Local Type Inference with Symbolic Closures

Ambrose Bonnaire-Sergeant
What is Local Type Inference?

1. Bidirectional type checking
2. Parameter type inference
3. Type argument inference
Bidirectional checking

▲ Synthesis mode (types propagate up)  ▼ Checking mode (types propagate down)
Bidirectional checking

- Synthesis mode (types propagate up)
- Checking mode (types propagate down)

\[ \Gamma \vdash n : \text{Int} \]
\[ \Gamma \vdash s : \text{Str} \]
\[ e : \text{Int} \]
\[ \Gamma \vdash (\text{inc } e) : \text{Int} \]
Bidirectional checking

- Synthesis mode (types propagate up)
- Checking mode (types propagate down)

\[ \Gamma \vdash n \triangleleft \text{Int} \]
\[ \Gamma \vdash s \triangleleft \text{Str} \]
\[ \Gamma \vdash e \triangleleft \text{Int} \]
\[ \Gamma \vdash (\text{inc } e) \triangleleft \text{Int} \]
\[ \Gamma, x : T \vdash e \triangleleft \text{S} \]
\[ \Gamma \vdash (\lambda (x : T) e) \triangleleft T \rightarrow S \]
Bidirectional checking

Synthesis mode (types propagate up)          Checking mode (types propagate down)

Example: Checking (inc 1)
Bidirectional checking

Synthesis mode (types propagate up)  
Checking mode (types propagate down)

Simple for implementors and users to conceptualize

Yields predictable, local error messages

Example: Checking (inc 1)

\[
\Gamma \vdash 1 \triangleright \text{Int} \\
\Gamma \vdash 1 \triangleright \text{Int} \\
\underline{\Gamma \vdash (inc \ 1) \triangleright \text{Int}}
\]
Parameter type inference

Input (Clojure)

\[(\text{ann} \ (fn \ [x] \ (inc \ x)) \ [\text{Int} \rightarrow \text{Int}])\]

Output (System F)

\[(fn \ [x :- \text{Int}] \ (inc \ x))\]

Infer function parameter types
Type Argument Reconstruction

Input (Clojure)

(map inc [1 2 3])

Infer type arguments

Output (System F)

(map<Int,Int> inc [1 2 3])
The “Hard-to-Synthesize Arguments” Problem

(map (fn [x] (inc x)) [1 2 3])
The “Hard-to-Synthesize Arguments” Problem

(map (fn [x] (inc x)) [1 2 3])

Cannot simultaneously infer type arguments to `map` and missing parameter type
The “Hard-to-Synthesize Arguments” Problem

`(map (fn [x] (inc x)) [1 2 3])`

Cannot simultaneously infer type arguments to `map` and missing parameter type

Why?
To infer type arguments, you must first synthesize types for operands…

…but unannotated functions are hard-to-synthesize types for!
Existing solutions

**Typed {Racket, Clojure}**  
Note: Any = ⊤

Still doesn't check!

(map (fn [x : Any] (inc x))  
[1 2 3])

**TypeScript**  
Note: any ≈ (void*)

Function body is trusted!

[1,2,3].map(((x:any)=>x+1)

**Reticulated Python**

Runtime overhead

map(lambda (x:Dyn): x+1,  
[1,2,3])
Existing solutions

Java Lambdas

List.of(1,2,3)
  .map(x->x+1)

Type args

Param type (inferred as Int)
Java Lambdas

```java
roster
  .stream()
  .filter(
    p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25)
  .map(p -> p.getEmailAddress())
  .forEach(email -> System.out.println(email));
```

...is this achievable
with non-OO idioms?
Solving the “Hard-to-synthesize arguments” problem with Symbolic Analysis
Another hard-to-synthesize term

(let [f (fn [x] x)]
  (f 1)
  (f "a")

How to check?
Wishful thinking

1. Infer polymorphic principal-like type for f

(let [f (ann (fn [x] x) (All [a] [a -> a]))]
  (let [f (fn [x] x)]
    (f 1)
    (f "a")
    (f "a")))
Wishful thinking

1. Infer polymorphic principal(-like) type for \( f \)

\[
\text{let } [f (\text{ann} (\text{fn} [x] x)) (\text{All} [a] [a \to a])]
\]

\[
(\text{let } [f (\text{fn} [x] x)]
(f 1)
(f "a")
)
\]

2. Infer sufficiently capable intersection type for \( f \)

\[
(\text{let } [f (\text{ann} (\text{fn} [x] x))
(\text{IFn} [\text{Int} \to \text{Int}]
[\text{Str} \to \text{Str}])])
\]

(f 1)
(f "a")
Wishful thinking

1. Infer polymorphic principal(-like) type for f

   \[
   \text{(let \textit{[f (ann (fn [x] x)
   \quad (\text{All} [a] [a \to a])])}}
   \]

\[
\begin{align*}
\text{(let \textit{[f (fn [x] x)]}} & \quad (f \ 1) \\
(f \ 1) & \quad (f \ \textit{``a''})
\end{align*}
\]

2. Infer sufficiently capable intersection type for f

\[
\text{(let \textit{[f (ann (fn [x] x)
   \quad (IFn [Int \to Int]\n   \quad [Str \to Str])])}}
   \]

\[
\begin{align*}
(f \ 1) & \quad (f \ \textit{``a''})
\end{align*}
\]

This talk:
Achieving this transformation within the framework of Local Type Inference
Challenges

(let [f (fn [x] x)]
  (f 1)
  (f "a")

Posed by Hosoya & Pierce,
“How Good is Local Type Inference?” (1999)
Challenges

(let [f (fn [x] x)]
  (f 1)
  (f "a")

1. How to delay the checking of hard-to-synthesize terms?

Posed by Hosoya & Pierce,
“How Good is Local Type Inference?” (1999)
Challenges

1. How to delay the checking of hard-to-synthesize terms?

2. How to force checking of hard-to-synthesize terms to preserve soundness?

Posed by Hosoya & Pierce, “How Good is Local Type Inference?” (1999)
Idea 1: Inline let-bound functions

(let [f (fn [x] x)]
  (f 1)
  (f "a"'))
Idea 1: Inline let-bound functions

(let [f (fn [x] x)]
  (f 1)
  (f "a"))

(let []
  ((fn [x] x) 1)
  ((fn [x] x) "a"))
Idea 1: Inline let-bound functions

(let [f (fn [x] x)]
  (f 1)
  (f "a")
)

1. How to delay the checking of hard-to-synthesize terms?
   A: Inline let-bound unannotated functions
Idea 1: Inline let-bound functions

(\[f (fn [x] x)\]
 (f 1)
 (f ‘a’))

(let []
 ((fn [x] x) 1)
 ((fn [x] x) ‘a’))

1. How to delay the checking of hard-to-synthesize terms?
   A: Inline let-bound unannotated functions

2. How to force checking of hard-to-synthesize terms to preserve soundness?
   A: Automatic
Idea 1: Inline let-bound functions

(let [f (fn [x] x)]
  (f 1)
  (f "a"))

1. How to delay the checking of hard-to-synthesize terms?
   A: Inline let-bound unannotated functions

2. How to force checking of hard-to-synthesize terms to preserve soundness?
   A: Automatic

Problem: Variable-capture
Idea 1: Inline let-bound functions

(let [f (let [y <DB-write>]  
         (fn [x] y y))]  
  (f 1)  
  (f “a’’))
Idea 1: Inline let-bound functions

(let [f (let [y <DB-write>]
    (fn [x] y y))]
(f 1)
(f "a")

(let []
  ((let [y <DB-write>]
      (fn [x] y y))
   1)
  ((let [y <DB-write>]
      (fn [x] y y)
    "a")))

?
Idea 1: Inline let-bound functions

(let [f (let [y <DB-write>]
            (fn [x] y y))]
  (f 1)
  (f "a"))
Idea 2: Let-polymorphism

(let [f (fn [x] x)]
  (f 1)
  (f "a"))

Let-polymorphism infers a principal type scheme for `f` and copies the type (with renamed unification variables) in each occurrence of `f` for separate instantiation.

...immediately doesn’t work because f’s type is hard-to-synthesize! (no unification variables in Local Type Inference)
Idea 3: “Delayed function type”

(let [f (fn [x] x)]
  (f 1)
  (f "a"))
Idea 3: “Delayed function type”

(let [f (fn [x] x)]
  f : @(fn [x] x)
  (f 1)
  (f "a")
)

1. How to delay the checking of hard-to-synthesize terms?

A: Introduction rule for unannotated functions makes a “delayed function type”
Idea 3: “Delayed function type”

(let [f (fn [x] x)]
  f : @(fn [x] x)
(f 1)
(f "a")

@/(fn [x] x) <: Int -> ?

1. How to delay the checking of hard-to-synthesize terms?
   A: Introduction rule for unannotated functions makes a “delayed function type”

2. How to force checking of hard-to-synthesize terms to preserve soundness?
   A: Applications of delayed function types rechecks the function’s source code with given argument types
Idea 3: “Delayed function type”

(let [f (fn [x] x)]
  (f 1)
  (f ‘a’))

1. How to delay the checking of hard-to-synthesize terms?
   A: Introduction rule for unannotated functions makes a “delayed function type”

2. How to force checking of hard-to-synthesize terms to preserve soundness?
   A: Applications of delayed function types rechecks the function’s source code with given argument types
Idea 3: “Delayed function type”

(let [f (fn [x] x)]
  f : @(fn [x] x)
  (f 1)
  (f ‘a’))

@(fn [x] x) <: Int -> ?
@(fn [x] x) <: Str -> ?

1. How to delay the checking of hard-to-synthesize terms?
   A: Introduction rule for unannotated functions makes a “delayed function type”

2. How to force checking of hard-to-synthesize terms to preserve soundness?
   A: Applications of delayed function types rechecks the function’s source code with given argument types

**Problem: Undecidable!**
Idea 3: “Delayed function type”

(let [f (fn [f] (f f))]
  (f f))

1. Delay (fn [f] (f f))
2. Check (f f)
3. Check (f f)
4. Check (f f)
...

Problem: Undecidable!
Restrictions

**Insight:**
Many local functions are not recursive
(implicitly or explicitly)
Restrictions

**Insight:**
Many local functions are not recursive (implicitly or explicitly)

**Insight:**
Most top-level functions have annotations anyway, and are otherwise valuable to add
Restrictions

**Insight:**
Many local functions are not recursive (implicitly or explicitly)

**Insight:**
Most top-level functions have annotations anyway, and are otherwise valuable to add

**New Restrictions:**
1. Only delay local functions
2. Do not allow delayed functions to escape its top-level form
3. Use fuel to make uncommon cases (recursive locals) conservatively decidable
Idea 3: “Delayed function type”

(let [f (fn [f] (f f))]
  (f f))

1. Delay (fn [f] (f f))
2. Check (f f) Fuel = 2
3. Check (f f) Fuel = 1
4. Check (f f) Fuel = 0
5. Type error: Reduction limit

Tradeoff: Platform dependency
Idea 3: “Delayed function type”

(let [f (let [y 1]
            (fn [x] y))]
  (f 1)
  (f "a"))

**Problem: Variable Capture!**
Idea 3: “Delayed function type”

(let [f (let [y 1]
             (fn [x] y))]
 (f 1)
 (f “a”))

**Problem: Variable Capture!**
Idea 3: “Delayed function type”

(let [f (let [y 1] (fn [x] y))]
  (f 1)
  (f "a")
  f : @(fn [x] y)

Problem: Variable Capture!
(let [f (let [y 1]
  (fn [x] y))]
  (f 1)
  (f "a"))

Keep type environment for when we need it ("type-level" closure)
Solution: Symbolic Closures

(let [f (let [y 1]
            (fn [x] y))]
  (f 1)
  (f "a")

Can check y!

Keep type environment for when we need it ("type-level" closure)
Example Elaboration
Elaboration with Symbolic Closures

Input
(let [f (fn [x] x)]
  (f 1)
  (f "a"))

Output
(let [f (ann (fn [x] x)
  (IFn [Int -> Int]
  [Str -> Str]))]
  (f 1)
  (f "a"))
Elaboration with Symbolic Closures

Input

(let [f (fn [x] x)] ➞ 1. Assign f a symbolic closure:  f : {}@(fn [x] x)
   (f 1)
   (f "a’")

Output

(let [f (ann (fn [x] x)
              (IFn [Int -> Int]
                [Str -> Str]))]
  (f 1)
  (f "a’")
Elaboration with Symbolic Closures

Input

(let [f (fn [x] x)]
  (f 1)
  (f "a"))

1. Assign f a symbolic closure:  
   \[ f : \{\}@(fn [x] x) \]

2. Check `f` with Int (returns Int)  
   \[ f <: Int \rightarrow ? \]

Output

(let [f (ann (fn [x] x)
           (IFn [Int \rightarrow Int]
                [Str \rightarrow Str]))]
  (f 1)
  (f "a"))
Elaboration with Symbolic Closures

Input

(let [f (fn [x] x)]
  (f 1)
  (f "a'"))

Output

(let [f (ann (fn [x] x)
               (IFn [Int -> Int]
                    [Str -> Str]))]
  (f 1)
  (f "a'"))
Elaboration with Symbolic Closures

Input
(let [f (fn [x] x)]
  (f 1)
  (f “a’”))

Output
(let [f (ann (fn [x] x)
  (IFn [Int -> Int]
  [Str -> Str])]
  (IFn [Int -> Int]
  [Str -> Str]))
  (f 1)
  (f “a’”))

1. Assign f a symbolic closure:  
   \( f : {}@(fn [x] x) \)
2. Check `f` with Int (returns Int)  
   \( f <: Int -> ? \)
3. Check `f` with Str (returns Str)  
   \( f <: Str -> ? \)
4. Replace f’s type with its capabilities
End example, break for questions?
More about Symbolic Closures
\textbf{C-AppClosure}
\[
\Gamma \vdash f : \Gamma' \circ \lambda x. e' \quad \Gamma \vdash^c e : \sigma \quad \Gamma', x : \sigma \vdash e' : \tau
\]
\[
\Gamma \vdash^c (f\ e) : \tau
\]
C-AppClosure
\[ \Gamma \vdash f : \Gamma' \circ \lambda x.e' \quad \Gamma \vdash^c e : \sigma \quad \Gamma', x : \sigma \vdash e' : \tau \]
\[ \Gamma \vdash^c (f e) : \tau \]

SC-Closure
\[ \Gamma, \bar{\alpha}, x : \tau \vdash e : \sigma \]
\[ \Gamma \circ \lambda x.e \leq (\tau \xrightarrow{\bar{\alpha}} \sigma) \]
**C-AppClosure**

\[ \Gamma \vdash f : \Gamma' @ \lambda x. e' \quad \Gamma \vdash^c e : \sigma \quad \Gamma', x : \sigma \vdash e' : \tau \]

\[ \Gamma \vdash^c (f \ e) : \tau \]

Subtyping relation calls type checker

**SC-Closure**

\[ \Gamma, \overline{\alpha}, x : \tau \vdash e : \sigma \]

\[ \Gamma @ \lambda x. e \leq (\tau \overline{\alpha} \rightarrow \sigma) \]
C-AppClosure
\[\Gamma \vdash f : \Gamma' \@ \lambda x . e' \quad \Gamma \vdash^c e : \sigma \quad \Gamma', x : \sigma \vdash e' : \tau\]
\[\Gamma \vdash^c (f \ e) : \tau\]

SC-Closure
\[\Gamma, \overline{\alpha}, x : \tau \vdash e : \sigma\]
\[\Gamma \@ \lambda x . e \leq (\tau \xrightarrow{\overline{\alpha}} \sigma)\]
How to check?

(map (fn [x] (inc x)) [1 2 3])
How to check?

Derive data-flow graph from operator

(map (fn [x] (inc x)) [1 2 3])

(All [a b]
 [[a -> b] (Seqable a) -> (Seq b)])
How to check?

(map (fn [x] (inc x))
    [1 2 3])

Derive data-flow graph from operator

(All [a b]
    ([[a -> b] (Seqable a) -> (Seq b)]))

Solve constraints to a fixed point

{ @(fn [x] (inc x)) <: [a -> b] => C1
    (Vec Number) <: (Seqable a) => C2
How to check?

Derive data-flow graph from operator

Solve constraints to a fixed point

Future work:

(map (fn [x] (inc x)) 
  [1 2 3])

(All [a b]
  [[a -> b] (Seqable a) -> (Seq b)])

{}@(fn [x] (inc x)) <: [a -> b] => C1
(Vec Number) <: (Seqable a) => C2

What if data-flow is recursive?
Related work
Related work

Expansion variables

\[
\begin{align*}
\langle (z : a) \vdash ((a \rightarrow b) \rightarrow b) \rightarrow c \rangle & \rightarrow c \\
\downarrow
\langle (z : a_1 \cap a_2) \vdash (((a_1 \rightarrow b_1) \rightarrow b_1) \cap ((a_2 \rightarrow b_2) \rightarrow b_2) \rightarrow c) \rightarrow c \rangle
\end{align*}
\]

Similar goal as “Expansion variables” in Intersection Type Inference

Similar cost:
Inference cost = Beta-reduction cost

Related work

Colored Local Type Inference

Allows partial type information to propagate down term

For instance, if \( g \) is known to have type \( \forall a. (\text{Int} \rightarrow a) \rightarrow a \), then

\[
g(\text{fun}(x)x + 1)
\]

Conservative extension of Local Type Inference

Odersky et al. Colored Local Type Inference (POPL 2001)
Review

**Background:**
Local type inference requires annotations
Review

**Background:**
Local type inference requires annotations

**Problem:**
Local annotations are annoying
Review

**Problem:**
Local annotations are annoying

**Background:**
Local type inference requires annotations

**Insight:**
Top-level annotations are provided

**Insight:**
Local functions are usually trivial
**Problem:**
Local annotations are annoying

**Background:**
Local type inference requires annotations

**Solution:**
Use symbolic analysis to infer simple local functions

**Insight:**
Top-level annotations are provided

**Insight:**
Local functions are usually trivial
Thanks!