Simple Types and Type Inference

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Based on notes from Arvind

September 27, 2011
Why Types

let
  f x = if x then 5 else 2
in
  f 5+1

let
  f x = if x then 5 else 2
in
  f 6

let
  f x = if x then 5 else 2
in
  if 6 then 5 else 2
What to do in this situation?

• Options
  1) Leave it up to the implementation
     • that’s the C approach
     • is it a good idea?
  2) Provide a mechanism to identify and rule out such “bad” programs
     • programs can only run if you can prove they will execute to completion according to the semantics of the language
     • type systems will allow us to do this!
  3) Prescribe correct behavior for every program
     • untyped $\lambda$-calculus works like this
     • do any practical languages do this?
     • type systems are useful in this situation too.
Self-application and Paradoxes

Self application, i.e., \((x \ x)\) is dangerous.

Suppose:
\[
u \equiv \lambda y. \text{if } (y \ y) = a \text{ then } b \text{ else } a
\]

What is \((u \ u)\)?
\[
(u \ u) \rightarrow \text{if } (u \ u) = a \text{ then } b \text{ else } a
\]

Contradiction!!!

This was one of the original motivations for types.
What is a type system

• Narrow View
  - It’s a mechanism for ensuring that variables only take values from predefined sets
    • Ex. Integers, Strings, Characters
  - A mechanism for avoiding unchecked errors
    • by ruling out programs with undefined behaviors
    • by specifying how a program should fail (eg. NullPointerException)

• Expansive View
  - It’s a light-weight proof system and annotation mechanism for efficiently checking for a specific property of interest
  - Address bugs that go beyond corner-cases in the semantics
    • Information flow violations
    • deadlocks
    • etc, etc, etc
What are Types?

• A method of classifying objects (values) in a language

\[ x :: \tau \]

says object \( x \) has type \( \tau \) or object \( x \) belongs to a type \( \tau \)

• \( \tau \) denotes a set of values.

*This notion of types is different from types in languages like C, where a type is a storage class specifier.*
Type Correctness

- If \( x :: \tau \) then only those operations that are \textit{appropriate} to set \( \tau \) may be performed on \( x \).

- A program is \textit{type correct} if it never performs a wrong operation on an object.
  - Add an \textit{Int} and a \textit{Bool}
  - Head of an \textit{Int}
  - Square root of a \textit{list}
Type Safety

• A language is *type safe* if only *type correct* programs can be written in that language.

• Most languages are *not* type safe, i.e., have “holes” in their type systems.

  *Fortran:* Equivalence, Parameter passing  
  *Pascal:* Variant records, files  
  *C, C++:* Pointers, type casting

*However, Java, Ada, CLU, ML, Id, Haskell, Bluespec, etc. are type safe.*
Type Declaration vs Reconstruction

- Languages where the user must declare the types
  - CLU, Pascal, Ada, C, C++, Fortran, Java

- Languages where type declarations are not needed and the types are reconstructed at run time
  - Scheme, Lisp

- Languages where type declarations are generally not needed but allowed, and types are reconstructed at compile time
  - ML, Id, Haskell, pH, Bluespec

A language is said to be statically typed if type-checking is done at compile time
Polymorphism

- In a **monomorphic language** like Pascal, one defines a different length function for each type of list.

- In a **polymorphic language** like ML, one defines a polymorphic type (list t), where t is a type variable, and a single function for computing the length.

- Haskell and most modern functional languages have polymorphic types and follow the Hindley-Milner type system.

Simple types = Non polymorphic types

more on polymorphic types – next time …
Formalizing a Type System
Formalizing a type system

- The type system is almost never orthogonal to the semantics of the language
  - The types in a program can affect its behavior (e.g. operator overloading)

- We don’t define the type system in isolation, we define a typed *language* including definitions of
  - The syntax
  - dynamic semantics (e.g. operational semantics)
  - static semantics
    - also known as typing rules
    - describe how types are assigned to elements in a program
  - type soundness argument
    - describe the relationship between static and dynamic semantics
Basic notation

• The type system assigns types to elements in the language
  - basic notation: \( e : T \) (\( e \) is of type \( T \))
  - What is the type of:
    \[
    5
    \]
  - The types of some elements depends on the environment

• The types of some elements depends on the environment
  - basic notation \( \Gamma \vdash e : T \)
    (Given environment \( \Gamma \), we can derive that \( e \) is of type \( T \))
  - An environment associates types with free variables
  - This is called a Judgment
  - Ex.
    \[
    x : \text{int}, y : \text{int} \vdash x + y : \text{int}
    \]
Static Semantics

• Typing rules
  – Typing rules tell us how to derive typing judgments
  – Very similar to derivation rules in Big Step OS

\[
\frac{}{\text{premises}} \quad \Gamma \vdash \text{Judgment}
\]

• Ex. Language of Expressions

\[
\begin{align*}
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} & \quad \frac{\Gamma \vdash N : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \\
\end{align*}
\]
Ex. Language of Expressions

\[
\begin{align*}
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} & \quad \frac{\Gamma \vdash N : \text{int}}{\Gamma \vdash e_1 : \text{int}} \quad \frac{\Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \\
\end{align*}
\]

- Show that the following Judgment is valid

\[
x : \text{int}, y : \text{int} \vdash x + (y + 5) : \text{int}
\]

\[
\begin{align*}
\frac{x : \text{int}, y : \text{int} \vdash x : \text{int}}{x : \text{int}, y : \text{int} \vdash x : \text{int}} & \quad \frac{x : \text{int}, y : \text{int} \vdash (y + 5) : \text{int}}{x : \text{int}, y : \text{int} \vdash x + (y + 5) : \text{int}} \\
\end{align*}
\]
Simply Typed \( \lambda \) Calculus (\( \mathbf{F}_1 \))

- **Basic Typing Rules**

\[
\begin{align*}
& x : \tau \in \Gamma \\
\implies & \quad \Gamma \vdash x : \tau \\
& \Gamma, x : \tau_1 \vdash e : \tau_2 \\
\implies & \quad \Gamma \vdash (\lambda x : \tau_1 \ e) : \tau_1 \to \tau_2 \\
& \Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau' \\
\implies & \quad \Gamma \vdash e_1 e_2 : \tau
\end{align*}
\]

- **Extensions**

\[
\begin{align*}
& \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\implies & \quad \Gamma \vdash e_1 + e_2 : \text{int} \\
& \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\implies & \quad \Gamma \vdash e_1 = e_2 : \text{bool}
\end{align*}
\]

\[
\begin{align*}
& \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau \\
\implies & \quad \Gamma \vdash \text{if } e \text{ then } e_t \text{ else } e_f : \tau
\end{align*}
\]
Example

• Is this a valid typing judgment?

\[ \vdash (\lambda x: \text{bool} \ \lambda y: \text{int} \ \text{if } x \text{ then } y \text{ else } y + 1): \text{bool} \rightarrow \text{int} \rightarrow \text{int} \]

• How about this one?

\[ \vdash (\lambda x: \text{int} \ \lambda y: \text{bool} \ x + y): \text{int} \rightarrow \text{bool} \rightarrow \text{int} \]
Example

- What’s the type of this function?

\[(\lambda f. \lambda x. \text{if } x = 1 \text{ then } x \text{ else } (f \ f \ (x-1)) \ast x)\]

- Hint: This IS a trick question
Simply Typed $\lambda$ Calculus ($F_1$)

- We have defined a really strong type system on $\lambda$-calculus
  - It’s so strong, it won’t even let us write non-terminating computation
  - We can actually prove this!
\(\lambda\)-calculus with Constants & Letrec

\[
E ::= x \mid \lambda x. E \mid E \ E \\
| \text{Cond} (E, E, E) \\
| PF_k(E_1, \ldots, E_k) \\
| CN_0 \\
| CN_k(E_1, \ldots, E_k) \mid CN_k(SE_1, \ldots, SE_k) \\
| \text{let} \ S \ \text{in} \ E
\]

\[
PF_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots \\
PF_2 ::= + \mid \ldots \\
CN_0 ::= \text{Number} \mid \text{Boolean} \\
CN_2 ::= \text{cons} \mid \ldots
\]

Statements
\[
S ::= \epsilon \mid x = E \mid S; S
\]

There are no types in the syntax of the language!
A Simple Type System

**Types**
\[ \tau ::= i \mid t \mid \tau_1 \rightarrow \tau_2 \]

**Type Environments**
\[ \text{TE ::= Identifiers } \rightarrow \text{Types} \]

base types

- type variables

- Function types

int, bool, ...
Type Inference Preliminaries

• What does it mean for two types $\tau_a$ and $\tau_b$ to be equal?
  - **Structural Equality**
    Suppose $\tau_a = \tau_1 \rightarrow \tau_2$
    $\tau_b = \tau_3 \rightarrow \tau_4$
    Is $\tau_a = \tau_b$?  
    iff $\tau_1 = \tau_3$ and $\tau_2 = \tau_4$

• Can two types be made equal by choosing appropriate substitutions for their type variables?
  - **Robinson’s unification algorithm**
    Suppose $\tau_a = t_1 \rightarrow \text{Bool}$
    $\tau_b = \text{Int} \rightarrow t_2$
    Are $\tau_a$ and $\tau_b$ unifiable?  
    if $t_1 = \text{Int}$ and $t_2 = \text{Bool}$

    Suppose $\tau_a = t_1 \rightarrow \text{Bool}$
    $\tau_b = \text{Int} \rightarrow \text{Int}$
    Are $\tau_a$ and $\tau_b$ unifiable?  
    No
Simple Type Substitutions

needed to define type unification

<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau ::= \iota$</td>
</tr>
<tr>
<td>$\mid ; t$</td>
</tr>
<tr>
<td>$; \mid ; \tau_1 \rightarrow \tau_2$</td>
</tr>
</tbody>
</table>

A substitution is a map

$S : \text{Type Variables} \rightarrow \text{Types}$

$S = [\tau_1 / t_1, \ldots, \tau_n / t_n]$

$\tau' = S \tau$ \hspace{1cm} $\tau'$ is a *Substitution Instance of $\tau$

Example:

$S = [(t \rightarrow \text{Bool}) / t_1]$

$S (t_1 \rightarrow t_1) = (t \rightarrow \text{Bool}) \rightarrow (t \rightarrow \text{Bool})$ ?

Substitutions can be *composed*, i.e., $S_2 S_1$

Example:

$S_1 = [(t \rightarrow \text{Bool}) / t_1] ; S_2 = [\text{Int} / t]$

$S_2 S_1 (t_1 \rightarrow t_1) = S_2 ((t \rightarrow \text{Bool}) \rightarrow (t \rightarrow \text{Bool}))$

$= (\text{Int} \rightarrow \text{Bool}) \rightarrow (\text{Int} \rightarrow \text{Bool})$ ?
Unification
An essential subroutine for type inference

Unify(τ₁, τ₂) tries to unify τ₁ and τ₂ and returns a substitution if successful

def Unify(τ₁, τ₂) =
    case (τ₁, τ₂) of
    (τ₁, t₂) = [τ₁ / t₂] provided t₂ ∉ FV(τ₁)
    (t₁, τ₂) = [τ₂ / t₁] provided t₁ ∉ FV(τ₂)
    (ι₁, ι₂) = if (eq? ι₁ ι₂) then [ ]
               else fail
    (τ₁₁ -> τ₁₂, τ₂₁ -> τ₂₂)
    = let S₁ = Unify(τ₁₁, τ₂₁)
       S₂ = Unify(S₁(τ₁₂), S₁(τ₂₂))
       in S₂ S₁
    otherwise = fail

Does the order matter? No
Type Inference

- Type inference is typically presented in two different forms:
  - *Type inference rules:* Rules define a way to deduce the type of each expression in terms of its environment and the types of its subexpressions
    - Clean and concise; needed to study the semantic properties, i.e., soundness, of the type system
  - *Type inference algorithm:* Needed by the compiler writer to deduce the type of each subexpression or to deduce that the expression is ill typed.

- Sometimes it is difficult to derive an inference algorithm for a given set of type inference rules.
Type Inference Rules

Typing: \( \text{TE} \vdash e : \tau \)

Suppose we want to assert (prove) that given some type environment \( \text{TE} \), the expression \((e_1 \ e_2)\) has the type \( \tau' \).

Then it must be the case that the same \( \text{TE} \) implies that there exists some a type \( \tau \), s.t. \( e_1 \) has type \( \tau \rightarrow \tau' \), and \( e_2 \) has the type \( \tau \).

Such an inference rule can be written as:

\[
\begin{array}{c}
\text{(App)} \\
\hline
\text{TE} \vdash e_1 : \tau \rightarrow \tau' \\
\text{TE} \vdash e_2 : \tau \\
\hline
\text{TE} \vdash (e_1 \ e_2) : \tau'
\end{array}
\]
# Simple Type Inference Rules

- **Typing:**
  \[ \text{TE} \vdash e : \tau \]

- **App**
  \[
  \begin{align*}
  \text{TE} & \vdash e_1 : \tau \rightarrow \tau' \\
  \text{TE} & \vdash e_2 : \tau \\
  \text{TE} & \vdash (e_1 \ e_2) : \tau'
  \end{align*}
  \]

- **Abs**
  \[
  \begin{align*}
  \text{TE} + \{x : \tau\} & \vdash e : \tau' \\
  \text{TE} & \vdash \lambda x. e : \tau \rightarrow \tau'
  \end{align*}
  \]

- **Var**
  \[
  \begin{align*}
  (x : \tau) & \in \text{TE} \\
  \text{TE} & \vdash x : \tau
  \end{align*}
  \]

- **Const**
  \[
  \text{typeof}(c) = \tau \\
  \text{TE} & \vdash c : \tau
  \]

- **Let**
  \[
  \begin{align*}
  \text{TE} + \{x : \tau\} & \vdash e_1 : \tau \\
  \text{TE} + \{x : \tau\} & \vdash e_2 : \tau' \\
  \text{TE} & \vdash (let \ x = e_1 \ in \ e_2) : \tau'
  \end{align*}
  \]
Simple Type Inference Rules cont

(Cond)

\[
\begin{align*}
&\text{TE} \vdash e_B : \text{bool} \quad \text{TE} \vdash e_1 : \tau \\
&\text{TE} \vdash e_2 : \tau \\
&\text{TE} \vdash \text{Cond}(e_B, e_1, e_2) : \tau \\
\end{align*}
\]

one such rule for each PF

(PF)

\[
\begin{align*}
&\text{TE} \vdash e_1 : \text{int} \\
&\text{TE} \vdash e_2 : \text{int} \\
&\text{TE} \vdash (e_1 + e_2) : \text{int} \\
\end{align*}
\]

one set of such rules for each data structure type

(CN)

\[
\begin{align*}
&\text{TE} \vdash e_1 : \tau_1 \\
&\text{TE} \vdash e_2 : \tau_2 \\
&\text{TE} \vdash \text{CN}(e_1, e_2) : \text{CN}(\tau_1, \tau_2) \\
\end{align*}
\]

(Pro)

\[
\begin{align*}
&\text{TE} \vdash e : \text{CN}(\tau_1, \tau_2) \\
&\text{TE} \vdash \text{Prj}_1(e) : \tau_1 \\
\end{align*}
\]
Simple Inference Algorithm

\[ W(TE, e) \] returns \((S, \tau)\) such that \(S (TE) \vdash e : \tau\)

The type environment \(TE\) records the most general type of each identifier while the substitution \(S\) records the changes in the type variables.

\[
\text{Def } W(TE, e) = \\
\text{Case } e \text{ of} \\
x = \ ...
\lambda x.e = \ ...
(e_1 \ e_2) = \ ...
let x = e_1 in e_2 = \ ...
\text{Plus } (e_1, e_2) = \ ...
...
\]

September 27, 2011

L05-29
Simple Inference Algorithm (cont-1)

\[ W(\text{TE}, e) = \]

\[ \text{Case } e \text{ of} \]

\[ c = (\{\}, \text{Typeof}(c)) \]
\[ x = \begin{cases} \text{if } (x \notin \text{Dom}(\text{TE})) \text{ then Fail} \\
\text{else let } \tau = \text{TE}(x); \\
\text{in } (\{\}, \tau) \end{cases} \]
\[ \lambda x. e = \begin{cases} \text{let } (S_1, \tau_1) = W(\text{TE} + \{ x : u \}, e) \\
\text{in } (S_1, S_1(u) -> \tau_1) \end{cases} \]
\[ (e_1 e_2) = \begin{cases} \text{let } (S_1, \tau_1) = W(\text{TE}, e_1); \\
(S_2, \tau_2) = W(S_1(\text{TE}), e_2); \\
S_3 = \text{Unify}(S_2(\tau_1), \tau_2 -> u); \\
\text{in } (S_3 S_2 S_1, S_3(u)) \end{cases} \]

\[ \text{let } x = e_1 \text{ in } e_2 \]
\[ = \begin{cases} \text{let } (S_1, \tau_1) = W(\text{TE} + \{ x : u \}, e_1); \\
S_2 = \text{Unify}(S_1(u), \tau_1); \\
(S_3, \tau_2) = W(S_2 S_1(\text{TE}) + \{ x : \tau_1 \}, e_2); \\
\text{in } (S_3 S_2 S_1, \tau_2) \end{cases} \]
Simple Inference Algorithm (cont-1)

\[ \text{Cond}(e_B, e_1, e_2) = \begin{aligned} \text{let } & (S_B, \tau_B) = W(TE, e_B); \\ & (S_1, \tau_1) = W(S_B(TE), e_1); \\ & (S_2, \tau_2) = W(S_1 S_B(TE), e_2); \\ & S_3 = \text{Unify}(\tau_B, \text{bool}); \\ & S_4 = \text{Unify}(\tau_1, \tau_2); \\ \text{in } & (S_4 S_3 S_2 S_1 S_B, S_4 S_3 \tau_2) \end{aligned} \]

\[ \text{Plus}(e_1, e_2) = \begin{aligned} \text{let } & (S_1, \tau_1) = W(TE, e_1); \\ & (S_2, \tau_2) = W(S_1(TE), e_2); \\ & S_3 = \text{Unify}(\tau_1, \text{int}); \\ & S_4 = \text{Unify}(\tau_2, \text{int}); \\ \text{in } & (S_4 S_3 S_2 S_1, \text{int} \to \text{int}) \end{aligned} \]

\[ \text{CN}(e_1, e_2) = \begin{aligned} \text{let } & (S_1, \tau_1) = W(TE, e_1); \\ & (S_2, \tau_2) = W(S_1(TE), e_2); \\ \text{in } & (S_2 S_1, \text{CN}(\tau_1, \tau_2)) \end{aligned} \]

\[ \text{Prj}_1(e) = \begin{aligned} \text{let } & (S, \tau) = W(TE, e); \\ & S_1 = \text{Unify}(\tau, \text{CN}(u_1, u_2)); \\ \text{in } & (S_1 S S_1(u1)) \end{aligned} \]

Each Prj is tied to some CN and \( u_1 \) and \( u_2 \) must be fresh type variables.
Type Inference : Example

\[\text{let } f = \lambda n . \text{cond } ((\text{eq0 } n), 1, n^* (f \ (\text{pred } n)) ) \\]

\[\text{in } f\]

\[W(\emptyset, A) = ( [ ] , \text{Int} \rightarrow \text{Int} )\]

\[W(\{f : u_1\}, \lambda n. B) = ([\text{Int} \rightarrow \text{Int} / u_1], \text{Int} \rightarrow \text{Int})\]

\[W(\{f : u_1, n : u_2\}, B) = ([\text{Int} \rightarrow \text{Int} / u_1, \text{Int} / u_2], \text{Int})\]

\[W(\{f : u_1, n : u_2\}, (\text{eq0 } n)) = ([\text{Int} / u_2], \text{Bool})\]

\[\text{Unify}(u_2, \text{Int}) = [\text{Int} / u_2]\]

\[W(\{f : u_1, n : \text{Int}\}, 1) = ( [ ], \text{Int})\]

\[W(\{f : u_1, n : \text{Int}\}, n^*(f \ (\text{pred } n))) = ([\text{Int} \rightarrow \text{Int} / u_1], \text{Int})\]

\[W(\{f : u_1, n : \text{Int}\}, n) = ( [ ], \text{Int})\]

\[W(\{f : u_1, n : \text{Int}\}, f \ (\text{pred } n)) = ([\text{Int} \rightarrow u_3 / u_1], u_3)\]

\[\text{Unify}(u_1, \text{Int} \rightarrow u_3) = [\text{Int} \rightarrow u_3 / u_1]\]

\[\text{Unify}(\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \text{Int} \rightarrow u_3 \rightarrow \text{Int}) = [\text{Int} / u_3]\]

\[W(\{f : \text{Int} \rightarrow \text{Int}\}, f) = ( [ ], \text{Int} \rightarrow \text{Int})\]
Soundness

• The proposed type system is said to be *sound* if $e : \tau$ then $e$ indeed evaluates to a value in $\tau$.

• To prove soundness, we need to show two properties

  Preservation: $\text{TE} \vdash e : \tau$ and $(e \rightarrow e') \Rightarrow \text{TE} \vdash e' : \tau$

  Progress: $\text{TE} \vdash e : \tau \Rightarrow$
  Either $e$ is a value or $\exists e'$ s.t. $(e \rightarrow e')$

*Proofs in the next lecture*
Some observations

• A type system restricts the class of programs that are considered “legal”
• It is possible a term in the untyped $\lambda$-calculus may be reducible to a value but may not be typeable in a particular type system

\[
\text{let}
\begin{align*}
\text{id} & = \lambda x. x \\
\text{in}
\end{align*}
\]
\[
\text{... (id True) ... (id 1) ...}
\]

*This term is not typeable in the simple type system we have discussed so far. However, it is typeable in the Hindley-Milner system*