Type Systems II

COMP 311
Rice University
Houston, Texas

Fall 2003

Type Systems: Simply Typed □-calculus

Realistic core language suitable for type-checking

M ::= c | x | (M M … M) | (□ x:□… x:□. M) | **if** M **then** M

□::= b | □□… □□□ □

b □ B (a set of base types including **bool**,)

c □ C (a set of constants including **true**, **false**), c □ C has a given type □(c)

x □ V (a set of variables), variables in (x:□… x:□) must be distinct

Typing rules:

$$\frac{}{\square, x{:}\square \mid\square\; x{:}\ \square}$$

$\square \mid\square\; c{:}\ \square(c)$ [□(**true**) = **bool**, □(**false**) = **bool**]

$$\frac{\square \mid\square\; M{:}\square_1 \square \dots \square \square_k \square\ \ \square,\quad \square \mid\square\; N_1{:}\square_1,\ \dots\ ,\ \square \mid\square\; N_k{:}\square_k}{\square \mid\square\; (M\ N_1 \dots N_k){:}\ \square.}$$ (app rule)

$$\frac{\square,\ x_1{:}\square_1, \dots, x_k{:}\square_k \mid\square\; M{:}\ \square.}{\square \mid\square\; (\square\, x_1{:}\square_1 \dots x_k{:}\square_k\ .\ M){:}\ \square_1 \square \dots \square \square_k \square\ \ \square}$$ (abs rule)

$$\frac{\square \mid\square\; M_1{:}\textbf{bool},\ \square \mid\square\; M_2{:}\square,\ \square \mid\square\; M_3{:}\square}{\square \mid\text{-}\ \textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3 : \square}$$ (if rule)

Type Systems : Sample Typing Proof

Show $\emptyset \mid ((\Box\, f{:}\text{bool}\Box\ \text{bool} .\ (\Box\, x{:}\text{bool} .\ (f\,(f\,x))))\ (\Box\, x{:}\text{bool} .\ x)) :\ \text{bool}\Box\ \text{bool}$

Tree1:

$$\frac{f{:}\text{bool}\Box\ \text{bool},\ x{:}\text{bool} \mid f{:}\text{bool}\Box\ \text{bool}, \quad f{:}\text{bool}\Box\ \text{bool},\ x{:}\text{bool} \mid x{:}\text{bool}}{f{:}\text{bool}\Box\ \text{bool},\ x{:}\text{bool} \mid (f\,x){:}\ \text{bool}}$$

Tree2:

$$\frac{\dfrac{\dfrac{f{:}\text{bool}\Box\ \text{bool},\ x{:}\text{bool} \mid f{:}\text{bool}\Box\ \text{bool}, \quad \text{Tree1}}{f{:}\text{bool}\Box\ \text{bool},\ x{:}\text{bool} \mid (f\,(f\,x)){:}\ \text{bool}}}{f{:}\text{bool}\Box\ \text{bool} \mid (\Box\, x{:}\text{bool} .\ (f\,(f\,x))){:}\ \text{bool}\Box\ \text{bool}}}{\emptyset \mid ((\Box\, f{:}\text{bool}\Box\ \text{bool} .\ (\Box\, x{:}\text{bool} .\ (f\,(f\,x)))) : (\text{bool}\Box\ \text{bool})\ \Box\ (\text{bool}\Box\ \text{bool})}$$

Tree3:

$$\frac{x{:}\text{bool} \mid x{:}\text{bool}}{\emptyset \mid (\Box\, x{:}\text{bool} .\ x)) :\ \text{bool}\Box\ \text{bool}}$$

Tree4:

$$\frac{\text{Tree2},\ \text{Tree3}}{\emptyset \mid ((\Box\, f{:}\text{bool}\Box\ \text{bool} .\ (\Box\, x{:}\text{bool} .\ (f\,(f\,x))))\ (\Box\, x{:}\text{bool} .\ x)) :\ \text{bool}\Box\ \text{bool}}$$

Type Systems: Sample Type Reconstruction

Is ((□ f :□ . (□ x:□. (f (f x)))) (□ x:□ . x)) typable? What is its principal type?

Tree1:

$$\frac{f : \square, x:\square \mid f :\square, \quad f :\square, x:\square \mid x:\square}{f :\square 1 \;\square\; \square 2, x:\square 1 \mid (f\ x): \square 2} \qquad [\square = \square 1 \;\square\; \square 2 ,\ \ \square=\square 1\ ]$$

Tree2:

$$\cfrac{\cfrac{\cfrac{f : \square 1 \;\square\; \square 2, x: \square 1 \mid f : \square 1 \;\square\; \square 2, \quad \text{Tree1}}{f : \square 1 \;\square\; \square 1, x: \square 1 \mid (f\ (f\ x)): \square 1}}{f : \square 1 \;\square\; \square 1 \mid (\square x:\square 1\ .\ (f\ (f\ x))) : \square 1 \;\square\; \square 1}}{\varnothing \mid ((\square f :\square 1 \;\square\; \square 1.\ (\square x:\square 1\ .\ (f\ (f\ x)))) : (\square 1 \;\square\; \square 1) \;\square\; (\square 1 \;\square\; \square 1)} \qquad [\square 2 = \square 1\ ]$$

Tree3:

$$\frac{x: \square \mid x: \square}{\varnothing \mid (\square x: \square.\ x)) : \square \;\square\; \square}$$

Tree4:

$$\frac{\text{Tree2, Tree3}}{\varnothing \mid ((\square f : \square 1 \;\square\; \square 1.\ (\square x: \square 1\ .\ (f\ (f\ x)))) (\square x: \square 1\ .\ x)) : \square 1 \;\square\; \square 1} \qquad [\square = \square 1]$$

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: □$_1$, □$_2$, ...
- Type schemes: □ ::= □□$_1$ ... □$_k$. □ where □ is a type. Type schemes are not types!
- Type environments (symbol tables) can contain type schemes; so can the table □.
- Additional inference rules:

  □, *x*: □□$_1$ ... □$_k$. □ |□ *x*: OPEN(□□$_1$ ... □$_k$. □, □$_1$, ..., □$_k$) (instantiation)

  [□$_1$, ..., □$_k$ are types]

  $$\frac{□, x{:}□_1 \mid□\ M{:}□_1, \quad □, x{:}\mathrm{CLOSE}(□_1, □) \mid□\ N{:}□}{□ \mid\!-\ (\mathbf{let}\ x := M\ \ \mathrm{in}\ N){:}\ □}$$ (letpoly)

- Additional axiom:

  □ |- *c*: OPEN(□(*c*), □$_1$, ..., □$_k$) where *c* □ *C* and □(*c*) = □□$_1$ ... □$_k$. □

Type Systems: Formalizing Polymorphism continued

Notes

- **The notation OPEN($\square\square_1 \ldots \square_k . \square, \square_1, \ldots, \square_k$) means convert the type scheme $\square\square_1 \ldots \square_k . \square$ to the type $\square'$ where $\square'$ is $\square$ with type variables $\square_1 \ldots \square_k$ replaced by "fresh" type variables $\square_1, \ldots, \square_k$.**
- **The notation CLOSE($\square_1, \square$) means convert the type $\square_1$ to the type scheme $\square\square_1 \ldots \square_k . \square_1$ where $\square_1, \ldots, \square_k$ are the type variables that appear in $\square_1$ but not $\square$.**

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!**

Type Systems: Sample Polymorphic Type Reconstruction

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:

```
    (let length := (λ x (if (null? x) then 0 else (+ 1 (length (rest x))))) in
                          (+ (length (cons 1 null)) (length (cons true null))))
```

Can we type it?

Sketch: $\emptyset$ **| length: α list → 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: ∅ | (**let** length := (□x. (**if** (null? x) **then** 0 **else** (+ 1 (length (rest x))))) **in** (+ (length (cons 1 null)) (length (cons true null)))):**int**

Tree1:

length: □, *x*:□$_1$ | null?:□-**list**□ **bool**, length:□, *x*: □$_1$ | *x*:□$_1$ [□$_1$ = □-**list**]

---

length:□, *x*:□-**list** | (null? *x*):**bool**

Tree2:

length:□, *x*:□-**list** | rest: □$_2$-**list**□ □$_2$-**list**, length:□, *x*:□-**list** | *x*:□-**list** [□$_2$ = □]

---

length:□, *x*:□-**list** | (rest *x*):□-**list**

---

length:□-**list**□ □$_2$, *x*:□-**list** | (length (rest *x*)):□$_2$ [□ = □-**list**□ □2]

---

length:□-**list**□ **int**, *x*:□-**list** | (+ 1 length(rest *x*)):**int** [□$_2$ = **int**]

Tree3:

Tree1, length:□-**list**□ **int**, *x*:□-**list** | 0:**int**, Tree2

---

length:□-**list**□ **int**, *x*:□-**list** | **if** (null? x) **then** 0 **else** (+ 1 (length (rest x)))) : **int**

---

length:□-**list**□ **int** | □x. **if** (null? x) **then** 0 **else** (+ 1 (length (rest x)))) : □-**list**□ **int**

Tree4:

length:□□.□-**list**□ **int** | cons:□$_3$□□$_3$-**list**□ □$_3$-**list**, length:□□.□-**list**□ **int** | 1:**int**, length:□□.□-**list**□ **int** | null:□$_4$-**list** [□$_3$ = □$_4$ = **int**]

---

length:□□.□-**list**□ **int** | (cons 1 null):**int-list**

Tree5:

length:□□.□-**list**□ **int** | length: □$_5$-**list**□ **int**, Tree4 [□$_5$ =**int**]

---

length:□□.□-**list**□ **int** | (length (cons 1 null)): **int**

Tree6:

length:□□.□-**list**□ **int** | cons:□$_6$□□$_6$-**list**□ □$_6$-**list**, length:□□.□-**list**□ **int** | true:**bool**, length:□□.□-**list**□ **int** | null:□$_7$-**list**

---

length:□□.□-**list**□ **int** | length: **int-list**□ **int**, length:□□.□-**list**□ **int** | (cons true null):**bool-list**

---

length:□□.□-**list**□ **int** | (length (cons true null)):**int**

Type Systems: Sample Polymorphic Type Reconstruction, cont.

Tree7:

$$\frac{\text{Tree5,}\quad \text{Tree6,}\quad \text{length:}\forall\alpha.\,\alpha\textbf{-list}\rightarrow\textbf{int} \mid +\text{:}\ \textbf{int}\times\textbf{int}\rightarrow\textbf{int}}{\text{length:}\forall\alpha.\,\alpha\textbf{-list}\rightarrow\textbf{int} \mid \text{(+ (length (cons 1 null)) (length (cons true null))):}\ \textbf{int}}$$

Tree8:

$$\frac{\text{Tree3,}\quad \text{Tree7}}{\emptyset \mid (\textbf{let}\ \text{length := (}\lambda\text{x}\textbf{.}\ (\textbf{if}\ \text{(null? x)}\ \textbf{then}\ \text{0}\ \textbf{else}\ \text{(+ 1 (length (rest x))))))}\ \textbf{in}\ \text{(+ (length (cons 1 null)) (length (cons true null)))):}\textbf{int}}$$

Type Systems: Coping with Imperativity

- 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 unit, the unary type constructor ref, the unary operations ref: * □ * ref and !: * ref □ * , and the binary operation □ : * ref □ * □ 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.**

```
  let x := ref null
   in { (□  x cons(true,null));
        (+ (! x) 1) }
```

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

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