CMSC 631 – Program Analysis and Understanding
Spring 2009

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

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

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

Example

$A = - : int \to int$

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

Another Example

$A = + : int \to int \to int$
$B = A, x : int$

$$\frac{\dfrac{\dfrac{+\ \mathrm{dom}(B)}{B \vdash + :} \quad \dfrac{x\ \mathrm{dom}(B)}{B \vdash x : i}}{B \vdash + \ x : int \to int} \qquad B \vdash 3 : int}{\dfrac{B \vdash + \ x \ 3 : int}{A \vdash (\lambda x{:}int.+\ x\ 3) : int \to int} \qquad \begin{array}{l} A \vdash 4 : \\ int \end{array}}$$

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

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 a small-step, call-by-value semantics
  - If an expression can't be evaluated any more and is not a value, then it is *stuck*

$$\frac{}{(\lambda x.e1)\ v2 \to e1[v2\backslash x]} \qquad \frac{e1 \to e1'}{e1\ e2 \to e1'\ e2}$$

$$\frac{e2 \to e2'}{v1\ e2 \to v1\ e2'}$$

$e ::= v \mid x \mid e\ e$

$v ::= n \mid \lambda x{:}t.e$ *values* – not evaluated