Spine-local Type Inference

Christopher Jenkins and Aaron Stump

Computer Science
University of Iowa

IFL ’18
Outline

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

2 The Specificational System
   - Terms and Terminology
   - Type Inference

3 Discussion
   - Specificational System Properties
   - Algorithmic System Properties
   - Future Work
Outline

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

2. The Specificational System
   - Terms and Terminology
   - Type Inference

3. Discussion
   - Specificational System Properties
   - Algorithmic System Properties
   - Future Work
What is “Local Type Inference”?

- Introduced by Pierce and Turner in ’98
- Extended by Odersky et al. in ’01
- Uses two main techniques
  - Bidirectional typing rules:

  ▶  Local type-argument inference:
What is “Local Type Inference”?

- Introduced by Pierce and Turner in '98
- Extended by Odersky et al. in '01
- Uses two main techniques
  - **Bidirectional typing rules:**
    - Synthesis mode: $\lambda x : \text{Nat}. x \uparrow \text{Nat} \rightarrow \text{Nat}$
    - Checking mode: $\lambda x. x \downarrow \text{Nat} \rightarrow \text{Nat}$
  - **Local type-argument inference:**
What is “Local Type Inference”?

- Introduced by Pierce and Turner in ’98
- Extended by Odersky et al. in ’01
- Uses two main techniques
  - **Bidirectional typing rules:**
    - Synthesis mode: \( \lambda x : \text{Nat} . \ x \uparrow \text{Nat} \to \text{Nat} \)
    - Checking mode: \( \lambda x . \ x \downarrow \text{Nat} \to \text{Nat} \)
  - **Local type-argument inference:**
    - Let \( id : \forall X . X \to X \)
    - Type \( id \ 0 \uparrow \text{Nat} \)
    - Infer \( X = \text{Nat} \) from \( 0 \)

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 and beyond
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 and beyond
- 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 and beyond

- 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
Limitations

Local type inference in its published form can sometimes still require “silly” type annotations, i.e. those for which there should be enough contextual information to omit

Let \( \textit{pair} : \forall X, Y. X \rightarrow Y \rightarrow X \times Y \)

Type \( \textit{pair} \; (\lambda \ x. \ x) \; 0 \)
Limitations

Local type inference in its published form can sometimes still require “silly” type annotations, i.e. those for which there should be enough contextual information to omit.

Let \( \textit{pair} : \forall X, Y. X \to Y \to X \times Y \)

Type \( \textit{pair} \ (\lambda x. x) \ 0 \)

We do not expect to locally \textit{synthesize} a type.
Limitations

Local type inference in its published form can sometimes still require “silly” type annotations, i.e. those for which there should be enough contextual information to omit.

Let \[ \text{pair} : \forall X, Y. X \rightarrow Y \rightarrow X \times Y \]
Type \[ \text{pair} (\lambda x. x) 0 \uparrow ??? \]

- We do not expect to locally \textit{synthesize} a type.
Limitations

Local type inference in its published form can sometimes still require “silly” type annotations, i.e. those for which there should be enough contextual information to omit.

Let \( \text{pair} : \forall X, Y. X \rightarrow Y \rightarrow X \times Y \)

<table>
<thead>
<tr>
<th>Type</th>
<th>( \text{pair} (\lambda x. x) 0 )</th>
<th>( \rightarrow )</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \text{pair} (\lambda x. x) 0 )</td>
<td>( \downarrow )</td>
<td>( \text{Nat} \rightarrow \text{Nat} \times \text{Nat} )</td>
</tr>
</tbody>
</table>

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

Local type inference in its published form can sometimes still require “silly” type annotations, i.e. those for which there should be enough contextual information to omit

Let \( \text{pair} : \forall X, Y. X \to Y \to X \times Y \)

<table>
<thead>
<tr>
<th>Type</th>
<th>( \text{pair} (\lambda x. x) 0 )</th>
<th>( \uparrow )</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \text{pair} (\lambda x. x) 0 )</td>
<td>( \downarrow )</td>
<td>( \text{Nat} \to \text{Nat} \times \text{Nat} )</td>
</tr>
</tbody>
</table>

- 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

Local type inference in its published form can sometimes still require “silly” type annotations, i.e. those for which there should be enough contextual information to omit.

Let $\text{pair} : \forall X, Y. X \rightarrow Y \rightarrow X \times Y$

<table>
<thead>
<tr>
<th>Type</th>
<th>$\text{pair} \ (\lambda x. x) \ 0$</th>
<th>$\uparrow$</th>
<th>$???$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\text{pair} \ (\lambda x. x) \ 0$</td>
<td>$\downarrow$</td>
<td>$\text{Nat} \rightarrow \text{Nat} \times \text{Nat}$</td>
</tr>
</tbody>
</table>

- We do not expect to locally {f synthesize} a type
- ... but we would expect to {f check} it against a type
  - We could call this “contextual” type-argument inference.
- Unfortunately, this is not done in the two major published systems
  - Popular “unofficial” extension (used in e.g. Scala, Rust)
Limitations (cont.)

- Usually uses “fully-uncurried” function applications

\[ f(t_1, \ldots, t_n) \]

- Maximize available info at a single application
Limitations (cont.)

- Usually uses “fully-uncurried” function applications

\[ f(t_1, \ldots, t_n) \]

- Maximize available info at a single application

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

\[ f[T_1, \ldots, T_m](t_1, \ldots, t_n) \]
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
  
  \[
  f \uparrow t_1 \uparrow t_2 \downarrow t_3
  \]

- 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

\[
f \overset{t_1}{\uparrow} \overset{t_2}{\downarrow} t_3 \downarrow T
\]

- Precise, specificationnal 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

\[
f \uparrow t_1 \downarrow t_2 \downarrow t_3 \downarrow T
\]

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

\[
f[S, T, V](t_1, t_2, t_3)
\]
Our Contributions

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

\[
\begin{align*}
  f & \quad t_1 \uparrow \quad t_2 \downarrow \quad t_3 \downarrow \quad T \\
\end{align*}
\]

- Precise, specificational account of this technique

- Better support function currying and partial type applications by being “spine-local.”

\[
\begin{align*}
  f[S][T][V] & \quad t_1 \quad t_2 \quad t_3 \\
\end{align*}
\]
Our Contributions

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

\[
\begin{array}{c}
f \uparrow t_1 \downarrow t_2 \downarrow t_3 \downarrow T \\
\end{array}
\]

- Precise, specificational account of this technique

- Better support function currying and partial type applications by being “spine-local.”

\[
\begin{array}{c}
f[S] \quad t_1 \quad t_2 \\
\end{array}
\]
Our type system(s)

- Two type systems: one *specificalional* and one *algorithmic*
- Spec. system abstracts contextual type-argument inference
  - Non-deterministic
- Sanity checks for spec. system, annotation requirements
- Equivalence of the two systems
Outline

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

2 The Specificational System
   - Terms and Terminology
   - Type Inference

3 Discussion
   - Specificational System Properties
   - Algorithmic System Properties
   - Future Work
Our Setting

- The setting for our type inference system is (impredicative) System F
- Internal and external term languages:
  - *internal*: all type annotations and arguments are provided
  - *external*: some of these can be elided
- Type inference viewed as relation between these two languages
  - *Elaborate* external $\rightsquigarrow$ internal terms
## Language Syntax

**Types**  \[ S, T, U, V ::= X, Y, Z | S \to T | \forall X.\ T | S \times T \]

**Contexts**  \[ \Gamma ::= \cdot | \Gamma, X | \Gamma, x: T \]

**Internal Terms**  \[ e, p ::= x | \lambda x: T.\ e | \Lambda X.\ e | e\ e' | e[T] \]

**External Terms**  \[ t ::= x | \lambda x: T.\ t | \Lambda X.\ t | t\ t' | t[T] \]
Language Syntax

Types

$S, T, U, V ::= X, Y, Z \mid S \rightarrow T \mid \forall X. T \mid S \times T$

Contexts

$\Gamma ::= \cdot \mid \Gamma, X \mid \Gamma, x : T$

Internal Terms

$e, p ::= x \mid \lambda x : T. e \mid \Lambda X. e \mid e \; e' \mid e[T]$

External Terms

$t ::= x \mid \lambda x : T. t \mid \Lambda X. t \mid t \; t' \mid t[T]$

- Pair types for illustration
Language Syntax

**Types**  \( S, T, U, V ::= X, Y, Z \mid S \to T \mid \forall X. T \mid S \times T \)

**Contexts**  \( \Gamma ::= \cdot \mid \Gamma, X \mid \Gamma, x : T \)

**Internal Terms**  \( e, p ::= x \mid \lambda x : T. e \mid \Lambda X. e \mid e \, e' \mid e[T] \)

**External Terms**  \( t ::= x \mid \lambda x : T. t \mid \Lambda X. t \mid t \, t' \mid t[T] \)

\[ | \lambda x. t \]

- Pair types for illustration
- Will try to infer binder annotations and type arguments in external language
Language Syntax

Types \( S, T, U, V ::= X, Y, Z \mid S \rightarrow T \mid \forall X. T \mid S \times T \)

Contexts \( \Gamma ::= \cdot \mid \Gamma, X \mid \Gamma, x : T \)

Internal Terms \( e, p ::= x \mid \lambda x : T. e \mid \Lambda X. e \mid e e' \mid e[T] \)

External Terms \( t ::= x \mid \lambda x : T. t \mid \Lambda X. t \mid t t' \mid t[T] \)

\[ \mid \lambda x. t \]

- Pair types for illustration
- Will try to infer binder annotations and type arguments in external language
Terminology

- **Application head**: variable or abstraction

\[ x, \  \Lambda X. t, \ \lambda x. t \]
Terminology

- **Application head**: variable or abstraction
  \[ x, \ \Lambda X. t, \ \lambda x. t \]

- **Application spine**: head followed by seq. of term, type arguments
  \[ x \ t_1 \ t_2 \ t_3 \text{ vs } (((x \ t_1) \ t_2) \ t_3) \]
Terminology

- **Application head**: variable or abstraction
  
  \[ x, \\ \Lambda X. t, \ \lambda x. t \]

- **Application spine**: head followed by seq. of term, type arguments
  
  \[ x t_1 t_2 t_3 \text{ vs } (((x t_1) t_2) t_3) \]

- **Applicand**: Term in the function position of an application
  
  \[ t_1 \text{ in } t_1 t_2 \]
Terminology

- **Application head**: variable or abstraction
  \[ x, \ \Lambda X. t, \ \lambda x. t \]

- **Application spine**: head followed by seq. of term, type arguments
  
  \[ \begin{array}{c|ccc} \times & t_1 & t_2 & t_3 \end{array} \quad \text{vs} \quad (((x \ t_1) \ t_2) \ t_3) \]

- **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
  
  \[ \begin{array}{c|ccc} \times & t_1 & t_2 & t_3 \end{array} \]
  
  Not \ max \quad \begin{array}{c|ccc} \times & t_1 & t_2 & t_3 \end{array}
  
  Max \quad \begin{array}{c|ccc} \times & t_1 & t_2 & t_3 \end{array} \]
Example from the intro: \( \Gamma \vdash \downarrow \text{pair} \ (\lambda x. x) \ 0 : (\text{Nat} \rightarrow \text{Nat}) \times \text{Nat} \)

- “Under context \( \Gamma \), the expression checks against the given type”
  (Where \text{pair} and 0 are suitably defined)
Example – High Level Goals

Example from the intro: $\Gamma \vdash \Downarrow pair (\lambda x. x) 0 : (Nat \rightarrow Nat) \times Nat$

- “Under context $\Gamma$, the expression checks against the given type”
  (Where $pair$ and 0 are suitably defined)
- System will elaborate to $pair[Nat \rightarrow Nat][Nat] (\lambda x : Nat. x) 0$
  For illustration, example shows synthetic and contextual type-arg. inference
Example – High Level Goals

Example from the intro: \( \Gamma \vdash \downarrow \ pair \ (\lambda x. x) \ 0 : (Nat \rightarrow Nat) \times Nat \)

- “Under context \( \Gamma \), the expression checks against the given type” (Where \( pair \) and 0 are suitably defined)
- System will elaborate to \( pair[Nat \rightarrow Nat][Nat] \ (\lambda x: Nat. x) \ 0 \)

For illustration, example shows synthetic and contextual type-arg. inference
Example – High Level Goals

Example from the intro: $\Gamma \vdash \Downarrow pair (\lambda x. x) 0 : (Nat \to Nat) \times Nat$

- “Under context $\Gamma$, the expression checks against the given type”
  (Where $pair$ and 0 are suitably defined)
- System will elaborate to $pair[Nat \to Nat][Nat] (\lambda x: Nat. x) 0$
  For illustration, example shows synthetic and contextual type-arg. inference
- ... however, elaboration clutters the rules, so omitted for the example
Spine Judgment

$$\Gamma \vdash^P t : T \rightsquigarrow \sigma$$

- “Spine $t$ partially synthesizes type $T$ with contextual type-args. $\sigma$”
- Big idea: enforce locality, contextuality at maximal applications
Spine Judgment

\[ \Gamma \vdash^P t : T \leadsto \sigma \]

- “Spine \( t \) partially synthesizes type \( T \) with contextual type-args. \( \sigma \)”
- Big idea: enforce locality, contextuality at maximal applications
  - cage meta-variables to just the spine with spine judgment (locality)
Spine Judgment

\[ \Gamma \vdash^P t : T \sim \sigma \]

- "Spine $t$ partially synthesizes type $T$ with contextual type-args. $\sigma$"
- Big idea: enforce locality, contextuality at maximal applications
  - cage meta-variables to just the spine with spine judgment (locality)
  - require meta-variable "guesses" justified contextuality

\[ f \vdash X \ Y \ Z \ t_1 \ldots t_n \]
Spine Judgment (Ex.)

\[ \Gamma \vdash^P \text{pair} (\lambda x. x) 0 : X \times \text{Nat} \leadsto [\text{Nat} \to \text{Nat}/X] \]
Spine Judgment (Ex.)

\[ \Gamma \vdash^P \textit{pair} (\lambda x. x) 0 : X \times \textit{Nat} \leadsto [\textit{Nat} \rightarrow \textit{Nat}/X] \]

Base case: synthesize type for head

\[ \Gamma \vdash^\uparrow \textit{pair} : \forall X, Y. X \rightarrow Y \rightarrow X \times Y \]
Spine Judgment (Ex.)

\[
\Gamma \vdash^P pair (\lambda x. x) 0 : X \times \text{Nat} \rightsquigarrow [\text{Nat} \rightarrow \text{Nat}/X]
\]

Begin walking up spine

\[
\Gamma \vdash^P pair : \forall X, Y. X \rightarrow Y \rightarrow X \times Y \rightsquigarrow \sigma_{id} \quad (\sigma_{id} \text{ is identity subst.})
\]
Spine Judgment (Ex.)

\[ \Gamma \vdash^P \text{pair} \ (\lambda x. x) \ 0 : X \times Nat \simto [Nat \to Nat / X] \]

Encounter term app. with missing type arg.

\[ \Gamma \vdash^P \text{pair} : \forall X, Y. X \to Y \to X \times Y \simto \sigma_{id} \ (\sigma_{id} \text{ is identity subst.}) \]
Spine Judgment (Ex.)

\[ \Gamma \vdash^P \text{pair} \ (\lambda x. x) \ 0 : X \times \text{Nat} \rightsquigarrow [\text{Nat} \to \text{Nat}/X] \]

Defer to last judgment form: *application* judgment

\[ \Gamma \vdash \ (\forall X, Y. X \to Y \to X \times Y, \sigma_{id}) \cdot (\lambda x. x) : Y \to X \times Y \rightsquigarrow [\text{Nat} \to \text{Nat}/X] \]
Application Judgment

\[ \Gamma \vdash (T, \sigma) \cdot t : T' \rightsquigarrow \sigma' \]

- “An applicand of type $T$ with ctxt. solutions $\sigma$ can be applied to argument $t$, producing result type $T'$ and result ctxt. solutions $\sigma'$”
- Infer missing type-args in term apps., synthetically and contextually
- Type application when arrow revealed
Application Judgment

\[ \Gamma \vdash (T, \sigma) \cdot t : T' \sim \sigma' \]

- “An applicand of type \( T \) with ctxt. solutions \( \sigma \) can be applied to argument \( t \), producing result type \( T' \) and result ctxt. solutions \( \sigma' \)”
- Infer missing type-args in term apps., synthetically and contextually
  - the whether and what of contextual inference is non-deterministic
- Type application when arrow revealed
Application Judgment (Ctx.)

\[ \Gamma \vdash (\forall X, Y. X \rightarrow Y \rightarrow X \times Y, \sigma_{id}) \cdot (\lambda x. x) : Y \rightarrow X \times Y \leadsto [Nat \rightarrow Nat/X] \]
Application Judgment (Ctx.)

\[ \Gamma \vdash (\forall X, Y. X \rightarrow Y \rightarrow X \times Y, \sigma_{id}) \cdot (\lambda x. x) : Y \rightarrow X \times Y \rightsquigarrow [\text{Nat} \rightarrow \text{Nat}/X] \]

Make a contextual guess for \( X \), \( \text{Nat} \rightarrow \text{Nat} \)
Application Judgment (Ctx.)

\[ \Gamma \vdash (\forall Y. X \rightarrow Y \rightarrow X \times Y, [Nat \rightarrow Nat/X]) \cdot (\lambda x. x) : Y \rightarrow X \times Y \leadsto [Nat \rightarrow Nat/X] \]
Application Judgment (Ctx.)

\[ \Gamma \vdash (\forall Y. X \rightarrow Y \rightarrow X \times Y, [\text{Nat} \rightarrow \text{Nat}/X]) \cdot (\lambda x. x) : Y \rightarrow X \times Y \rightsquigarrow [\text{Nat} \rightarrow \text{Nat}/X] \]

Non-deterministically choose to instantiate \( Y \) synthetically
Application Judgment (Ctx.)

\[
\Gamma \vdash (X \to Y \to X \times Y, [Nat \to Nat/X]) \cdot (\lambda x. x) : Y \to X \times Y \leadsto [Nat \to Nat/X]
\]
Application Judgment (Ctx.)

\[ \Gamma \vdash (X \to Y 	o X \times Y, [Nat \to Nat/X]) \cdot (\lambda x. x) : Y \to X \times Y \leadsto [Nat \to Nat/X] \]

Reveal an arrow in applicand type
Two cases arise when we reveal an arrow.
Application Judgment (Arrow)

Two cases arise when we reveal an arrow.

- Expected type of arg. is *fully known* (from spine head, contextual type, previous arguments)
  Use checking mode for arg.
Two cases arise when we reveal an arrow.

- Expected type of arg. is *fully known* (from spine head, contextual type, previous arguments)
  Use checking mode for arg.

- Expected type has unsolved meta-vars
  Use synthesis mode for arg. to learn instantiations
Application Judgment (Ctx)

\[ \Gamma \vdash (X \rightarrow Y \rightarrow X \times Y, [Nat \rightarrow Nat/X]) \cdot (\lambda x. x) : Y \rightarrow X \times Y \rightsquigarrow [Nat \rightarrow Nat/X] \]
Application Judgment (Ctx)

\[\Gamma \vdash (X \rightarrow Y \rightarrow X \times Y, [\text{Nat} \rightarrow \text{Nat}/X]) \cdot (\lambda x. x) : Y \rightarrow X \times Y \leadsto [\text{Nat} \rightarrow \text{Nat}/X]\]

Type is fully known: \(\Gamma \Downarrow \lambda x. x : \text{Nat} \rightarrow \text{Nat}\)
Application Judgment (Ctx)

\[ \Gamma \vdash (X \rightarrow Y \rightarrow X \times Y, [Nat \rightarrow Nat/X]) \cdot (\lambda x. x) : Y \rightarrow X \times Y \rightsquigarrow [Nat \rightarrow Nat/X] \]

Produced result type of the app, with ctxt. solution
Application Judgment (Syn)

- Last part of the spine judgment is typing \( \text{pair} \ (\lambda x. x) \) to 0
- We defer again to application judgment
- \( Y \) will be inferred synthetically from 0
Application Judgment (Syn)

\[
\Gamma \vdash (Y \to X \times Y, [Nat \to Nat/X]) \cdot 0 : X \times Nat \leadsto [Nat \to Nat/X]
\]
Application Judgment (Syn)

\[ \Gamma \vdash (Y \to X \times Y, [Nat \to Nat/X]) \cdot 0 : X \times Nat \leadsto [Nat \to Nat/X] \]

Arrow revealed
Application Judgment (Syn)

\[ \Gamma \vdash (\mathcal{Y} \rightarrow \mathcal{X} \times \mathcal{Y}, [\text{Nat} \rightarrow \text{Nat}/\mathcal{X}]) \odot 0 : \mathcal{X} \times \text{Nat} \leadsto [\text{Nat} \rightarrow \text{Nat}/\mathcal{X}] \]

Incomplete info. for expected arg. type \( \mathcal{Y} \)
Application Judgment (Syn)

\[
\Gamma \vdash (Y \to X \times Y, [\text{Nat} \to \text{Nat}/X]) \cdot 0 : X \times \text{Nat} \rightsquigarrow [\text{Nat} \to \text{Nat}/X]
\]

Synthesize type for arg. (note \(Y\) not passed down!)

\[
\Gamma \vdash \uparrow 0 : \text{Nat}
\]
Application Judgment (Syn)

\[
\Gamma \vdash (Y \rightarrow X \times Y, [Nat \rightarrow Nat/X]) \cdot 0 : X \times Nat \leadsto [Nat \rightarrow Nat/X]
\]

Must match expectation \( Y \), provide instantiation \([Nat/Y]\)

\[
\Gamma \vdash 0 : [Nat/Y]Y
\]
Application Judgment (Syn)


Use syn. type-arg in result type of app
Enforcement at Maximal Application

\[ \Gamma \vdash^p \text{pair} \ (\lambda x. x) \ 0 : X \times \text{Nat} \sim \{\text{Nat} \to \text{Nat} / X\} \]

Earlier I said “enforce locality, contextuality…” how?
Enforcement at Maximal Application

\[ \Gamma \vdash^P pair (\lambda x. x) \ 0 : X \times \text{Nat} \rightsquigarrow [\text{Nat} \to \text{Nat}/X] \]

\[ \text{dom}([\text{Nat} \to \text{Nat}/X]) = X = \text{MV}(\Gamma, X \times \text{Nat}) \]

Earlier I said “enforce locality, contextuality...” how?

- All remaining meta-variables are solved by \( \sigma \)

\[ \text{MV}(\Gamma, T) : \text{meta-vars of } T \ \text{wrt declared variables of } \Gamma \]
Enforcement at Maximal Application

\[
\Gamma \vdash^P pair (\lambda x. x) 0 : X \times \text{Nat} \rightsquigarrow [\text{Nat} \rightarrow \text{Nat}/X]
\]

\[
dom([\text{Nat} \rightarrow \text{Nat}/X]) = X = MV(\Gamma, X \times \text{Nat})
\]

\[
[Nat \rightarrow Nat/X] (X \times \text{Nat}) = (Nat \rightarrow Nat) \times Nat
\]

Earlier I said “enforce locality, contextuality...” how?

- All remaining meta-variables are solved by \(\sigma\)
  \(MV(\Gamma, T)\): meta-vars of \(T\) wrt declared variables of \(\Gamma\)
- Contextual solutions \textit{really are} contextual
Enforcement at Maximal Application

\[ \Gamma \vdash^P \text{pair } (\lambda x. x) \ 0 : X \times \text{Nat} \rightsquigarrow [\text{Nat} \rightarrow \text{Nat}/X] \]

\[ \text{dom}([\text{Nat} \rightarrow \text{Nat}/X]) = X = MV(\Gamma, X \times \text{Nat}) \]

\[ [\text{Nat} \rightarrow \text{Nat}/X] (X \times \text{Nat}) = (\text{Nat} \rightarrow \text{Nat}) \times \text{Nat} \]

\[ \Gamma \vdash_\downarrow \text{pair } (\lambda x. x) \ 0 : (\text{Nat} \rightarrow \text{Nat}) \times \text{Nat} \]

Earlier I said “enforce locality, contextuality...” how?

- All remaining meta-variables are solved by \( \sigma \)
  \( MV(\Gamma, T) \): meta-vars of \( T \) wrt declared variables of \( \Gamma \)
- Contextual solutions \textit{really are} contextual
- We clear these conditions and can type the expression
Outline

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

2 The Specificational System
   - Terms and Terminology
   - Type Inference

3 Discussion
   - Specificational System Properties
   - Algorithmic System Properties
   - Future Work
Specificational System Properties

Sanity check wrt. internal language (System F; $\Gamma \vdash t : T$)
Specificational System Properties

Sanity check wrt. internal language (System F; $\Gamma \vdash t : T$)

- **Soundness:**
  \[ \Gamma \vdash_\delta t : T \leadsto e \implies \Gamma \vdash e : T \]

- **Trivial completeness:**
  \[ \Gamma \vdash e : T \implies \Gamma \vdash_\uparrow e : T \leadsto e \]
Specificational System Properties (cont.)

- Typeability of the *external* language (i.e. type annotation requirements)
- Assume $\Gamma \vdash e : T$. Erase binder, type args to get external term $t$.
- $\Gamma \vdash \uparrow t : T \leadsto e$ when given
Specificational System Properties (cont.)

- Typeability of the external language (i.e. type annotation requirements)

- Assume $\Gamma \vdash e : T$. Erase binder, type args to get external term $t$.

- $\Gamma \vdash \uparrow t : T \leadsto e$ when given
  - Binder annotations to $\lambda$s when its context or spine-context lack this info
  - Instantiations for “phantom” type-arguments
    $\forall X, Y. X \rightarrow X$
  - Enough info to “see” a term or type application
    e.g. applicand of type $X$ given $[S]$ or $t$
Algorithmic system

- “Prototypes” track expected result type, num args to spine head

\[ ? \to ? \to \text{Nat} \]
Algorithmic system

- “Prototypes” track expected result type, num args to spine head

\[ ? \rightarrow ? \rightarrow \text{Nat} \]

- Matched against head type, produces a “decorated” function type

\[ \forall X = \text{Nat}. \forall Y = Y. X \rightarrow Y \rightarrow X \]
Prototype Matching (Ex. 1)

Check \( \text{pair} (\lambda x. x) 0 \) against \((\text{Nat} \to \text{Nat}) \times \text{Nat}\)
Prototype Matching (Ex. 1)

Check \(\text{pair } (\lambda x. x) 0\) against \((\text{Nat} \rightarrow \text{Nat}) \times \text{Nat}\)

Prototype: \(\_ \rightarrow \_ \rightarrow (\text{Nat} \rightarrow \text{Nat}) \times \text{Nat}\)

Head type: \(\forall X. \forall Y. X \rightarrow Y \rightarrow X \times Y\)
Prototype Matching (Ex. 1)

Check \( \text{pair } (\lambda x. x) \ 0 \) against \((\text{Nat} \rightarrow \text{Nat}) \times \text{Nat}\)

Prototype: \( ? \rightarrow ? \rightarrow (\text{Nat} \rightarrow \text{Nat}) \times \text{Nat} \)

Head type: \( \forall X. \forall Y. X \rightarrow Y \rightarrow X \times Y \)

Decoration: \( \forall X = \text{Nat} \rightarrow \text{Nat}. \forall Y = \text{Nat}. X \rightarrow Y \rightarrow X \times Y \)
Prototype Matching (Ex. 1)

Check \( \text{pair} (\lambda x. x) 0 \) against \((\text{Nat} \rightarrow \text{Nat}) \times \text{Nat}\)

Prototype: \( \rightarrow \rightarrow (\text{Nat} \rightarrow \text{Nat}) \times \text{Nat} \)

Head type: \( \forall X. \forall Y. X \rightarrow Y \rightarrow X \times Y \)

Decoration: \( \forall X = \text{Nat} \rightarrow \text{Nat}. \forall Y = \text{Nat}. X \rightarrow Y \rightarrow X \times Y \)

No “guessing” for contextual type-args.
Prototype Matching (Ex. 2)

Careful handling needed when prototype arity exceeds the spine head’s

Check \textit{id suc 0} against type \textit{Nat}
Prototype Matching (Ex. 2)

Careful handling needed when prototype arity exceeds the spine head’s

Check \textit{id suc 0} against type \textit{Nat}

Prototype: \(? \rightarrow \ ? \rightarrow \textit{Nat}\)
Head type: \(\forall X . X \rightarrow X\)
Prototype Matching (Ex. 2)

Careful handling needed when prototype arity exceeds the spine head’s

Check *id suc 0* against type *Nat*

Prototype: \( ? \to ? \to \text{Nat} \)

Head type: \( \forall X \ . \ X \to X \)

Decoration: \( \forall X = X \ . \ X \to (X, \ ? \to \text{Nat}) \)
Prototype Matching (Ex. 2)

Careful handling needed when prototype arity exceeds the spine head’s

Check $id \ suc \ 0$ against type $Nat$

Prototype: $? \rightarrow \ ? \rightarrow Nat$
Head type: $\forall X . X \rightarrow X$
Decoration: $\forall X \equiv X . X \rightarrow (X, \ ? \rightarrow Nat)$

- Don’t know how to instantiate $X$, save for later
Prototype Matching (Ex. 2)

Careful handling needed when prototype arity exceeds the spine head’s

Check \( \text{id } \text{suc } 0 \) against type \( \text{Nat} \)

Prototype: \( \text{? } \rightarrow \text{? } \rightarrow \text{Nat} \)

Head type: \( \forall X . X \rightarrow X \)

Decoration: \( \forall X = X . X \rightarrow (X, \text{? } \rightarrow \text{Nat}) \)

- Don’t know how to instantiate \( X \), save for later
- From \textit{synthesis} instantiate \( X \), then compare
  Match \( \text{Nat } \rightarrow \text{Nat} \) with \( \text{? } \rightarrow \text{Nat} \) once we reach first arg. \( \text{suc} \)
Algorithmic Systems Properties

\[ \Gamma \vdash_{\delta} t : T \rightsimeq e \]

- **Algorithmic:**
  The system is given as a set of syntax-directed inference rules

- **Equivalent to Specification:**
  - Soundness:
    \[ \Gamma \vdash_{\delta} t : T \rightsimeq e \implies \Gamma \vdash_{\delta} t : T \rightsimeq e \]
  - Completeness:
    \[ \Gamma \vdash_{\delta} t : T \rightsimeq e \implies \Gamma \vdash_{\delta} t : T \rightsimeq e \]

Even though we never mentioned prototype matching or "stuck" decorations in the spec!
Algorithmic Systems Properties

\[
\Gamma \vdash_\delta t : T \leadsto e
\]

- **Algorithmic:**
  The system is given as a set of syntax-directed inference rules

- **Equivalent to Specification:**
  - **Soundness:**
    \[
    \Gamma \vdash_\delta t : T \leadsto e \implies \Gamma \vdash_\delta t : T \leadsto e
    \]
  - **Completeness:**
    \[
    \Gamma \vdash_\delta t : T \leadsto e \implies \Gamma \vdash_\delta t : T \leadsto e
    \]

- ... even though we never mentioned prototype matching or “stuck” decorations in the spec!
Algorithmic System Properties

Spec. system

\[
\begin{align*}
\Gamma \vdash^P t : T & \rightsimeq \sigma \\
\Gamma \vdash (T, \sigma) \cdot t : T' & \rightsimeq \sigma'
\end{align*}
\]

Alg. system

\[
\begin{align*}
\Gamma; P \vdash t : W & \rightsimeq \sigma \\
\Gamma \vdash (W, \sigma) \cdot t : W' & \rightsimeq \sigma' \\
\overline{X} \vdash X \triangleq T \triangleq P \Rightarrow (W, \sigma)
\end{align*}
\]
Algorithmic System Properties

\[ \Gamma \vdash P : T \leadsto \sigma \]
\[ \Gamma \vdash (T, \sigma) \cdot t : T' \leadsto \sigma' \]

Spec. system

\[ \equiv \]

\[ \Gamma; P \vdash ? t : W \leadsto \sigma \]
\[ \Gamma \vdash (W, \sigma) \cdot t : W' \leadsto \sigma' \]
\[ \overline{X} \vdash := T := P \Rightarrow (W, \sigma) \]

Alg. system
Type inference algorithm is implemented in Cedille, a language with impredicativity, dependent types, and dependent intersections. A local type inference system will be a good foundation for considering the following extensions:
Type inference algorithm is implemented in Cedille, a language with impredicativity, dependent types, and dependent intersections. A local type inference system will be a good foundation for considering the following extensions:

- *partial* type propagation a la “Colored Local Type Inference”
- higher-order type inference using matching
- inference for *erased* term arguments (Cedille feature)
- subsumption based on some form of “type containment”
Thanks!

Questions?