Type Inference
For the Simply-Typed Lambda Calculus

Peter Thiemann, Manuel Geffken
University of Freiburg
24. Januar 2013
Outline

1 Introduction

2 Applied Lambda Calculus

3 Simple Types for the Lambda Calculus

4 Type Inference for the Simply-Typed Lambda Calculus
“Static type systems are the world’s most successful application of formal methods” (Simon Peyton Jones)

Formally, a type system defines a relation between a set of executable syntax and a set of types

To express properties of the execution, the typing relation must be compatible with execution

⇒ Type soundness

A type system for analysis must be able to construct a typing from executable syntax

⇒ Type inference
Type inference is also referred to as *type reconstruction*.

Included in ML, OCaml, Haskell, Scala, . . .

. . . and to some extend in C#, C++11

We can distinguish different kinds of type inference

- Global type inference: ML, OCaml, Haskell
  - Infers the „most general“ type
  - Difficult to combine with subtyping
- Local type inference: Scala, C#, . . .
Example: Global and Local Type Inference

Global: No type annotations needed (OCaml)

```ocaml
let addToSecond (tuple, i) =
  let x = snd tuple + i in
  x;
addToSecond (("a", 1), 1);
```

Local: Type annotations at function boundaries required (Scala)

```scala
def addToSecond(tuple: (String, Int), i: Int) = {
  var x = tuple._2 + i;
  x;
}
addToSecond ("a", 1, 1);
```
Outline

1. Introduction

2. Applied Lambda Calculus

3. Simple Types for the Lambda Calculus

4. Type Inference for the Simply-Typed Lambda Calculus
Applied Lambda Calculus

Syntax of Applied Lambda Calculus

Let $x \in \text{Var}$, a countable set of variables, and $n \in \mathbb{N}$.

$$\text{Exp} \ni e ::= x | \lambda x . e | e \\ e | \lceil n \rceil | \text{succ } e$$

A term is either a variable, an abstraction (with body $e$), an application, a numeric constant, or a primitive operation.

Conventions

- Applications associate to the left.
- The body of an abstraction extends as far right as possible.
- $\lambda xy . e$ stands for $\lambda x . \lambda y . e$ (and so on).
- Abstraction and constant are introduction forms, application and primitive operation are elimination forms.
Values of Applied Lambda Calculus

\[ \text{Val} \ni v ::= \lambda x. e \mid \boxed{n} \]

A value is either an abstraction or a numeric constant. Each value is an expression: \( \text{Val} \subseteq \text{Exp} \).
The functions $FV(\cdot), BV(\cdot): \text{Exp} \rightarrow \mathcal{P}((\text{Var})$ return the set of free and bound variables of a lambda term, respectively.

<table>
<thead>
<tr>
<th>$e$</th>
<th>$FV(e)$</th>
<th>$BV(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>${x}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\lambda x. e$</td>
<td>$FV(e) \setminus {x}$</td>
<td>$BV(e) \cup {x}$</td>
</tr>
<tr>
<td>$e_0 ; e_1$</td>
<td>$FV(e_0) \cup FV(e_1)$</td>
<td>$BV(e_0) \cup BV(e_1)$</td>
</tr>
<tr>
<td>$[n]$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\text{succ } e$</td>
<td>$FV(e)$</td>
<td>$BV(e)$</td>
</tr>
</tbody>
</table>

$\text{Var}(e) := FV(e) \cup BV(e)$ is the set of variables of $e$. A lambda term $e$ is closed ($e$ is a combinator) iff $FV(e) = \emptyset$. 
Computation in Applied Lambda Calculus

- Computation defined by term rewriting / reduction
- Three reduction relations
  - Alpha reduction (alpha conversion)
  - Beta reduction
  - Delta reduction
- Each relates a family of redexes to a family of contracta.
Reduction Rules of Lambda Calculus

## Alpha Conversion

- Renaming of bound variables

\[ \lambda x. e \rightarrow_\alpha \lambda y. e[x \mapsto y], \quad y \notin FV(e) \]

- Alpha conversion is often applied tacitly and implicitly.

## Beta Reduction

- Only computation step
- Intuition: Function call

\[ (\lambda x. e) f \rightarrow_\beta e[x \mapsto f] \]
Reduction Rules, cont’d

Delta Reduction

- Operations on built-in types

\[
\text{succ } [n] \quad \rightarrow_\delta \quad [n + 1]
\]
Delta Reduction

- Operations on built-in types

\[ \text{succ } [n] \mapsto_{\delta} [n + 1] \]

Reduction in Context

In Lambda Calculus, the reduction rules may be applied anywhere in a term. Execution in a programming language is more restrictive. It is usually reduces according to a reduction strategy:

- call-by-name or
- call-by-value
Reduction Rules, cont’d

Call-by-Name Reduction

\[
\begin{align*}
\text{Beta} & \quad e \to_\beta e' \\
& \quad \frac{e \to_n e'}{e} \\
\text{AppL} & \quad f \to_n f' \\
& \quad \frac{f \to_n f' e}{f e} \\
\text{SuccL} & \quad e \to_n e' \\
& \quad \frac{\text{succ} e \to_n \text{succ} e'}{\text{succ} e} \\
\text{Delta} & \quad e \to_\delta e' \\
& \quad \frac{e \to_n e'}{e}
\end{align*}
\]
Reduction Rules, cont’d

### Call-by-Value Reduction

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntax</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta-V</td>
<td>$(\lambda x. e) , v \rightarrow_v e[x \mapsto v]$</td>
<td>$\frac{f \rightarrow_v f'}{f , e \rightarrow_v f' , e}$</td>
</tr>
<tr>
<td>AppL</td>
<td>$f \rightarrow_v f'$</td>
<td>$\frac{v , e \rightarrow_v v , e'}{e \rightarrow_v e'}$</td>
</tr>
<tr>
<td>VAppR</td>
<td>$e \rightarrow_v e'$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Syntax</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuccL</td>
<td>$e \rightarrow_v e'$</td>
<td>$\frac{e \rightarrow_v e'}{\text{succ} , e \rightarrow_v \text{succ} , e'}$</td>
</tr>
<tr>
<td>Delta</td>
<td>$e \rightarrow_\delta e'$</td>
<td>$\frac{e \rightarrow_\delta e'}{e \rightarrow_v e'}$</td>
</tr>
</tbody>
</table>
Computation in Lambda Calculus

Computation = Iterated Reduction

Let $x \in \{n, v\}$.

$$e \rightarrow^* e$$

$$e \rightarrow_x e' \quad e' \rightarrow^* e''$$

$$e \rightarrow^* e''$$

Outcomes of Computation

Starting a computation at $e$ may lead to

- Nontermination: $\forall e', e \rightarrow^*_x e'$ exists $e''$ such that $e' \rightarrow_x e''$
- Termination: $\exists e', e \rightarrow^*_x e'$ such that for all $e''$, $e' \not\rightarrow_x e''$
  $e'$ is irreducible.
  If $e'$ is a value, then it is the result of the computation.
Examples of Irreducible Forms

1. $[42]$
2. $\lambda f x y . f \ x \ y$
3. $[1] \ \lambda x . x$
5. $\textit{succ} \ \lambda x . x$
Examples of Irreducible Forms

1. \([42]\)
2. \(\lambda f x y. f \times y\)
3. \([1] \lambda x. x\)
4. \([1] [2]\)
5. \(\text{succ} \lambda x. x\)

Expected Benefits of a Type System

- 1–2 are values
- 3–5 contain elimination forms that try to eliminate non-variables without a corresponding rule (run-time errors)
- should be ruled out by a type system
Outline

1. Introduction

2. Applied Lambda Calculus

3. Simple Types for the Lambda Calculus

4. Type Inference for the Simply-Typed Lambda Calculus
• Language of types

\[ \tau ::= \alpha \mid \text{Nat} \mid \tau \rightarrow \tau \]

• Typing environment (function from variables to types)

\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

• Typing judgment (relation between terms and types):
In typing environment \( \Gamma \), \( e \) has type \( \tau \)

\[ \Gamma \vdash e : \tau \]
Inference Rules for STLC

\[ \text{VAR} \]
\[ \Gamma \vdash x : \Gamma(x) \]

\[ \text{LAM} \]
\[ \Gamma, x : \tau \vdash e : \tau' \]
\[ \frac{\Gamma \vdash \lambda x. e : \tau \to \tau'}{\Gamma \vdash \lambda x. e : \tau \to \tau'} \]

\[ \text{APP} \]
\[ \Gamma \vdash e_0 : \tau \to \tau' \quad \Gamma \vdash e_1 : \tau \]
\[ \frac{\Gamma \vdash e_0 \ e_1 : \tau'}{\Gamma \vdash e_0 \ e_1 : \tau'} \]

\[ \text{NUM} \]
\[ \Gamma \vdash [n] : \text{Nat} \]

\[ \text{Succ} \]
\[ \frac{\Gamma \vdash e : \text{Nat}}{\Gamma \vdash \text{succ} \ e : \text{Nat}} \]
Example Inference Tree

\[
\begin{align*}
\cdots \vdash f : \alpha & \rightarrow \alpha \\
\cdots \vdash f : \alpha & \rightarrow \alpha \\
\cdots \vdash x : \alpha \\
\cdots \vdash f x : \alpha \\
\vdash f : \alpha & \rightarrow \alpha, x : \alpha \vdash f (f x) : \alpha \\
\vdash f : \alpha & \rightarrow \alpha \vdash \lambda x. f (f x) : \alpha \rightarrow \alpha \\
\vdash \lambda f. \lambda x. f (f x) : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha
\end{align*}
\]
Type Soundness

Type Preservation

If \( \Gamma \vdash e : \tau \) and \( e \rightarrow_x e' \), then \( \Gamma \vdash e' : \tau \).

Proof by induction on \( e \rightarrow e' \)

Progress

If \( \cdot \vdash e : \tau \), then either \( e \) is a value or there exists \( e' \) such that \( e \rightarrow_x e' \).

Proof by induction on \( \Gamma \vdash e : \tau \)

Type Soundness

If \( \cdot \vdash e : \tau \), then either

1. exists \( v \) such that \( e \rightarrow^*_x v \) or

2. for each \( e' \), such that \( e \rightarrow^*_x e' \) there exists \( e'' \) such that \( e' \rightarrow_x e'' \).
Outline

1. Introduction

2. Applied Lambda Calculus

3. Simple Types for the Lambda Calculus

4. Type Inference for the Simply-Typed Lambda Calculus
Typing Problems

- Type checking: Given environment $\Gamma$, a term $e$ and a type $\tau$, is $\Gamma \vdash e : \tau$ derivable?
- Type inference: Given a term $e$, are there $\Gamma$ and $\tau$ such that $\Gamma \vdash e : \tau$ is derivable?
Typing Problems

- Type checking: Given environment $\Gamma$, a term $e$ and a type $\tau$, is $\Gamma \vdash e : \tau$ derivable?
- Type inference: Given a term $e$, are there $\Gamma$ and $\tau$ such that $\Gamma \vdash e : \tau$ is derivable?

Typing Problems for STLC

- Type checking and type inference are decidable for STLC
- Moreover, for each typable $e$ there is a principal typing $\Gamma \vdash e : \tau$ such that any other typing is a substitution instance of the principal typing.
Let $\mathcal{E}$ be a set of equations on types.

### Unifiers and Most General Unifiers

- A substitution $S$ is a **unifier of $\mathcal{E}$** if, for each $\tau \equiv \tau' \in \mathcal{E}$, it holds that $S\tau = S\tau'$.

- A substitution $S$ is a **most general unifier of $\mathcal{E}$** if $S$ is a unifier of $\mathcal{E}$ and for every other unifier $S'$ of $\mathcal{E}$, there is a substitution $T$ such that $S' = T \circ S$. 
Let $E$ be a set of equations on types.

### Unifiers and Most General Unifiers

- A substitution $S$ is a unifier of $E$ if, for each $\tau \equiv \tau' \in E$, it holds that $S\tau = S\tau'$.

- A substitution $S$ is a most general unifier of $E$ if $S$ is a unifier of $E$ and for every other unifier $S'$ of $E$, there is a substitution $T$ such that $S' = T \circ S$.

### Unification

There is an algorithm $U$ that, on input of $E$, either returns a most general unifier of $E$ or fails if none exists.
Principal Type Inference for STLC

The algorithm transforms a term into a principal typing judgment for the term (or fails if no typing exists).

\[
\begin{align*}
\mathcal{P}(x) &= \text{return } x : \alpha \vdash x : \alpha \\
\mathcal{P}(\lambda x . e) &= \text{let } \Gamma \vdash e : \tau \leftarrow \mathcal{P}(e) \text{ in} \\
&\quad \text{if } x : \tau_x \in \Gamma \text{ then return } \Gamma_x \vdash \lambda x . e : \tau_x \to \tau \\
&\quad \text{else choose } \alpha \notin \text{Var}(\Gamma, \tau) \text{ in} \\
&\quad \quad \text{return } \Gamma \vdash \lambda x . e : \alpha \to \tau \\
\mathcal{P}(e_0 \ e_1) &= \text{let } \Gamma_0 \vdash e_0 : \tau_0 \leftarrow \mathcal{P}(e_0) \text{ in} \\
&\quad \text{let } \Gamma_1 \vdash e_1 : \tau_1 \leftarrow \mathcal{P}(e_1) \text{ in} \\
&\quad \text{with disjoint type variables in } (\Gamma_0, \tau_0) \text{ and } (\Gamma_1, \tau_1) \text{ in} \\
&\quad \text{choose } \alpha \notin \text{Var}(\Gamma_0, \Gamma_1, \tau_0, \tau_1) \text{ in} \\
&\quad \text{let } S \leftarrow \mathcal{U}(\Gamma_0 \hat{=} \Gamma_1, \tau_0 \hat{=} \tau_1 \to \alpha) \text{ in} \\
&\quad \text{return } S \Gamma_0 \cup S \Gamma_1 \vdash e_0 \ e_1 : S\alpha \\
\mathcal{P}([n]) &= \text{return } \cdot \vdash [n] : \text{Nat} \\
\mathcal{P}(\text{succ } e) &= \text{let } \Gamma \vdash e : \tau \leftarrow \mathcal{P}(e) \text{ in} \\
&\quad \text{let } S \leftarrow \mathcal{U}(\tau \hat{=} \text{Nat}) \text{ in} \\
&\quad \text{return } \Gamma \vdash \text{succ } e : \text{Nat}
\end{align*}
\]