Program Analysis and Verification . . . Using Types
Applied Simply-Typed Lambda Calculus

Peter Thiemann
University of Freiburg
May 2014
Outline

1 Introduction
2 Applied Lambda Calculus
3 Simple Types for the Lambda Calculus
4 Type Inference for the Simply-Typed Lambda Calculus
Static Program Analysis (PA)

Find a safe approximation of program properties without executing the program.
Static Program Analysis (PA)

Find a safe approximation of program properties without executing the program.
Terms

Type-Based Program Analysis

- PA (and verification) using types
  - Program is typed $\Rightarrow$ Program has property
  - Dependent types

- PA on top of type structure
  - Analysis builds abstraction on a typed program
  - Typing improves the precision by eliminating impossible scenarios

- PA using type inference
  - Piggy-back properties on types
  - Use inference to propagate properties
“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.

$\Rightarrow$ *Type soundness*

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

$\Rightarrow$ *Type inference*
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 \( n \in \mathbb{N} \).

\[
\text{Exp} \ni e ::= x | \lambda x . e | e \ e | [n] | \text{succ} \ e
\]

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.
A value is either an abstraction or a numeric constant.
Each value is an expression: Val ⊆ Exp.
Variable Occurrences
Free and Bound Variables

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>$\left[ n \right]$</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

\[ \text{succ } [n] \rightarrow_{\delta} [n + 1] \]
Reduction Rules, cont’d

**Delta Reduction**

- Operations on built-in types

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

**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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Beta</strong></td>
<td>$e \to \beta e'$</td>
<td>$e \to_n e'$</td>
</tr>
<tr>
<td><strong>AppL</strong></td>
<td>$f \to_n f'$</td>
<td>$f \to_n f' e$</td>
</tr>
<tr>
<td><strong>SuccL</strong></td>
<td>$e \to_n e'$</td>
<td>$\text{succ } e \to_n \text{succ } e'$</td>
</tr>
<tr>
<td><strong>Delta</strong></td>
<td>$e \to_\delta e'$</td>
<td>$e \to_n e'$</td>
</tr>
</tbody>
</table>
Reduction Rules, cont’d

Call-by-Value Reduction

\[
\begin{align*}
\text{Beta-V} & \quad (\lambda x. e) v \rightarrow v e[x \mapsto v] \\
\text{AppL} & \quad f \rightarrow v f' \quad \frac{f e \rightarrow v f' e}{v e \rightarrow v v e'} \\
\text{VAppR} & \quad e \rightarrow v e' \quad \frac{v e \rightarrow v v e'}{v e \rightarrow v v e'} \\
\text{SuccL} & \quad e \rightarrow v e' \quad \frac{\text{succ } e \rightarrow v \text{ succ } e'}{	ext{succ } e \rightarrow v \text{ succ } e'} \\
\text{Delta} & \quad e \rightarrow \delta e' \quad \frac{e \rightarrow v e'}{e \rightarrow v e'}
\end{align*}
\]
Computation in Lambda Calculus

Computation = Iterated Reduction

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

\[
e \rightarrow^*_x e
\]

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 $e''$, $e' \not\rightarrow^*_x e''$

If $e'$ is a value, then it is the result of the computation.
Examples of Irreducible Forms

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

1 \[42\]
2 \(\lambda fxy. f \times y\)
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
Simple Types for the 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

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

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

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

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

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

\[ \ldots \vdash f : \alpha \to \alpha \]  
\[ \vdash f : \alpha \to \alpha \]  
\[ \vdash x : \alpha \]  
\[ \vdash f x : \alpha \]  
\[ f : \alpha \to \alpha, x : \alpha \vdash f(f x) : \alpha \]  
\[ f : \alpha \to \alpha \vdash \lambda x . f(f x) : \alpha \to \alpha \]  
\[ \vdash \lambda f . \lambda x . f(f x) : (\alpha \to \alpha) \to \alpha \to \alpha \]
Type Soundness

Type Preservation

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

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?
Type Inference for the Simply-Typed Lambda Calculus (STLC)

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 $\tau \doteq \tau' \in \mathcal{E}$, 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 $S' = T \circ S$. 
Unification

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 $\tau \doteq \tau' \in \mathcal{E}$, 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 $S' = T \circ S$.

**Unification**

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

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

\[
P(x) = \text{return } x : \alpha \vdash x : \alpha
\]

\[
P(\lambda x. e) = \begin{align*}
&\text{let } \Gamma \vdash e : \tau \leftarrow P(e) \text{ in} \\
&\text{if } x : \tau_x \in \Gamma \text{ then return } \Gamma_x \vdash \lambda x. e : \tau_x \rightarrow \tau \\
&\text{else choose } \alpha \notin \text{Var}(\Gamma, \tau) \text{ in} \\
&\quad \text{return } \Gamma \vdash \lambda x. e : \alpha \rightarrow \tau
\end{align*}
\]

\[
P(e_0 \ e_1) = \begin{align*}
&\text{let } \Gamma_0 \vdash e_0 : \tau_0 \leftarrow P(e_0) \text{ in} \\
&\text{let } \Gamma_1 \vdash e_1 : \tau_1 \leftarrow P(e_1) \text{ in} \\
&\text{with disjoint type variables in } (\Gamma_0, \tau_0) \text{ and } (\Gamma_1, \tau_1) \\
&\text{choose } \alpha \notin \text{Var}(\Gamma_0, \Gamma_1, \tau_0, \tau_1) \text{ in} \\
&\text{let } S \leftarrow U(\Gamma_0 \doteq \Gamma_1, \tau_0 \doteq \tau_1 \rightarrow \alpha) \text{ in} \\
&\text{return } S\Gamma_0 \cup S\Gamma_1 \vdash e_0 \ e_1 : S\alpha
\end{align*}
\]

\[
P([n]) = \text{return } \cdot \vdash [n] : \text{Nat}
\]

\[
P(\text{succ } e) = \begin{align*}
&\text{let } \Gamma \vdash e : \tau \leftarrow P(e) \text{ in} \\
&\text{let } S \leftarrow U(\tau \doteq \text{Nat}) \text{ in} \\
&\text{return } S\Gamma \vdash \text{succ } e : \text{Nat}
\end{align*}
\]