CMSC 631 – Program Analysis and Understanding
Fall 2003

Type Systems

The Need for a Type System

- Consider the (untyped) lambda calculus
  - false = λx.λy.x
  - 0 (Scott) = λx.λy.x
- Everything is encoded as a function
  - So we can easily misuse combinators
    - false 0 if 0 then ... etc...
  - This is no better than assembly language!

What is a Type System?

- A *type system* is some mechanism for distinguishing good programs from bad
  - Good programs = well typed
  - Bad programs = ill typed or not typable
- Examples:
  - 0 + 1 // well typed
  - false 0 // ill-typed: can't apply a boolean
  - 1 + (if true then 0 else false) // ill-typed: can't add boolean to integer

A Definition of Type Systems

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

– Benjamin Pierce, *Types and Programming Languages*

Simply-Typed Lambda Calculus

- e ::= n | x | λx:t.e | e e
  - Functions include the type of their argument
  - We don't really need this, but it will come in handy
- t ::= int | t → t
  - t1 → t2 is a the type of a function that, given an argument of type t1, returns a result of type t2
    - t1 is the *domain*, and t2 is the *range*

Type Judgments

- Our type system will prove *judgments* of the form
  - A ⊢ e : t
  - "In type environment A, expression e has type t"

Type Environments

- A *type environment* is a map from variables to types (a kind of symbol table)
  - ∅ is the empty type environment
    - A closed term e is *well-typed* if ∅ ⊢ e : t for some t
    - We'll abbreviate this as ⊢ e : t
  - A, x:t is just like A, except x now has type t
    - The type of x in A, x:t is t
    - The type of z≠x in A, x:t in the type of z in A
- When we see a variable in a program, we look in the type environment to find its type

Type Rules

$$\frac{}{A \vdash n : int} \qquad \frac{x \in dom(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e1 : t \to t' \quad A \vdash e2 : t}{A \vdash e1\ e2 : t}$$

Example

A = - : int→int

$$\frac{\dfrac{- \in dom(A)}{A \vdash - : int \to int} \qquad A \vdash 3 : int}{A \vdash - 3 : int}$$

Another Example

A = + : int→int→int
B = A, x : int

$$\dfrac{\dfrac{\dfrac{\dfrac{+ \in dom(B)}{B \vdash + : i \to i \to i} \quad \dfrac{x \in dom(B)}{B \vdash x : i}}{B \vdash + x : int \to int} \qquad B \vdash 3 : int}{B \vdash + x\ 3 : int}}{A \vdash (\lambda x{:}int.+ x\ 3) : int \to int} \qquad A \vdash 4 : int$$

$$A \vdash (\lambda x{:}int.+ x\ 3)\ 4 : int$$

We'd usually use infix x + 3

An Algorithm for Type Checking

- Our type rules are deterministic
  - For each syntactic form, only one possible rule
  -
- They define a natural type checking algorithm
  - TypeCheck : type env × expression → type

```
TypeCheck(A, n) = int
TypeCheck(A, x) = if x in dom(A) then A(x) else fail
TypeCheck(A, λx:t.e) = TypeCheck((A, x:t), e)
TypeCheck(A, e1 e2) =
    let t1 = TypeCheck(A, e1) in
    let t2 = TypeCheck(A, e2) in
      if dom(t1) = t2 then range(t1) else fail
```


Semantics

- Here is our semantics, with integers
  - Notice that the last rule requires that e1 is a function
    - The other cases are undefined (i.e., an error)

$$\frac{}{n \to^I n} \qquad \frac{}{(\lambda x.e1) \to^I (\lambda x.e1)}$$

$$\frac{e1 \to^I \lambda x.e \quad e[e2/x] \to^I e'}{e1\ e2 \to^I e'}$$

Semantics with Error

- If we want to make this precise, can add a new term error
  - Invalid programs reduce to error

$$\frac{e1 \rightarrow^{I} n}{e1\ e2 \rightarrow^{I} \text{error}}$$

$$\frac{e1 \rightarrow^{I} \text{error}}{e1\ e2 \rightarrow^{I} \text{error}}$$

Soundness

- Theorem: If e is a closed term and $\vdash e : t$ for some t, then $e \rightarrow^{I} e$ where e is not error
  - Note: I will omit the argument types in the proof
  -
- Lemma: If $\vdash e : t$ and $e \rightarrow^{I} e'$, then $\vdash e' : t$
  - Notice e' cannot be error, because no type rules for it
  - Proof: By induction on the reduction $e \rightarrow^{I} e'$
  - Base case $e = n$ trivial
  - Base case $e = \lambda x.e$ trivial

Soundness (cont'd)

- Induction: Suppose $e = e1\ e2$
  - Then by assumption this type checks by the rule

$$\frac{\vdash e1 : u \rightarrow u' \quad \vdash e2 : u}{\vdash e1\ e2 : u}$$

  - Since $\vdash e1 : u \rightarrow u'$, by induction we have $e1 \rightarrow^{I} e1$ where $\vdash e1' : u \rightarrow u'$. Then looking at the type rules, the only reduction result that could be typed as a function is an expression $\lambda x.e$. But then we can use the reduction rule

$$\frac{e1 \rightarrow^{I} \lambda x.e \quad e[e2\backslash x] \rightarrow^{I} e'}{e1\ e2 \rightarrow^{I} e'}$$

Soundness (cont'd)

- Also, since $\vdash \lambda x.e : u \rightarrow u'$, we must have applied the type rule

$$\frac{x{:}u \vdash e : u'}{\vdash \lambda x.e : u \rightarrow u'}$$

- Then since $x{:}u \vdash e : u'$ and $\vdash e2 : u$, by the substitution lemma (omitted), we know $\vdash e[e2\backslash x] : u'$. Then by induction again, we know that $\vdash e' : u'$, and so we're done.

Product Types (Tuples)

$e ::= \ldots \mid (e, e) \mid \text{fst } e \mid \text{snd } e$

$$\frac{A \vdash e : t \quad A \vdash e : t}{A \vdash (e1, e2) : t \times t}$$

$$\frac{A \vdash e : t \times t}{A \vdash \text{fst } e : t} \qquad \frac{A \vdash e : t \times t}{A \vdash \text{snd } e : t}$$

- Or, maybe, just add functions
  - pair : $t \rightarrow t' \rightarrow t \times t'$
  - fst : $t \times t' \rightarrow t$
  - snd : $t \times t' \rightarrow t'$

Sum Types (Tagged Unions)

$e ::= \ldots \mid \text{inL}_{t2}\ e \mid \text{inR}_{t1}\ e$
$\mid (\text{case } e \text{ of } x1{:}t1 \rightarrow e1 \mid x2{:}t2 \rightarrow e2)$

$$\frac{A \vdash e : t1}{A \vdash \text{inL}_{t2}\ e : t1 + t2} \qquad \frac{A \vdash e : t2}{A \vdash \text{inR}_{t1}\ e : t1 + t2}$$

$$\frac{A, x1{:}t1 \vdash e1 : t \quad A, x2{:}t2 \vdash e2 : t}{A \vdash (\text{case } e \text{ of } x1{:}t1 \rightarrow e1 \mid x2{:}t2 \rightarrow e2) : t}$$