Type Systems II

COMP 311
Rice University
Houston, Texas

Fall 2003

Copyright 2000, Robert Cartwright, all rights reserved. Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Type Systems: Simply Typed $\lambda$-calculus

Realistic core language suitable for type-checking

$$M ::= c \mid x \mid (M M \ldots M) \mid (\lambda \, x_1 : t_1 \ldots x_k : t_k \, . \, M) \mid \text{if } M \text{ then } M$$

$$t ::= b \mid t_1 \ldots t_k \ldots$$

$b \in B$ (a set of base types including bool, )

$c \in C$ (a set of constants including true, false), $c \in C$ has a given type $\tau(c)$

$x \in V$ (a set of variables), variables in $(x_1 : t_1 \ldots x_k : t_k)$ must be distinct

Typing rules:

$$\Gamma \vdash x : \tau$$

$$\Gamma \vdash c : \tau(c) \quad \left[ \begin{array}{l} \tau(c(\text{true})) = \text{bool}, \quad \tau(c(\text{false})) = \text{bool} \end{array} \right]$$

$$\Gamma \vdash M : s_1 \ldots t_k \ldots, \Gamma \vdash N_1 : s_1, \ldots, \Gamma \vdash N_k : s_k \quad \text{(app rule)}$$

$$\Gamma \vdash (M \, N_1 \ldots N_k) : \tau$$

$$\Gamma, \, x_1 : s_1, \ldots, x_k : s_k \vdash M : \tau \quad \text{(abs rule)}$$

$$\Gamma \vdash (\lambda \, x_1 : s_1 \ldots x_k : s_k \, . \, M) : s_1 \ldots s_k$$

$$\Gamma \vdash M_1 : \text{bool}, \quad \Gamma \vdash M_2 : \tau, \quad \Gamma \vdash M_3 : \tau \quad \text{(if rule)}$$

$$\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau$$
Type Systems: Sample Typing Proof

Show $\emptyset \mid ((\emptyset \, f : \text{bool} \rightarrow \text{bool} \cdot (\emptyset \, x : \text{bool} \cdot (f(fx)))) \, (\emptyset \, x : \text{bool} \cdot x)) : \text{bool} \rightarrow \text{bool}$

Tree 1: $f : \text{bool} \rightarrow \text{bool}, \, x : \text{bool} \mid f : \text{bool} \rightarrow \text{bool}, \, f : \text{bool} \rightarrow \text{bool}, \, x : \text{bool} \mid x : \text{bool}$

$f : \text{bool} \rightarrow \text{bool}, \, x : \text{bool} \mid (fx) : \text{bool}$

Tree 2:

$f : \text{bool} \rightarrow \text{bool}, \, x : \text{bool} \mid f : \text{bool} \rightarrow \text{bool}, \, \text{Tree 1}$

$f : \text{bool} \rightarrow \text{bool}, \, x : \text{bool} \mid (fx) : \text{bool}$

$f : \text{bool} \rightarrow \text{bool} \mid (\emptyset \, x : \text{bool} \cdot (f(fx)) : \text{bool} \rightarrow \text{bool}$

$\emptyset \mid ((\emptyset \, f : \text{bool} \rightarrow \text{bool} \cdot (\emptyset \, x : \text{bool} \cdot (f(fx)))) : (\text{bool} \rightarrow \text{bool}) \rightarrow (\text{bool} \rightarrow \text{bool})$}

Tree 3:

$x : \text{bool} \mid x : \text{bool}$

$\emptyset \mid (\emptyset \, x : \text{bool} \cdot x) : \text{bool} \rightarrow \text{bool}$

Tree 4:

Tree 2, Tree 3

$\emptyset \mid ((\emptyset \, f : \text{bool} \rightarrow \text{bool} \cdot (\emptyset \, x : \text{bool} \cdot (f(fx)))) \, (\emptyset \, x : \text{bool} \cdot x) : \text{bool} \rightarrow \text{bool}$
Is \(((\lambda f : s . (\lambda x : s . (f (f x)))) (\lambda x : s . x))\) typable? What is its principal type?

Tree1:

\[
\begin{align*}
f & : s, x : t \\
f & : s, x : t \quad | \quad x : t \\
f & : s_1 \rightarrow s_2, x : s_1 & \mid (f x) : s_2 \\
\end{align*}
\]

Tree2:

\[
\begin{align*}
f & : s_1 \rightarrow s_2, x : s_1 & \mid f : s_1 \rightarrow s_2, \quad \text{Tree1} \\
f & : s_1 \rightarrow s_1, x : s_1 & \mid (f (f x)) : s_1 \\
f & : s_1 \rightarrow s_1 & \mid ((\lambda f : s_1 \rightarrow s_1 . (\lambda x : s_1 . (f (f x)))) : (s_1 \rightarrow s_1) \rightarrow (s_1 \rightarrow s_1)) \\
\end{align*}
\]

Tree3:

\[
\begin{align*}
x & : u \\
\end{align*}
\]

\[
\begin{align*}
\emptyset & \mid (\lambda x : u . x) : u \\
\end{align*}
\]

Tree4:

\[
\begin{align*}
\emptyset & \mid ((\lambda f : s_1 \rightarrow s_1 . (\lambda x : s_1 . (f (f x)))) (\lambda x : s_1 . x)) : s_1 \rightarrow s_1 \\
\end{align*}
\]
Type Systems: Formalizing Polymorphism

One extension to the simply typed language:

(let x := M in M)

where let is recursive (scope of x includes the right hand side of definition of x)

Five extensions to our simple type system:

- Type variables: $\alpha_1, \alpha_2, ...$
- Type schemes: $\alpha ::= \alpha_1 ... \alpha_k . \alpha$ where $\alpha$ is a type. Type schemes are not types!
- Type environments (symbol tables) can contain type schemes; so can the table $\alpha$.
- Additional inference rules:

\[
\Gamma, x: \alpha_1 ... \alpha_k . \alpha |\vdash x: \text{OPEN}(\alpha_1 ... \alpha_k . \alpha_1, ..., \alpha_k) \quad \text{(instantiation)}
\]

\[
[\alpha_1, ..., \alpha_k \text{ are types}]
\]

\[
\Gamma, x: \alpha |\vdash M: \alpha, \quad \Gamma, x: \text{CLOSE}(\alpha_1, \alpha) |\vdash N: \alpha \quad \text{(letpoly)}
\]

\[
\vdash (\text{let} \ x := M \ \text{in} \ N): \alpha
\]

- Additional axiom:

\[
\vdash c: \text{OPEN}(\square(c), \alpha_1, ..., \alpha_k) \quad \text{where} \ c \in C \text{ and } \square(c) = \alpha_1 ... \alpha_k . \alpha
\]
Type Systems: Formalizing Polymorphism continued

Notes

- The notation OPEN(a_1 ... a_k. t, b_1, ..., b_k) means convert the type scheme a_1 ... a_k. t to the type t' where t' is t with type variables a_1 ... a_k replaced by “fresh” type variables b_1, ..., b_k.

- The notation CLOSE(t_1, G) means convert the type t_1 to the type scheme a_1 ... a_k. t_1 where a_1, ..., a_k are the type variables that appear in t_1 but not G.

Intuition:

- Polymorphism abbreviates brute force replication of the definition introduced in a let. The new type variables that appear in the type of M (rhs of the let binding) are arbitrary. The instantiation and polylet rules lets us adapt a symbolic type for M to each of the specific uses of x (the lhs of the let binding) in N (the body of the let).

- The rhs side of the let binding cannot use x polymorphically because such usage is inconsistent with the fact that polymorphic let is an abbreviation mechanism!
Consider a functional language with polymorphic lists. The operations on polymorphic lists include the binary function cons, unary functions first and rest, and the constant null. A sample program in this language is:

\[
\begin{align*}
\text{(let length := (\lambda x (if (null? x) then 0 else (+ 1 (length (rest x)))))) in} \\
(+ (length (cons 1 null)) (length (cons true null)))
\end{align*}
\]

Can we type it?

**Sketch:** \(\emptyset \mid \text{length: } \text{list \to int}\)

Use letpoly rule to add length to type environment with polymorphic type.

Instantiate it twice: once for bool and once for int.
**Type Systems: Sample Polymorphic Type Reconstruction**

**Claim:** \( \emptyset \) | (let length := (\( \square \). (if (null? x) then 0 else (+ 1 (length (rest x)))))) in (+ (length (cons 1 null)) (length (cons true null)))): int

**Tree1:**
- length: \( \square \), x: \( \square \) | null?: \( \square \)-list\( \square \) bool, length: \( \square \), x: \( \square \) | x: \( \square \)
- length: \( \square \), x: \( \square \)-list | (null? x): bool

**Tree2:**
- length: \( \square \), x: \( \square \)-list | rest: \( \square \)-2-list \( \square \)-2-list, length: \( \square \), x: \( \square \)-list | x: \( \square \)-list
- length: \( \square \), x: \( \square \)-list | (rest x): \( \square \)-list
- length: \( \square \)-list \( \square \)-2-list \( \square \)-2-list | (length (rest x)): \( \square \)
- length: \( \square \)-list \( \square \)-int, x: \( \square \)-list | (+ 1 length(rest x)): int
- \( \square \) = \( \square \)-list \( \square \)

**Tree3:**
- Tree1, length: \( \square \)-list \( \square \)-int, x: \( \square \)-list | 0:int, Tree2
- length: \( \square \)-list \( \square \)-int, x: \( \square \)-list | if (null? x) then 0 else (+ 1 (length (rest x)))) : int
- length: \( \square \)-list \( \square \)-int | \( \square \). if (null? x) then 0 else (+ 1 (length (rest x)))) : \( \square \)-list\( \square \) int

**Tree4:**
- length: \( \square \)-\( \square \)-list \( \square \)-int | cons: \( \square \)-3-list \( \square \)-3-list, length: \( \square \)-\( \square \)-list \( \square \)-int | 1:int, length: \( \square \)-\( \square \)-list \( \square \)-int | null: \( \square \)-4-list
- \( \square \) = \( \square \)-3-list \( \square \)
- length: \( \square \)-\( \square \)-list \( \square \)-int | (cons 1 null): \( \square \)-list

**Tree5:**
- length: \( \square \)-\( \square \)-list \( \square \)-int | length: \( \square \)-5-list \( \square \)-int, Tree4
- length: \( \square \)-\( \square \)-list \( \square \)-int | (length (cons 1 null)): int

**Tree6:**
- length: \( \square \)-\( \square \)-list \( \square \)-int | cons: \( \square \)-6-list \( \square \)-6-list, length: \( \square \)-\( \square \)-list \( \square \)-int | true: \( \square \)-bool, length: \( \square \)-\( \square \)-list \( \square \)-int | null: \( \square \)-7-list
- length: \( \square \)-\( \square \)-list \( \square \)-int | length: \( \square \)-list \( \square \)-int, length: \( \square \)-\( \square \)-list \( \square \)-int | (cons true null): \( \square \)-bool
- length: \( \square \)-\( \square \)-list \( \square \)-int | (length (cons true null)): \( \square \)-list

**COMP 311**
Type Systems: Sample Polymorphic Type Reconstruction, cont.

Tree7: Tree5, Tree6, length: list int | +: int int int
      length: list int | (+ (length (cons 1 null)) (length (cons true null))): int

Tree8: Tree3, Tree7
Ø | (let length := (λx. if (null? x) then 0 else (+ 1 (length (rest x)))) in (+ (length (cons 1 null)) (length (cons true null)))): int
Type Systems: Coping with Imperatitivity

• If replicating the let definition
  \( x := M \)
  (renaming the defined variable \( x \)) for each use of \( x \) in \( M \) does not preserve the meaning of programs, then programs written using Milner style polymorphism may not be type correct. In an imperative language, this phenomenon can happen in several ways. First, the evaluation of \( M \) may have side effects. Second, the value of \( x \) may allocate mutable storage which is shared when

  \( x := M \)

  is a single definition but split (among the various type instantiations) when the definition is replicated. In an imperative language this splitting can be detected by mutating allocated storage.

• To avoid this problem, we can restrict \( M \) to a form that guarantees replication does not change the meaning of programs. This restricted form prevents \( M \) from performing side-effects and from allocating shared mutable storage.

• If we restrict \( M \) to a syntactic value (a constant, variable, or λ-expression) then no side effect or sharing of mutable storage can occur. The most modern languages that use Hindley-Milner polymorphism use this restriction. Standard ML uses a much more complicated and less useful restriction (that is incomparable to the syntactic value test).
Type Systems: Coping with Imperativity

- We can incorporate the value restriction in our type system by refining the definition of CLOSE so that it does not generalize the free type variables when then rhs of a definition is not a syntactic value.

- Let us extend our polymorphic -calculus language to imperative form in the same way that we did for Jam by adding the type constant \texttt{unit}, the unary type constructor \texttt{ref}, the unary operations \texttt{ref: * \rightarrow * ref} and \texttt{!: * ref \rightarrow * }, and the binary operation \texttt{\& : * ref \rightarrow * ref \rightarrow unit}.

- The following polymorphic imperative program generates a run-time type error even though it can be statically typed checked using our rules omitting the value restriction.

  
  \begin{verbatim}
  let x := ref null
  in { (\& x cons(true,null));
       (+ (! x) 1) }
  \end{verbatim}

  In the absence of the value restriction, this program is typable, because \texttt{x} has polymorphic type \texttt{* ref}, enabling each occurrence of \texttt{x} in the \texttt{let} body to be separately typed. Hence, the first occurrence of \texttt{x} has type \texttt{bool ref} while the second has type \texttt{int ref}.

- The value restriction prevents polymorphic generalization in this case, preserving type soundness.