\quad \mathit{TokenProductionRule}\ p\ t,\\ &\quad \mathit{PenaltyProductionRule}\ p) \Rightarrow\\ &\quad \mathit{PenaltyExtendedContextFreeRule}\ \mathit{phi}\ r\ t\ v \rightarrow\\ &\quad (\forall ix.\mathit{phi}\ ix \rightarrow p\ [r\ ix]) \rightarrow (\forall ix.\mathit{phi}\ ix \rightarrow p\ [r\ ix]) \rightarrow p\ v\end{aligned}$$

Distinguish important type information from distracting type information

# Motivation

$$\begin{aligned}
&\mathit{replaceLoopsRuleP} :: (\mathit{ProductionRule}\ p,\\
&\quad \mathit{EpsProductionRule}\ p,\\
&\quad \mathit{RecProductionRule}\ p\ \mathit{phi}\ r,\\
&\quad \mathit{TokenProductionRule}\ p\ t,\\
&\quad \mathit{PenaltyProductionRule}\ p) \Rightarrow\\
&\quad \mathit{PenaltyExtendedContextFreeRule}\ \mathit{phi}\ r\ t\ v \to\\
&\quad (\forall \mathit{ix}.\mathit{phi}\ \mathit{ix} \to p\ [r\ \mathit{ix}]) \to (\forall \mathit{ix}.\mathit{phi}\ \mathit{ix} \to p\ [r\ \mathit{ix}]) \to p\ v
\end{aligned}$$

Distinguish important type information from distracting type information

$$\begin{aligned}
&\mathit{replaceLoopsRuleP} :: _ \Rightarrow\\
&\quad \mathit{PenaltyExtendedContextFreeRule}\ \mathit{phi}\ r\ t\ v \to\\
&\quad (\forall \mathit{ix}.\mathit{phi}\ \mathit{ix} \to p\ [r\ \mathit{ix}]) \to (\forall \mathit{ix}.\mathit{phi}\ \mathit{ix} \to p\ [r\ \mathit{ix}]) \to p\ v
\end{aligned}$$

# Motivation

Noninferable types, e.g. higher-rank types:

$$\mathit{foo}\ x = (x\ [\mathit{True}, \mathit{False}], x\ [\texttt{'a'}, \texttt{'b'}])$$
$$\mathit{test} = \mathit{foo}\ \mathit{reverse} \quad \text{-- } \mathit{reverse} :: \forall a.[a] \to [a]$$

# Motivation

Noninferable types, e.g. higher-rank types:

$$\mathit{foo} :: (\forall a.[a] \to [a]) \to ([\mathit{Bool}], [\mathit{Char}])$$
$$\mathit{foo}\ x = (x\ [\mathit{True}, \mathit{False}], x\ [\texttt{'a'}, \texttt{'b'}])$$
$$\mathit{test} = \mathit{foo}\ \mathit{reverse} \quad \text{-- } \mathit{reverse} :: \forall a.[a] \to [a]$$

# Motivation

Noninferable types, e.g. higher-rank types:

$$foo :: (\forall a.[a] \to [a]) \to \_$$
$$foo\ x = (x\ [True, False], x\ [\texttt{'a'}, \texttt{'b'}])$$
$$test = foo\ reverse \quad \text{-- } reverse :: \forall a.[a] \to [a]$$

# Syntax

# Type Wildcards

## Syntax

$$
\begin{aligned}
&\mathit{filter} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a] \\
&\mathit{filter}\ \_ \qquad [\,] = [\,] \\
&\mathit{filter}\ \mathit{pred}\ (x : \mathit{xs}) \\
&\ \mid \mathit{pred}\ x \qquad = x : \mathit{filter}\ \mathit{pred}\ \mathit{xs} \\
&\ \mid \mathit{otherwise} = \qquad \mathit{filter}\ \mathit{pred}\ \mathit{xs}
\end{aligned}
$$

# Type Wildcards

## Syntax

$$filter :: (a \to \_) \to [a] \to [a]$$

$$
\begin{aligned}
&filter\ \_\ \ [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\ \mid pred\ x \quad = x : filter\ pred\ xs \\
&\ \mid otherwise = \quad filter\ pred\ xs
\end{aligned}
$$

# Type Wildcards

## Syntax

$$filter :: (\_ \rightarrow Bool) \rightarrow [a] \rightarrow [a]$$

$$\begin{aligned}
&filter\ _\quad [\,] = [\,]\\
&filter\ pred\ (x : xs)\\
&\quad \mid pred\ x \quad = x : filter\ pred\ xs\\
&\quad \mid otherwise = \quad filter\ pred\ xs
\end{aligned}$$

# Type Wildcards

## Syntax

$$filter :: \_ \to [a] \to [a]$$

$$
\begin{aligned}
&filter\ \_ \quad\ \ [\,] = [\,]\\
&filter\ pred\ (x : xs)\\
&\quad \mid pred\ x \qquad = x : filter\ pred\ xs\\
&\quad \mid otherwise = \qquad filter\ pred\ xs
\end{aligned}
$$

# Type Wildcards

## Syntax

$$filter :: \_ \rightarrow [a] \rightarrow [\_]$$

$$\begin{aligned}
&filter\ _\quad [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\ \mid pred\ x \quad = x : filter\ pred\ xs \\
&\ \mid otherwise = \quad filter\ pred\ xs
\end{aligned}$$

# Type Wildcards

## Syntax

$$filter :: \_ \rightarrow [a] \rightarrow \_$$

$$\begin{aligned}
&filter\ _\quad\ [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\quad \mid pred\ x \quad\ \ = x : filter\ pred\ xs \\
&\quad \mid otherwise = \quad\ \ filter\ pred\ xs
\end{aligned}$$

# Type Wildcards

## Syntax

$$\mathit{filter} :: \_ \to \_ \to \_$$

$$
\begin{aligned}
&\mathit{filter}\ \_\qquad [\,] = [\,]\\
&\mathit{filter}\ \mathit{pred}\ (x : \mathit{xs})\\
&\quad \mid \mathit{pred}\ x \qquad = x : \mathit{filter}\ \mathit{pred}\ \mathit{xs}\\
&\quad \mid \mathit{otherwise} = \qquad \mathit{filter}\ \mathit{pred}\ \mathit{xs}
\end{aligned}
$$

# Type Wildcards

## Syntax

$$filter :: \_ \rightarrow \_$$

$$\begin{aligned}
&filter\ _\ \ [\,] = [\,]\\
&filter\ pred\ (x : xs)\\
&\ \mid pred\ x \quad = x : filter\ pred\ xs\\
&\ \mid otherwise = \quad filter\ pred\ xs
\end{aligned}$$

# Type Wildcards

## Syntax

$$filter :: \_$$

$$\begin{aligned}
&filter\ _\ \ [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\ \mid pred\ x \quad = x : filter\ pred\ xs \\
&\ \mid otherwise = \quad filter\ pred\ xs
\end{aligned}$$

# Type Wildcards

## Syntax

$$
\begin{array}{l}
\mathit{filter}\ \_\ \ \ \ \ \ [\,] = [\,] \\
\mathit{filter}\ \mathit{pred}\ (x : \mathit{xs}) \\
\ \ \mid \mathit{pred}\ x \ \ \ \ \ \ = x : \mathit{filter}\ \mathit{pred}\ \mathit{xs} \\
\ \ \mid \mathit{otherwise} = \ \ \ \ \ \ \mathit{filter}\ \mathit{pred}\ \mathit{xs}
\end{array}
$$

# Named Wildcards

## Syntax

# Named Wildcards

## Syntax

$$
\begin{aligned}
&\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}\ \_ \quad\quad [\,] = [\,] \\
&\mathit{filter}\ \mathit{pred}\ (x : \mathit{xs}) \\
&\ \mid \mathit{pred}\ x \quad\quad = x : \mathit{filter}\ \mathit{pred}\ \mathit{xs} \\
&\ \mid \mathit{otherwise} = \quad\ \ \mathit{filter}\ \mathit{pred}\ \mathit{xs}
\end{aligned}
$$

# Named Wildcards

## Syntax

$$filter :: (\_x \to \_x) \to [\_x] \to [\_x]$$

$$
\begin{aligned}
&filter\ \_\ \ [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\quad \mid pred\ x \quad = x : filter\ pred\ xs \\
&\quad \mid otherwise = \quad filter\ pred\ xs
\end{aligned}
$$

# Named Wildcards

## Syntax

Inferred: $(Bool \to Bool) \to [Bool] \to [Bool]$

$$filter :: (\_x \to \_x) \to [\_x] \to [\_x]$$

$$\begin{aligned}
&filter\ _\qquad [\,] = [\,]\\
&filter\ pred\ (x : xs)\\
&\quad \mid pred\ x \qquad = x : filter\ pred\ xs\\
&\quad \mid otherwise = \qquad filter\ pred\ xs
\end{aligned}$$

# Named Wildcards

## Syntax

$$filter :: (\_x \rightarrow Bool) \rightarrow [\_x] \rightarrow [\_x]$$

$$\begin{aligned}
&filter\ _\quad\ [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\ \mid pred\ x \quad\ \ = x : filter\ pred\ xs \\
&\ \mid otherwise = \quad\ \ filter\ pred\ xs
\end{aligned}$$

# Named Wildcards

## Syntax

Inferred: $(w\_x \rightarrow Bool) \rightarrow [w\_x] \rightarrow [w\_x]$

$$filter :: (\_x \rightarrow Bool) \rightarrow [\_x] \rightarrow [\_x]$$

$$\begin{aligned}
&filter\ _\quad\ [\,] = [\,] \\
&filter\ pred\ (x : xs) \\
&\quad \mid pred\ x \quad = x : filter\ pred\ xs \\
&\quad \mid otherwise = \quad filter\ pred\ xs
\end{aligned}$$

# Named Wildcards

## Syntax

$$eq :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$eq\ x\ y = x \equiv y$$

# Named Wildcards

## Syntax

$$eq :: Eq\ \_x \Rightarrow \_x \rightarrow \_x \rightarrow Bool$$

$$eq\ x\ y = x \equiv y$$

# Named Wildcards

## Syntax

Inferred: $Eq\ w\_x \Rightarrow w\_x \rightarrow w\_x \rightarrow Bool$

$$eq :: Eq\ \_x \Rightarrow \_x \rightarrow \_x \rightarrow Bool$$

$eq\ x\ y = x \equiv y$

# Named Wildcards

## Syntax

$$eq :: Eq\ \_x \Rightarrow \_x \rightarrow \_x \rightarrow \_x$$

$$eq\ x\ y = x \equiv y$$

# Named Wildcards

## Syntax

Inferred: $Eq\ Bool \Rightarrow Bool \rightarrow Bool \rightarrow Bool$

$$eq :: Eq\ \_x \Rightarrow \_x \rightarrow \_x \rightarrow \_x$$

$$eq\ x\ y = x \equiv y$$

# Named Wildcards

## Syntax

Inferred: $\mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$

$$eq :: \mathit{Eq}\ \_x \Rightarrow \_x \rightarrow \_x \rightarrow \_x$$

$$eq\ x\ y = x \equiv y$$

# Constraint wildcards

## Syntax

# Constraint wildcards

## Syntax

$$bar :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$bar\ x\ y = x \equiv y$$

-- class Eq a => Ord x

# Constraint wildcards

## Syntax

$$bar :: Ord\ \_ \Rightarrow a \rightarrow a \rightarrow Bool$$

$bar\ x\ y = x \equiv y$

-- class Eq a => Ord x

# Constraint wildcards

## Syntax

Mismatch: inferred $Eq\ a$ vs. annotated $Ord\ \_$

$$bar :: Ord\ \_ \Rightarrow a \rightarrow a \rightarrow Bool$$

$bar\ x\ y = x \equiv y$

```
-- class Eq a => Ord x
```

# Constraint wildcards

## Syntax

# Constraint wildcards

## Syntax

$$foo :: (Show\ a, Num\ a) \Rightarrow a \rightarrow String$$
$$foo\ x = show\ (x + 1)$$

# Constraint wildcards

## Syntax

$$foo :: \_\ a \Rightarrow a \rightarrow String$$

$$foo\ x = show\ (x + 1)$$

# Constraint wildcards

## Syntax

Infer? $\mathit{Show}\ a \Rightarrow a \rightarrow \mathit{String}$

$$\mathit{foo} :: \_\ a \Rightarrow a \rightarrow \mathit{String}$$
$$\mathit{foo}\ x = \mathit{show}\ (x + 1)$$

# Constraint wildcards

## Syntax

Infer? $Num\ a \Rightarrow a \rightarrow String$

$$foo :: \_\ a \Rightarrow a \rightarrow String$$

$$foo\ x = show\ (x+1)$$

# Constraint wildcards

## Syntax

Compromise

- Only named wildcards in constraints…
- …when present in the rest of the type

# Constraint wildcards

## Syntax

Compromise

- Only named wildcards in constraints…
- …when present in the rest of the type

$Eq\ \_ \Rightarrow a \rightarrow a \rightarrow Bool$ No

# Constraint wildcards

## Syntax

Compromise

- Only named wildcards in constraints…
- …when present in the rest of the type

| | |
|---|---|
| $Eq\ \_ \Rightarrow a \rightarrow a \rightarrow Bool$ | No |
| $Eq\ \_x \Rightarrow a \rightarrow a \rightarrow Bool$ | No |

# Constraint wildcards

## Syntax

Compromise

- Only named wildcards in constraints …
- … when present in the rest of the type

| | |
|---|---|
| $Eq\ \_ \Rightarrow a \rightarrow a \rightarrow Bool$ | No |
| $Eq\ \_x \Rightarrow a \rightarrow a \rightarrow Bool$ | No |
| $Eq\ \_x \Rightarrow \_x \rightarrow \_x \rightarrow Bool$ | Yes |

# Extra-constraints Wildcard

## Syntax

# Extra-constraints Wildcard

## Syntax

$$foo :: (Show\ a, Num\ a) \Rightarrow a \rightarrow String$$
$$foo\ x = show\ (x + 1)$$

# Extra-constraints Wildcard

## Syntax

$$foo :: \_ \Rightarrow a \rightarrow String$$

$$foo\ x = show\ (x + 1)$$

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $(Show\ a, Num\ a)$

$$foo :: \_ \Rightarrow a \rightarrow String$$

$foo\ x = show\ (x + 1)$

# Extra-constraints Wildcard

## Syntax

$$foo :: (Num\ a, \_) \Rightarrow a \rightarrow String$$

$foo\ x = show\ (x + 1)$

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $\mathit{Show}\ a$

$$\mathit{foo} :: (\mathit{Num}\ a, \_) \Rightarrow a \rightarrow \mathit{String}$$

$$\mathit{foo}\ x = \mathit{show}\ (x + 1)$$

# Extra-constraints Wildcard

Syntax

# Extra-constraints Wildcard

## Syntax

$$bar :: Show\ a \Rightarrow a \rightarrow a$$
$$bar\ x = show\ x$$

# Extra-constraints Wildcard

## Syntax

$$bar :: \_ \Rightarrow a \to a$$

$$bar\ x = show\ x$$

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $Show\ a$

Inferred: $Show\ a \Rightarrow a \rightarrow a$

$$bar :: \_ \Rightarrow a \rightarrow a$$

$$bar\ x = show\ x$$

# Extra-constraints Wildcard

## Syntax

$$bar :: (Num\ a, \_) \Rightarrow a \rightarrow a$$

$$bar\ x = show\ x$$

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $\mathit{Show}\ a$
Inferred: $(\mathit{Num}\ a, \mathit{Show}\ a) \Rightarrow a \rightarrow a$

$$\mathit{bar} :: (\mathit{Num}\ a, \_) \Rightarrow a \rightarrow a$$

$\mathit{bar}\ x = \mathit{show}\ x$

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $\mathit{Show}\ a$

Inferred: $(\mathit{Num}\ a, \mathit{Show}\ a) \Rightarrow a \to a$

$$\mathit{bar} :: (\mathit{Num}\ a, \_) \Rightarrow a \to a$$

$$\mathit{bar}\ x = \mathit{show}\ x$$

Proposed simplification:
ignore annotated constraints

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $\mathit{Show}\ a$

Inferred: $(\cancel{\mathit{Num}\ a}, \mathit{Show}\ a) \Rightarrow a \to a$

$$\mathit{bar} :: (\mathit{Num}\ a, \_) \Rightarrow a \to a$$

$$\mathit{bar}\ x = \mathit{show}\ x$$

Proposed simplification:
ignore annotated constraints

# Extra-constraints Wildcard

## Syntax

Inferred constraints: $\mathit{Show}\ a$
Inferred: $\mathit{Show}\ a \Rightarrow a \rightarrow a$

$$bar :: (\mathit{Num}\ a, \_) \Rightarrow a \rightarrow a$$

$$bar\ x = show\ x$$

Proposed simplification:
ignore annotated constraints

# Formalisation

*Partial Type Signatures for Haskell*.
Thomas Winant, Dominique Devriese,
Frank Piessens, Tom Schrijvers.
In *Practical Aspects of Declarative Languages* 2014
(PADL'14)

# Idea

## Formalisation

# Idea

## Formalisation

$$secondArg :: \_ \rightarrow \_ \rightarrow Bool$$
$$secondArg\ \_\ x = x$$

# Idea

## Formalisation

$$secondArg\ \_\ x = x$$

# Idea

## Formalisation

$$\textit{secondArg}\ \underbrace{\_}_{\alpha}\ \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

# Idea

## Formalisation

$$\mathit{secondArg}\ \underbrace{\_}_{\alpha}\ \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

# Idea

## Formalisation

$$\textit{secondArg}\ \underbrace{\_}_{\alpha}\ \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$

# Idea

## Formalisation

$$\textit{secondArg}\ \underbrace{\_}_{\alpha}\ \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\leadsto \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$
$\Rightarrow \textit{secondArg} :: \alpha \to \beta \to \beta$

# Idea

## Formalisation

$$\mathit{secondArg}\ \underbrace{\_}_{\alpha}\ \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\leadsto \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$

$\Rightarrow \mathit{secondArg} :: \alpha \to \beta \to \beta$

$\Rightarrow$ Generalise: $\mathit{secondArg} :: \forall a\ b. a \to b \to b$

# Idea

## Formalisation

$$secondArg :: \_ \to \_ \to Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\leadsto \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$

$\Rightarrow secondArg :: \alpha \to \beta \to \beta$

$\Rightarrow$ Generalise: $secondArg :: \forall a\ b.a \to b \to b$

# Idea

## Formalisation

$$secondArg :: \_ \rightarrow \_ \rightarrow Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \rightarrow \beta \rightarrow \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$

$\Rightarrow secondArg :: \alpha \rightarrow \beta \rightarrow \beta$

$\Rightarrow$ Generalise: $secondArg :: \forall a\ b . a \rightarrow b \rightarrow b$

Idea: replace wildcards with unification variables

# Idea

## Formalisation

$$secondArg :: \_ \to \_ \to Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma)}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$

$\Rightarrow secondArg :: \alpha \to \beta \to \beta$

$\Rightarrow$ Generalise: $secondArg :: \forall a\ b. a \to b \to b$

Idea: replace wildcards with unification variables

Wildcard desugaring relation:

$$(\_ \to \_ \to Bool) \Rrightarrow (\omega_1 \to \omega_2 \to Bool)$$

# Idea

## Formalisation

$$secondArg :: \_ \to \_ \to Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma, (\omega_1 \to \omega_2 \to Bool) \sim (\alpha \to \beta \to \gamma))}_{\text{Constraints}}$$

Solve the constraints: $[\gamma \mapsto \beta]$

$\Rightarrow secondArg :: \alpha \to \beta \to \beta$

$\Rightarrow$ Generalise: $secondArg :: \forall a\ b.a \to b \to b$

Idea: replace wildcards with unification variables

Wildcard desugaring relation:

$$(\_ \to \_ \to Bool) \Rrightarrow (\omega_1 \to \omega_2 \to Bool)$$

# Idea

## Formalisation

$$secondArg :: \_ \to \_ \to Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma, (\omega_1 \to \omega_2 \to Bool) \sim (\alpha \to \beta \to \gamma))}_{\text{Constraints}}$$

Solve the constraints:

$[\gamma \mapsto Bool, \beta \mapsto Bool, \omega_2 \mapsto Bool, \alpha \mapsto \omega_1]$

Idea: replace wildcards with unification variables

Wildcard desugaring relation:

$$(\_ \to \_ \to Bool) \Rrightarrow (\omega_1 \to \omega_2 \to Bool)$$

# Idea

## Formalisation

$$secondArg :: \_ \to \_ \to Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma, (\omega_1 \to \omega_2 \to Bool) \sim (\alpha \to \beta \to \gamma))}_{\text{Constraints}}$$

Solve the constraints:

$[\gamma \mapsto Bool, \beta \mapsto Bool, \omega_2 \mapsto Bool, \alpha \mapsto \omega_1]$

$\Rightarrow secondArg :: \omega_1 \to Bool \to Bool$

Idea: replace wildcards with unification variables

Wildcard desugaring relation:

$$(\_ \to \_ \to Bool) \Rrightarrow (\omega_1 \to \omega_2 \to Bool)$$

# Idea

## Formalisation

$$secondArg :: \_ \to \_ \to Bool$$

$$secondArg \underbrace{\_}_{\alpha} \underbrace{x}_{\beta} = \overbrace{\underbrace{x}_{\gamma}}^{\beta} : \overbrace{\alpha \to \beta \to \gamma}^{\text{type}}$$

$$\rightsquigarrow \underbrace{(\beta \sim \gamma, (\omega_1 \to \omega_2 \to Bool) \sim (\alpha \to \beta \to \gamma))}_{\text{Constraints}}$$

Solve the constraints:

$[\gamma \mapsto Bool, \beta \mapsto Bool, \omega_2 \mapsto Bool, \alpha \mapsto \omega_1]$

$\Rightarrow secondArg :: \omega_1 \to Bool \to Bool$

$\Rightarrow$ Generalise: $secondArg :: \forall a.a \to Bool \to Bool$

Idea: replace wildcards with unification variables

Wildcard desugaring relation:

$$(\_ \to \_ \to Bool) \Rrightarrow (\omega_1 \to \omega_2 \to Bool)$$

# Proofs

## Formalisation

**Theorem 1: Conservative extension**
*For functions with non-partial type signatures,* GHC *infers the same types as before.*

# Proofs

## Formalisation

**Theorem 1: Conservative extension**
*For functions with non-partial type signatures,* GHC *infers the same types as before.*

**Theorem 2: Generalisation of type inference**
$f :: \_ \Rightarrow \_ = e$ *is equivalent with* $f = e$.

# Proofs

## Formalisation

**Theorem 1: Conservative extension**
*For functions with non-partial type signatures,* GHC *infers the same types as before.*

**Theorem 2: Generalisation of type inference**
$f :: \_ \Rightarrow \_ = e$ *is equivalent with* $f = e$.

**Theorem 3: Algorithm soundness**

# Implementation

# Implementation

- Parser support for wildcards

# Implementation

- Parser support for wildcards
- Named wildcard syntax clashes with type variable syntax:

$$\mathit{foo} :: \_a \to \_a$$
$$\mathit{foo}\ x = \neg\ x$$

# Implementation

- Parser support for wildcards
- Named wildcard syntax clashes with type variable syntax:

$$foo :: \_a \rightarrow \_a$$
$$foo\ x = \neg\ x$$

```
Couldn't match expected type '_a'
  with actual type 'Bool'
  '_a' is a rigid type variable bound by ...
```

# Implementation

- Parser support for wildcards
- Named wildcard syntax clashes with type variable syntax:

  {-# LANGUAGE NamedWildcards #-}

  $$foo :: \_a \rightarrow \_a$$
  $$foo\ x = \neg\ x$$

```
Couldn't match expected type '_a'
  with actual type 'Bool'
  '_a' is a rigid type variable bound by ...
```

  backwards compatible unless the `NamedWildcards` extension is enabled.

# Implementation

- Parser support for wildcards
- Named wildcard syntax clashes with type variable syntax:

  {-# LANGUAGE NamedWildcards #-}

  $$\mathit{foo} :: \_a \to \_a$$
  $$\mathit{foo}\ x = \neg\ x$$

```
Found hole '_' with type: Bool
    In the type signature:
      foo :: _a -> _a
```

  backwards compatible unless the `NamedWildcards` extension is enabled.

# Implementation

- Disallow wildcards in particular types:

$$\textbf{class}\ \mathit{Show}\ a\ \textbf{where}$$
$$\quad \mathit{show} :: a \rightarrow \_$$
$$\textbf{instance}\ \mathit{Show}\ \_\ \textbf{where}\ \ldots$$
$$\textbf{data}\ \mathit{Foo} = \{\, \mathit{bar} :: \mathit{Maybe}\ \_ \,\}$$
$$\ldots$$

# Implementation

- Quantify desugared wildcards per *TypeSig*, imitating the scoping behaviour of `ScopedTypeVariables`.

# Implementation

- Quantify desugared wildcards per $\mathit{TypeSig}$, imitating the scoping behaviour of `ScopedTypeVariables`.

```
{-# LANGUAGE NamedWildcards #-}
```

$$
\begin{aligned}
&\mathit{foo} :: \_a \rightarrow \mathit{Char}\\
&\mathit{foo}\ x = \mathbf{let}\ v = \neg\ x\\
&\qquad\qquad\quad g :: \_a \rightarrow \_a\\
&\qquad\qquad\quad g\ y = y\\
&\qquad\quad\ \ \mathbf{in}\ (g\ \texttt{'z'})
\end{aligned}
$$

# Implementation

- Quantify desugared wildcards per *TypeSig*, imitating the scoping behaviour of `ScopedTypeVariables`.

{-# LANGUAGE NamedWildcards, ScopedTypeVariables #-}

$$
\begin{aligned}
& foo :: \_a \to Char \\
& foo\ x = \mathbf{let}\ v = \neg\ x \\
& \qquad\qquad g :: \_a \to \_a \\
& \qquad\qquad g\ y = y \\
& \qquad\quad \mathbf{in}\ (g\ \texttt{'z'})
\end{aligned}
$$

```
Couldn't match expected type ‘Bool’
  with actual type ‘Char’
In the first argument of ‘g’, namely ‘'z'’
```

# Implementation

- Just like for `TypedHoles`, when type checking, we generate an insoluble hole constraint between each wildcard unification variable and its inferred type.

# Implementation

- Just like for `TypedHoles`, when type checking, we generate an insoluble hole constraint between each wildcard unification variable and its inferred type.
- After solving the constraints, these hole constraints are left over, and are converted into error messages.

# Implementation

- Just like for `TypedHoles`, when type checking, we generate an insoluble hole constraint between each wildcard unification variable and its inferred type.
- After solving the constraints, these hole constraints are left over, and are converted into error messages.
- They are not generated when `PartialTypeSignatures` is enabled.

# Code

## Implementation

| | |
|---|---|
| Code | `https://github.com/mrBliss/ghc` |
| Phabricator | `https://phabricator.haskell.org/D168` |
| Trac Ticket | #9478 |

Coming to GHC some time soon!

# Thank you

Q & A