Type Inference

For the Simply-Typed Lambda Calculus

Albert-Ludwigs-Universität Freiburg

Peter Thiemann, Manuel Geffken

University of Freiburg

24. Januar 2013

UNI FREIBURG

Outline

Foundation: Type Systems

- "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*

Foundation: 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)

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

Local: Type annotations at function boundaries required (Scala)

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

Outline

Applied Lambda Calculus

Syntax of Applied Lambda Calculus

Let $x \in \mathrm{Var}$, a countable set of *variables*, and $n \in \mathbf{N}$.

$$\mathrm{Exp} \ni \quad e \quad ::= \quad x \mid \lambda x\,.\,e \mid e\ e \mid \lceil n \rceil \mid \mathit{succ}\ e$$

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*.

Applied Lambda Calculus

Values of Applied Lambda Calculus

$$\mathsf{Val} \ni \quad v \quad ::= \quad \lambda x \,.\, e \mid \lceil n \rceil$$

A *value* is either an abstraction or a numeric constant.
Each value is an expression: $\mathsf{Val} \subseteq \mathsf{Exp}$.

The functions $FV(\cdot), BV(\cdot) : \mathrm{Exp} \to \mathcal{P}(\mathrm{Var})$ return the set of *free* and *bound* variables of a lambda term, respectively.

e	$FV(e)$	$BV(e)$
x	$\{x\}$	$\emptyset$
$\lambda x \,.\, e$	$FV(e) \setminus \{x\}$	$BV(e) \cup \{x\}$
$e_0\ e_1$	$FV(e_0) \cup FV(e_1)$	$BV(e_0) \cup BV(e_1)$
$\lceil n \rceil$	$\emptyset$	$\emptyset$
$succ\ e$	$FV(e)$	$BV(e)$

$Var(e) := FV(e) \cup BV(e)$ 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

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

- Alpha conversion is often applied tacitly and implicitly.

Beta Reduction

- Only computation step
- Intuition: Function call

$$(\lambda x.e)\ f \quad \rightarrow_\beta e[x \mapsto f]$$

Reduction Rules, cont'd

Delta Reduction

- Operations on built-in types

$$succ\ \lceil n \rceil \quad \rightarrow_\delta \lceil n + 1 \rceil$$

Reduction Rules, cont'd

Delta Reduction

- Operations on built-in types

$$\mathit{succ}\ \lceil n \rceil \quad \rightarrow_\delta \lceil n+1 \rceil$$

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

$$\textsc{Beta}\ \frac{e \rightarrow_\beta e'}{e \rightarrow_n e'} \qquad \textsc{AppL}\ \frac{f \rightarrow_n f'}{f\ e \rightarrow_n f'\ e} \qquad \textsc{SuccL}\ \frac{e \rightarrow_n e'}{succ\ e \rightarrow_n succ\ e'} \qquad \textsc{Delta}\ \frac{e \rightarrow_\delta e'}{e \rightarrow_n e'}$$

Reduction Rules, cont'd

Call-by-Value Reduction

$$\textsc{Beta-V} \quad (\lambda x\,.\,e)\ v \rightarrow_v e[x \mapsto v]$$

$$\textsc{AppL} \quad \frac{f \rightarrow_v f'}{f\ e \rightarrow_v f'\ e}$$

$$\textsc{VAppR} \quad \frac{e \rightarrow_v e'}{v\ e \rightarrow_v v\ e'}$$

$$\textsc{SuccL} \quad \frac{e \rightarrow_v e'}{succ\ e \rightarrow_v succ\ e'}$$

$$\textsc{Delta} \quad \frac{e \rightarrow_\delta e'}{e \rightarrow_v e'}$$

Computation in Lambda Calculus

Computation = Iterated Reduction

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

$$e \rightarrow_x^* e \qquad\qquad \frac{e \rightarrow_x e' \qquad e' \rightarrow_x^* e''}{e \rightarrow_x^* e''}$$

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'', $e' \not\rightarrow_x e''$
  e' is irreducible.
  If e' is a value, then it is the result of the computation.

Examples of Irreducible Forms

1. $\lceil 42 \rceil$
2. $\lambda fxy\,.\,f\;x\;y$
3. $\lceil 1 \rceil\;\lambda x\,.\,x$
4. $\lceil 1 \rceil\;\lceil 2 \rceil$
5. $succ\;\lambda x\,.\,x$

Examples of Irreducible Forms

1. $\lceil 42 \rceil$
2. $\lambda fxy\,.\,f\ x\ y$
3. $\lceil 1 \rceil\ \lambda x\,.\,x$
4. $\lceil 1 \rceil\ \lceil 2 \rceil$
5. $succ\ \lambda x\,.\,x$

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

Simple Types for the Lambda Calculus

- Language of types

$$\tau ::= \alpha \mid \texttt{Nat} \mid \tau \to \tau$$

- Typing environment (function from variables to types)

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

- Typing judgment (relation between terms and types):
  In typing environment Γ, e has type τ

$$\Gamma \vdash e : \tau$$

Inference Rules for STLC

VAR
$$\Gamma \vdash x : \Gamma(x)$$

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

APP
$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'}$$

NUM
$$\Gamma \vdash \lceil n \rceil : \mathtt{Nat}$$

SUCC
$$\frac{\Gamma \vdash e : \mathtt{Nat}}{\Gamma \vdash \mathit{succ}\ e : \mathtt{Nat}}$$

Example Inference Tree

$$
\dfrac{\ldots \vdash f : \alpha \to \alpha \qquad \dfrac{\ldots \vdash f : \alpha \to \alpha \qquad \ldots \vdash x : \alpha}{\ldots \vdash f\ x : \alpha}}{\dfrac{f : \alpha \to \alpha, x : \alpha \vdash f\ (f\ x) : \alpha}{\dfrac{f : \alpha \to \alpha \vdash \lambda x . f\ (f\ x) : \alpha \to \alpha}{\cdot \vdash \lambda f . \lambda x . f\ (f\ x) : (\alpha \to \alpha) \to \alpha \to \alpha}}}
$$

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

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

Typing Problems

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

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

Typing Problems

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

Prerequisites for Type Inference for STLC

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 $S' = T \circ S$.

Prerequisites for Type Inference for STLC

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

$$
\begin{array}{lcl}
\mathcal{P}(x) & = & \textbf{return } x : \alpha \vdash x : \alpha \\
\mathcal{P}(\lambda x . e) & = & \textbf{let } \Gamma \vdash e : \tau \leftarrow \mathcal{P}(e) \textbf{ in} \\
 & & \textbf{if } x : \tau_x \in \Gamma \textbf{ then return } \Gamma_x \vdash \lambda x . e : \tau_x \to \tau \\
 & & \textbf{else choose } \alpha \notin \mathit{Var}(\Gamma, \tau) \textbf{ in} \\
 & & \qquad \textbf{return } \Gamma \vdash \lambda x . e : \alpha \to \tau \\
\mathcal{P}(e_0\ e_1) & = & \textbf{let } \Gamma_0 \vdash e_0 : \tau_0 \leftarrow \mathcal{P}(e_0) \textbf{ in} \\
 & & \textbf{let } \Gamma_1 \vdash e_1 : \tau_1 \leftarrow \mathcal{P}(e_1) \textbf{ in} \\
 & & \textbf{with disjoint type variables in } (\Gamma_0, \tau_0) \textbf{ and } (\Gamma_1, \tau_1) \\
 & & \textbf{choose } \alpha \notin \mathit{Var}(\Gamma_0, \Gamma_1, \tau_0, \tau_1) \textbf{ in} \\
 & & \textbf{let } S \leftarrow \mathcal{U}(\Gamma_0 \doteq \Gamma_1, \tau_0 \doteq \tau_1 \to \alpha) \textbf{ in} \\
 & & \textbf{return } S\Gamma_0 \cup S\Gamma_1 \vdash e_0\ e_1 : S\alpha \\
\mathcal{P}(\lceil n \rceil) & = & \textbf{return } \cdot \vdash \lceil n \rceil : \mathtt{Nat} \\
\mathcal{P}(\mathit{succ}\ e) & = & \textbf{let } \Gamma \vdash e : \tau \leftarrow \mathcal{P}(e) \textbf{ in} \\
 & & \textbf{let } S \leftarrow \mathcal{U}(\tau \doteq \mathtt{Nat}) \textbf{ in} \\
 & & \textbf{return } \Gamma \vdash \mathit{succ}\ e : \mathtt{Nat}
\end{array}
$$