Spine-local Type Inference in Cedille

Christopher Jenkins and Aaron Stump

Computer Science
University of Iowa

ICFP '18 (Cedille Tutorial)

Outline

Outline

1. Background and Motivation
   - Local Type Inference
   - Spine-local Type Inference

2. Detailed Example

3. Annotation Requirements

What is “Local Type Inference”?

- Uses two main techniques
  - *Bidirectional typing rules:*
  - *Local type-argument inference:*

What is “Local Type Inference”?

- Uses two main techniques
  - *Bidirectional typing rules:*

Synthesis mode:	$\lambda\, x : Nat.\, x$	$\Uparrow$	$Nat \to Nat$
Checking mode:	$\lambda\, x.\, x$	$\Downarrow$	$Nat \to Nat$

  - *Local type-argument inference:*

What is "Local Type Inference"?

- Uses two main techniques
  - *Bidirectional typing rules:*

Synthesis mode:	$\lambda\, x : Nat.\, x$	$\Uparrow$	$Nat \to Nat$
Checking mode:	$\lambda\, x.\, x$	$\Downarrow$	$Nat \to Nat$

  - *Local type-argument inference:*

Let	$id : \forall\, X.\, X \to X$	
Type	$\boxed{id\ 0}$	$\Uparrow Nat$
Infer	$X = Nat$	from 0

$\boxed{\text{Local}}$ and Synthetic

Why use local type inference?

- It is a method of *partial* type inference
  - *Complete* type inference: no annotations **ever** (e.g. *Damas-Hindley-Milner* and ML)
  - Undecidable for System F (let alone Cedille!)

Why use local type inference?

- It is a method of *partial* type inference
  - *Complete* type inference: no annotations **ever** (e.g. *Damas-Hindley-Milner* and ML)
  - Undecidable for System F (let alone Cedille!)
- It is user-friendly
  - Infers many type annotations
  - Predictable annotation requirements
  - Better-quality error messages

Why use local type inference?

- It is a method of *partial* type inference
  - *Complete* type inference: no annotations **ever** (e.g. *Damas-Hindley-Milner* and ML)
  - Undecidable for System F (let alone Cedille!)
- It is user-friendly
  - Infers many type annotations
  - Predictable annotation requirements
  - Better-quality error messages
- It is implementer-friendly
  - Relatively simple implementation
  - *Extensible*: new features added without threatening decidability

Local Type Inference in Cedille

Cedille provides a way to interrogate the two core features of LTI

- *Bidirectional Typechecking* gets special highlighting
- *Type-argument Inference* gets a dedicated buffer

Local Type Inference in Cedille

Cedille provides a way to interrogate the two core features of LTI

- *Bidirectional Typechecking* gets special highlighting
  - In navigation mode, type `C-h 3` to see *bidirectional highlighting*.
- *Type-argument Inference* gets a dedicated buffer

Local Type Inference in Cedille

Cedille provides a way to interrogate the two core features of LTI

- *Bidirectional Typechecking* gets special highlighting
  - In navigation mode, type `C-h 3` to see *bidirectional highlighting*.
- *Type-argument Inference* gets a dedicated buffer
  - In navigation mode, type `m` to see the *meta-variables buffer.*
  - **all** meta-variables present, **where** introduced, **what** solutions

Local Type Inference in Cedille

Cedille provides a way to interrogate the two core features of LTI

- *Bidirectional Typechecking* gets special highlighting
  - In navigation mode, type `C-h 3` to see *bidirectional highlighting*.
- *Type-argument Inference* gets a dedicated buffer
  - In navigation mode, type `m` to see the *meta-variables buffer.*
  - **all** meta-variables present, **where** introduced, **what** solutions

Cedille also has a novel inference system: *spine-local type inference*

Limitations of other LTI Systems

Why a novel system? Other local type inference systems can sometimes still require "silly" type annotations...

Assume	*mkpair*	:	$\forall X\ Y.\, X \rightarrow Y \rightarrow Pair\ \cdot X\ \cdot Y$
Type	$mkpair\ (\lambda\, x.\, x)\ 0$		

Limitations of other LTI Systems

Why a novel system? Other local type inference systems can sometimes still require "silly" type annotations...

$$\begin{array}{lccl} \text{Assume} & \mathit{mkpair} & : & \forall X\ Y.\, X \to Y \to \mathit{Pair}\ \cdot X\ \cdot Y \\ \text{Type} & \mathit{mkpair}\ (\lambda x.\, x)\ 0 & \Uparrow & ??? \end{array}$$

- We do not expect to locally synthesize a type

Limitations of other LTI Systems

Why a novel system? Other local type inference systems can sometimes still require "silly" type annotations...

Assume	$mkpair$	:	$\forall X\ Y.\, X \to Y \to Pair\ \cdot X\ \cdot Y$
Type	$mkpair\ (\lambda x.\, x)\ 0$	$\Uparrow$	???

- We do not expect to locally synthesize a type

Limitations of other LTI Systems

Why a novel system? Other local type inference systems can sometimes still require "silly" type annotations...

Assume	$mkpair$	:	$\forall X\ Y. X \to Y \to Pair \cdot X \cdot Y$
Type	$mkpair\ (\lambda\, x.\, x)\ 0$	$\Uparrow$	???
Type	$mkpair\ (\lambda\, x.\, x)\ 0$	$\Downarrow$	$Pair \cdot (Nat \to Nat) \cdot Nat$

- We do not expect to locally synthesize a type
- ... but we would expect to check it against a type

Limitations of other LTI Systems

Why a novel system? Other local type inference systems can sometimes still require "silly" type annotations...

Assume	$mkpair$	$:$	$\forall X\ Y.\, X \rightarrow Y \rightarrow Pair \cdot X \cdot Y$
Type	$mkpair\ (\lambda\, x.\, x)\ 0$	$\Uparrow$	???
Type	$mkpair\ (\lambda\, x.\, x)\ 0$	$\Downarrow$	$Pair \cdot (Nat \rightarrow Nat) \cdot Nat$

- We do not expect to locally synthesize a type
- ... but we would expect to check it against a type
  - We could call this "contextual" type-argument inference.

Limitations of other LTI Systems

Why a novel system? Other local type inference systems can sometimes still require "silly" type annotations...

Assume	$mkpair$	:	$\forall X\ Y.\, X \to Y \to Pair\ \cdot X\ \cdot Y$
Type	$mkpair\ (\lambda\, x.\, x)\ 0$	$\Uparrow$	???
Type	$mkpair\ (\lambda\, x.\, x)\ 0$	$\Downarrow$	$Pair \cdot (Nat \to Nat) \cdot Nat$

- We do not expect to locally synthesize a type
- ... but we would expect to check it against a type
  - We could call this "contextual" type-argument inference.
- Unfortunately, not done in the two major published LTI systems
  - Popular "unofficial" extension (used in e.g. Scala, Rust)

Limitations of other LTI Systems (cont.)

- Usually uses "fully-uncurried" function applications

$$f\,(t_1, .., t_n)$$

  - Maximize available info at a single application

Limitations of other LTI Systems (cont.)

- Usually uses "fully-uncurried" function applications

$$f(t_1, .., t_n)$$

  - Maximize available info at a single application

- Usually without partial type application ("all-or-nothing")

$$f[T_1, ..., T_m](t_1, ..., t_n)$$

Our Contributions

- Type inference for some expressions not typed by other variants of local type inference, by using *contextual* type-argument inference
- Precise, specificational account of this technique
- Better support function currying and partial type applications by being "spine-local."

Our Contributions

- Type inference for some expressions not typed by other variants of local type inference, by using *contextual* type-argument inference

$$f\ \ t_1^{\Uparrow}\ \ t_2^{\Uparrow}\ \ t_3^{\Downarrow}$$

- Precise, specificational account of this technique
- Better support function currying and partial type applications by being "spine-local."

Our Contributions

- Type inference for some expressions not typed by other variants of local type inference, by using *contextual* type-argument inference

$$f\ \ t_1^{\Uparrow}\ \ t_2^{\Downarrow}\ \ t_3^{\Downarrow} \Downarrow T$$

- Precise, specificational account of this technique
- Better support function currying and partial type applications by being "spine-local."

Our Contributions

- Type inference for some expressions not typed by other variants of local type inference, by using *contextual* type-argument inference

$$f\ \ t_1^{\Uparrow}\ \ t_2^{\Downarrow}\ \ t_3^{\Downarrow}\ \Downarrow T$$

- Precise, specificational account of this technique
- Better support function currying and partial type applications by being "spine-local."

$$\boxed{f[S, T, V](t_1, t_2, t_3)}$$

Our Contributions

- Type inference for some expressions not typed by other variants of local type inference, by using *contextual* type-argument inference

$$f\ \ t_1^{\Uparrow}\ \ t_2^{\Downarrow}\ \ t_3^{\Downarrow}\ \Downarrow T$$

- Precise, specificational account of this technique
- Better support function currying and partial type applications by being "spine-local."

$$\boxed{f\ \cdot S\ \cdot T\ \cdot V\ \ t_1\ \ t_2\ \ t_3}$$

Our Contributions

- Type inference for some expressions not typed by other variants of local type inference, by using *contextual* type-argument inference

$$f\ \ t_1^{\Uparrow}\ \ t_2^{\Downarrow}\ \ t_3^{\Downarrow}\ \Downarrow\ T$$

- Precise, specificational account of this technique
- Better support function currying and partial type applications by being "spine-local."

$$\boxed{f\ \cdot S \qquad\qquad t_1\ \ t_2}$$

Room for Improvement

Type inference in Cedille will continue to be improved upon. Some things we want to address:

- Higher-order type arguments must be provided explicitly:
  $(\forall\, F : \star \rightarrow \star.\,)$

- Type arguments inferred *only* between applications:
  $nil \Downarrow List\ \ \cdot Nat$

Room for Improvement

Type inference in Cedille will continue to be improved upon. Some things we want to address:

- Higher-order type arguments must be provided explicitly:
  $(\forall\, F : \star \to \star.\ )$
  - **Comming soon:** second-order *matching* is decidable and finitary
- Type arguments inferred *only* between applications:
  $nil \Downarrow List \quad \cdot Nat$

Room for Improvement

Type inference in Cedille will continue to be improved upon. Some things we want to address:

- Higher-order type arguments must be provided explicitly:
  $(\forall\, F : \star \to \star.\ )$
  - **Comming soon:** second-order *matching* is decidable and finitary
- Type arguments inferred *only* between applications:
  $\cancel{nil \Downarrow List\ \ \cdot Nat}$ Must write $nil\ \ \cdot Nat$

Room for Improvement

Type inference in Cedille will continue to be improved upon. Some things we want to address:

- Higher-order type arguments must be provided explicitly:
  $(\forall F : \star \to \star.\ )$
  - **Comming soon:** second-order *matching* is decidable and finitary
- Type arguments inferred *only* between applications:
  $\cancel{nil \Downarrow List \cdot Nat}$ Must write $nil\ \cdot Nat$
  - **Coming soon:** polymorphic subsumption
    $\forall A. List \cdot A \ \leq\ List \cdot Nat$

Outline

1. Background and Motivation
   - Local Type Inference
   - Spine-local Type Inference
2. Detailed Example
3. Annotation Requirements

Terminology

- *Application head*: variable or abstraction

$$x, \quad \Lambda X.\, t, \quad \lambda x.\, t$$

Terminology

- *Application head*: variable or abstraction

$$x, \quad \Lambda X.\, t, \quad \lambda x.\, t$$

- *Application spine*: head followed by seq. of term, type arguments

$$\boxed{x \quad} \boxed{t_1 \;\; t_2 \;\; t_3} \text{ vs } (((x\ t_1)\ t_2)\ t_3)$$

Terminology

- *Application head*: variable or abstraction

$$x, \quad \Lambda X.\, t, \quad \lambda x.\, t$$

- *Application spine*: head followed by seq. of term, type arguments

$$\boxed{x \quad} \boxed{t_1 \;\; t_2 \;\; t_3} \text{ vs } (((x \;\; t_1) \;\; t_2) \;\; t_3)$$

- *Applicand:* Term in the function position of an application

$$t_1 \text{ in } t_1 \;\; t_2$$

Terminology

- *Application head*: variable or abstraction

$$x, \quad \Lambda X.\, t, \quad \lambda x.\, t$$

- *Application spine*: head followed by seq. of term, type arguments

$$\boxed{x \quad}\boxed{t_1 \;\; t_2 \;\; t_3} \text{ vs } (((x \;\; t_1) \;\; t_2) \;\; t_3)$$

- *Applicand:* Term in the function position of an application

$$t_1 \text{ in } t_1 \;\; t_2$$

- *Maximal application:* spine that is not an applicand

$$\begin{array}{ll} \text{Not max} & \underline{x \;\; t_1 \;\; t_2} \;\; t_3 \\ \text{Max} & \underline{x \;\; t_1 \;\; t_2 \;\; t_3} \end{array}$$

Example – High Level Goals

Assume	*const*	:	$\forall X\ Y.\, X \rightarrow Y \rightarrow X$
Assume	*zero*	:	*Nat*
Type	*const* $(\lambda\, x.\, x)$ *zero*	$\Downarrow$	$Nat \rightarrow Nat$

- Check the spine *const* $(\lambda\, x.\, x)$ *zero* against $Nat \rightarrow Nat$

Example – High Level Goals

Assume	$const$	:	$\forall X\ Y.\, X \to Y \to X$
Assume	$zero$	:	Nat
Type	$const\ (\lambda\, x.\, x)\ zero$	$\Downarrow$	$Nat \to Nat$

- Check the spine $const\ (\lambda\, x.\, x)\ zero$ against $Nat \to Nat$
- We fill in missing types to "elaborate" to $const\ \cdot (Nat \to Nat)\ \cdot Nat\ (\lambda\, x : Nat.\, x)\ zero$

Example – High Level Goals

$$\begin{array}{ccll} \text{Assume} & \mathit{const} & : & \forall X\ Y.\, X \to Y \to X \\ \text{Assume} & \mathit{zero} & : & \mathit{Nat} \\ \text{Type} & \mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} & \Downarrow & \mathit{Nat} \to \mathit{Nat} \end{array}$$

- Check the spine *const* $(\lambda\, x.\, x)$ *zero* against $\mathit{Nat} \to \mathit{Nat}$
- We fill in missing types to "elaborate" to $\mathit{const}\ \cdot (\mathit{Nat} \to \mathit{Nat})\ \cdot \mathit{Nat}\ (\lambda\, x : \mathit{Nat}.\, x)\ \mathit{zero}$
- X inferred contextually, Y synthetically

Example – High Level Goals

$$\begin{array}{ccccl} \text{Assume} & \mathit{const} & : & \forall X\ Y.\, X \to Y \to X \\ \text{Assume} & \mathit{zero} & : & \mathit{Nat} \\ \text{Type} & \mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} & \Downarrow & \mathit{Nat} \to \mathit{Nat} \end{array}$$

- Check the spine *const* $(\lambda\, x.\, x)$ *zero* against $\mathit{Nat} \to \mathit{Nat}$
- We fill in missing types to "elaborate" to $\mathit{const}\ \cdot (\mathit{Nat} \to \mathit{Nat})\ \cdot \mathit{Nat}\ (\lambda\, x : \mathit{Nat}.\, x)\ \mathit{zero}$
- X inferred contextually, Y synthetically
  In Cedille: $\boxed{\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}}$
  - $?X : \star \lhd \mathit{Nat} \to \mathit{Nat}$
  - $?Y : \star = \mathit{Nat}$

Example – High Level Goals

$$\begin{array}{rccl} \text{Assume} & \mathit{const} & : & \forall X\ Y.\, X \to Y \to X \\ \text{Assume} & \mathit{zero} & : & \mathit{Nat} \\ \text{Type} & \mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} & \Downarrow & \mathit{Nat} \to \mathit{Nat} \end{array}$$

- Check the spine $\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}$ against $\mathit{Nat} \to \mathit{Nat}$
- We fill in missing types to "elaborate" to $\mathit{const}\ \cdot (\mathit{Nat} \to \mathit{Nat})\ \cdot \mathit{Nat}\ (\lambda\, x : \mathit{Nat}.\, x)\ \mathit{zero}$
- X inferred contextually, Y synthetically
  In Cedille: $\boxed{\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}}$
  - $?X : \star\ \triangleleft\ \mathit{Nat} \to \mathit{Nat}$
  - $?Y : \star\ = \mathit{Nat}$
- And term argument $\lambda\, x.\, x$ *checked* against $\mathit{Nat} \to \mathit{Nat}$

Spine Local

$$\boxed{\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}} \Downarrow \mathit{Nat} \to \mathit{Nat}$$

- **Big idea:** locality for type-argument inference is *the spine*

Spine Local

$$\boxed{const\ (\lambda\, x.\, x)\ zero} \Downarrow Nat \to Nat$$

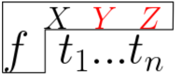
- **Big idea:** locality for type-argument inference is *the spine*
  - $\boxed{\text{cage meta-variables}}$ here
    never "escape" the spine or "descend" into the arguments

$$\boxed{f}\ \overset{X\quad Y\quad Z}{t_1 \ldots t_n}$$

Spine Local

$$\boxed{\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}} \Downarrow \color{red}{\mathit{Nat} \rightarrow \mathit{Nat}}$$

- **Big idea:** locality for type-argument inference is *the spine*
  - $\boxed{\text{cage meta-variables}}$ here
    never "escape" the spine or "descend" into the arguments
  - some meta-vars inferred contextually
    Using the expected result type



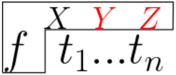
Spine Local

$$\boxed{\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}} \Downarrow \mathit{Nat} \to \mathit{Nat}$$

- **Big idea:** locality for type-argument inference is *the spine*
  - $\boxed{\text{cage meta-variables}}$ here

    never "escape" the spine or "descend" into the arguments
  - some meta-vars inferred contextually

    Using the expected result type
- **Consequence:** you know where to look when type-argument inference fails!



Example - Details

Assume	const	:	$\forall X\ Y.\, X \to Y \to X$
Assume	zero	:	Nat
Type	$\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}$	$\Downarrow$	$\mathit{Nat} \to \mathit{Nat}$

Example - Details

Assume	$const$	$:$	$\forall X\ Y. X \to Y \to X$
Assume	$zero$	$:$	Nat
Type	$const\ (\lambda x. x)\ zero$	$\Downarrow$	$Nat \to Nat$

Synthesize type for head

Example - Details

Assume	$const$	$:$	$\forall X\ Y.\, X \to Y \to X$
Assume	$zero$	$:$	Nat
Type	$const\ (\lambda\, x.\, x)\ zero$	$\Downarrow$	$Nat \to Nat$
Match	X	$\lhd_X$	$Nat \to Nat$

Match head return with expected return type

Example - Details

$$
\begin{array}{llcl}
\text{Assume} & \mathit{const} & : & \forall X\ Y.\, X \to Y \to X \\
\text{Assume} & \mathit{zero} & : & \mathit{Nat} \\
\text{Type} & \mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} & \Downarrow & \mathit{Nat} \to \mathit{Nat}\ \ = [\mathit{Nat} \to \mathit{Nat}/X]X \\
\text{Match} & X & \lhd_X & \mathit{Nat} \to \mathit{Nat}
\end{array}
$$

Get a contextual *instantiation*

Example - Details

$$
\begin{array}{cccl}
\text{Assume} & \mathit{const} & : & \forall X\ Y.\, X \to Y \to X \\
\text{Assume} & \mathit{zero} & : & \mathit{Nat} \\
\text{Type} & \mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} & \Downarrow & \mathit{Nat} \to \mathit{Nat}\ = [\mathit{Nat} \to \mathit{Nat}/X]X \\
\text{Match} & X & \lhd_X & \mathit{Nat} \to \mathit{Nat} \\
\text{Type} & \lambda\, x.\, x & &
\end{array}
$$

Type first argument

Example - Details

Assume	const	$:$	$\forall X\ Y.\, X \to Y \to X$
Assume	zero	$:$	Nat
Type	$\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero}$	$\Downarrow$	$\mathit{Nat} \to \mathit{Nat}\ = [\mathit{Nat} \to \mathit{Nat}/X]X$
Match	X	$\lhd_X$	$\mathit{Nat} \to \mathit{Nat}$
Type	$\lambda\, x.\, x$	$\Downarrow$	$[\mathit{Nat} \to \mathit{Nat}/X]X$

Type first argument in *checking mode*

Example - Details

$$\begin{array}{lccl}
\text{Assume} & \mathit{const} & : & \forall X\ Y.\, X \to Y \to X \\
\text{Assume} & \mathit{zero} & : & \mathit{Nat} \\
\text{Type} & \mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} & \Downarrow & \mathit{Nat} \to \mathit{Nat}\ = [\mathit{Nat} \to \mathit{Nat}/X]X \\
\text{Match} & X & \lhd_X & \mathit{Nat} \to \mathit{Nat} \\
\text{Type} & \lambda\, x.\, x & \Downarrow & [\mathit{Nat} \to \mathit{Nat}/X]X \\
\text{Type} & \mathit{zero} & &
\end{array}$$

Type second argument

Example - Details

Assume	$const$	:	$\forall X\ Y.\, X \to Y \to X$
Assume	$zero$	:	Nat
Type	$const\ (\lambda\, x.\, x)\ zero$	$\Downarrow$	$Nat \to Nat\ = [Nat \to Nat/X]X$
Match	X	$\lhd_X$	$Nat \to Nat$
Type	$\lambda\, x.\, x$	$\Downarrow$	$[Nat \to Nat/X]X$
Type	$zero$	$\Uparrow$	$[Nat/Y]Y$

Type second argument in *synthesis mode*

Example - Details

Assume	$const$	$:$	$\forall X\ Y. X \to Y \to X$
Assume	$zero$	$:$	Nat
Type	$const\ (\lambda x. x)\ zero$	$\Downarrow$	$Nat \to Nat\ = [Nat \to Nat/X]X$
Match	X	$\lhd_X$	$Nat \to Nat$
Type	$\lambda x. x$	$\Downarrow$	$[Nat \to Nat/X]X$
Type	$zero$	$\Uparrow$	$[Nat/Y]Y$

Conclude the spine has the expected type!

Outline

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda x.\, t$, $\Lambda X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda x.\, t$, $\Lambda X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize
- **But:** when is a term's type checked? In a spine?

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda\, x.\, t$, $\Lambda\, X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize
- **But:** when is a term's type checked? In a spine?

$$f \quad t_1 \;\; t_2 \;\; t_3 \Downarrow T$$

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda\, x.\, t$, $\Lambda\, X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize
- **But:** when is a term's type checked? In a spine?
  - when the checked type of the spine

$$f \quad t_1 \;\; t_2 \;\; t_3 \Downarrow T$$

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda\, x.\, t$, $\Lambda\, X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize
- **But:** when is a term's type checked? In a spine?
  - when the checked type of the spine
  - and the synthesized type of the head

$$f^{\Uparrow}\ t_1\ \ t_2\ \ t_3 \Downarrow T$$

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda x.\, t$, $\Lambda X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize
- **But:** when is a term's type checked? In a spine?
  - when the checked type of the spine
  - and the synthesized type of the head
  - and any synthesized types from earlier args

$$f^{\Uparrow}\ t_1^{\Uparrow}\ t_2\ \ t_3 \Downarrow T$$

Annotation Requirements: Abstractions (1/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Term and type abstractions* $\lambda x.\, t$, $\Lambda X.\, t$
  Bare abstractions can only be checked; require type / kind annotations to synthesize
- **But:** when is a term's type checked? In a spine?
  - when the checked type of the spine
  - and the synthesized type of the head
  - and any synthesized types from earlier args
  - tells us the *complete* type to check

$$f^{\Uparrow}\ t_1^{\Uparrow}\ t_2^{\Downarrow}\ t_3 \Downarrow T$$

Annotation Requirements: Type Arguments (2/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type arguments* (FO only)
  Must provide type args explicitly when not informed by

Annotation Requirements: Type Arguments (2/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type arguments* (FO only)
  Must provide type args explicitly when not informed by
  - Checked type for spine: $const\ (\lambda\, x.\, x)\ zero \Downarrow [Nat \to Nat/X]X$

Annotation Requirements: Type Arguments (2/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type arguments* (FO only)
  Must provide type args explicitly when not informed by
  - Checked type for spine: $\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} \Downarrow [\mathit{Nat} \to \mathit{Nat}/X]X$
  - Synthesized type for args: $\mathit{zero} \Uparrow [\mathit{Nat}/Y]Y$

Annotation Requirements: Type Arguments (2/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type arguments* (FO only)
  Must provide type args explicitly when not informed by
  - Checked type for spine: $\mathit{const}\ (\lambda\, x.\, x)\ \mathit{zero} \Downarrow [\mathit{Nat} \to \mathit{Nat}/X]X$
  - Synthesized type for args: $\mathit{zero} \Uparrow [\mathit{Nat}/Y]Y$
- **Example:** $\forall\, Y\ X.\, X \to X$
  Y doesn't occur in result or arg position
  $\Longrightarrow$ must be instantiated explicitly.

Annotation Requirements: Applicability (3/3)

Saw how type inference *works*. Now – where does it *need help?*

Annotation Requirements: Applicability (3/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type of a function* in a term or type application must reveal resp. a type quantifier or arrow

$$\begin{array}{ccc} f \ \cdot T & \Longrightarrow & f : \forall X.\, S \\ f \ t & \Longrightarrow & f : \forall \overline{X}.\, S \rightarrow T \end{array}$$

Annotation Requirements: Applicability (3/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type of a function* in a term or type application must reveal resp. a type quantifier or arrow

$$\begin{array}{ccc} f \ \cdot T & \Longrightarrow & f : \forall\, X.\, S \\ f\ t & \Longrightarrow & f : \forall\, \overline{X}.\, S \to T \end{array}$$

- **Example:** assume *absurd* $: \forall\, X : \star.\, X$

Annotation Requirements: Applicability (3/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type of a function* in a term or type application must reveal resp. a type quantifier or arrow

$$\begin{array}{ccc} f \ \cdot T & \Longrightarrow & f : \forall X.\, S \\ f \ t & \Longrightarrow & f : \forall \overline{X}.\, S \rightarrow T \end{array}$$

- **Example:** assume $absurd : \forall X{:}\star.\, X$
  - Will **not** type $absurd\ zero$
  - Will type $absurd \cdot (Nat \rightarrow Nat)\ zero$
    Or even $absurd \cdot (\forall X.\, X \rightarrow X)\ zero$

Annotation Requirements: Applicability (3/3)

Saw how type inference *works*. Now – where does it *need help?*

- *Type of a function* in a term or type application must reveal resp. a type quantifier or arrow

$$\begin{array}{ccc} f\ \ \cdot T & \Longrightarrow & f : \forall\, X.\, S \\ f\ \ t & \Longrightarrow & f : \forall\, \overline{X}.\, S \rightarrow T \end{array}$$

- **Example:** assume $absurd : \forall\, X{:}\star.\, X$
  - Will **not** type $absurd\ \ zero$
  - Will type $absurd \cdot (Nat \rightarrow Nat)\ zero$
    Or even $absurd \cdot (\forall\, X.\, X \rightarrow X)\ zero$
- **Advantage:** meta-variables are *one-to-one* with quantified type variables in functions!
  It's always easy to understand why a meta-var was introduced!

Thanks!

- `language-overview/type-inference.ced`
  For practical examples
- Paper can be found on arXiv
  (To appear in proceedings of IFL 2018)