Lecture 14: Type Safety & Run-Time Stack

CSC 131

Kim Bruce

Type Safety

- Is there any connection between type checking rules and semantics?
- If $E \vdash e: T$, what does that say about computation $(e, \mathit{env}) \Rightarrow v$?
- If E and *env* "correspond", then expect v: T

Typed PCF

- T :: = Int | Bool | T -> T
- Provide identifiers w/type when introduced.
- e ::= x | n | true | false | succ | pred | iszero |
  if e then e else e | (fn (x:T) => e) | (e e) |
  rec (x:T) => e *Ignore recursion for now!*

Type-checking Rules

E is *type environment*: identifiers → types

$E \vdash$ n: Int, if n is an integer

$E \vdash$ true: Bool, $E \vdash$ false: Bool

$E \vdash$ succ: Int → Int, $E \vdash$ pred: Int → Int

$E \vdash$ iszero: Int → Bool

$E \vdash$ x: E(x)

More Type-Checking Rules

$$\frac{E \vdash e\text{: Bool},\ \ E \vdash e1\text{: T},\ \ E \vdash e2\text{: T}}{E \vdash \text{if e then e1 else e2 : T}}$$

$$\frac{E \vdash f\text{: T} \rightarrow \text{U},\ \ E \vdash x\text{: T}}{E \vdash \text{(f x): U}}$$

$$\frac{E[x{:}T] \vdash \text{body: U}}{E \vdash \text{(fn (x:T) => body): T} \rightarrow \text{U}}$$

Computation Rules

(id, *env*) $\Rightarrow$ *env*(id)

(n, *env*) $\Rightarrow$ *n*, for n an int

(true, *env*) $\Rightarrow$ *true*

(false, *env*) $\Rightarrow$ *false*

(succ, *env*) $\Rightarrow$ *succ*

(pred, *env*) $\Rightarrow$ *pred*

(iszero, *env*) $\Rightarrow$ *iszero*

$$\frac{(b, \mathit{env}) \Rightarrow \mathit{true},\ \ (e1, \mathit{env}) \Rightarrow v}{(\text{if b then e1 else e2}, \mathit{env}) \Rightarrow v}$$

....

Computation Rules

((fn (x:T) => e), *env*) $\Rightarrow$< fn (x:T) => e, *env* >

$$\frac{(e1, \mathit{env}) \Rightarrow \langle \text{fn (x:T) => e3}, \mathit{env}' \rangle \quad (e2, \mathit{env}) \Rightarrow v2,\ \ (e3, \mathit{env}'[v2/x]) \Rightarrow v}{((e1\ e2), \mathit{env}) \Rightarrow v}$$

Typing Values

$\vdash_v n$: Int, for n an integer

$\vdash_v \mathit{true}$: Bool

$\vdash_v \mathit{false}$: Bool

$\vdash_v \mathit{succ}$: Int $\rightarrow$ Int

$\vdash_v \mathit{pred}$: Int $\rightarrow$ Int

$\vdash_v \mathit{iszero}$: Int $\rightarrow$ Bool

Environment Compatibility

$$\frac{E' \vdash \text{fn } (x{:}T) \Rightarrow e{:}\ T \to U}{\vdash_v < \text{fn } (x{:}T) \Rightarrow e,\ env >{:}\ T \to U}$$

for E' s.t E' and *env* are *compatible*

E and *env* are *compatible* iff

domain(E) = domain(*env*), and

for all x in domain(*env*), $\vdash_v$ *env*(x): E(x)

Safety

Theorem: (Subject Reduction) Let E and *env* be compatible environments.

Let e be a term of typed PCF. If E $\vdash$ e: T and (e,*env*) $\Rightarrow$ *v*, then $\vdash_v$ *v*: T.

Proof: By induction on proof of E $\vdash$ e: T

Proof

Conditional: S'pose E $\vdash$ if b then e1 else e2: T because
E $\vdash$ b: boolean, E$\vdash$e1: T, and E$\vdash$e2: T.
Two cases depending on the evaluation of b.

Case 1: (b, *env*) $\Rightarrow$ *true*.

Then if (e1,*env*) $\Rightarrow$ *v*, it follows that
(if b then e1 else e2, *env*) $\Rightarrow$ *v*.

By induction and E$\vdash$e1: T, it follows that $\vdash_v$ *v*: T, which is all we need.

Case 2: (b, *env*) $\Rightarrow$ *false* is similar.

Skip rest

Type Safety

- Errors have been made in type systems.
  - See examples in OO languages
- Need to verify that type system is consistent with semantics.
- Progress Lemma (*computations don't get stuck*) not shown here, but also important

Attributes of Variable

- Scope Done!
- Lifetime
- Location
- Value

Lifetime

- FORTRAN - all allocated statically - ∞
- Stack-based (C/C++/Java/Pascal/...)
  - local vbles/ parameters: method/procedure/block entry to exit
  - allocate space in activation record on run-time stack
- Heap allocated variable
  - lifetime independent of scope
- Static - global vbles or static vbles

Value & Location

- Sometimes referred to as l-value & r-value
  - x = x + 1 *What does each occurrence of x stand for?*
  - location normally bound when declaration processed
- Normally values change dynamically
  - if frozen at compilation then called constants
  - Java final variables frozen when declaration processed.
  - Java static final bound at compile time.

Aliases

- x and y are aliases if both refer to same location.
- If x, y are aliases then changes to x affect value of y.
- Java has uniform model where assignment is by "sharing", so create aliases.
- Languages that mix are more confusing.
  - Common mistakes occur when not realize aliases. E.g, add elt to priority queue and then change it ...

Pointers

- "Pointers have been lumped with the goto statement as a marvelous way to create impossible to understand programs"
  - K & R, C Programming Language
- Problems
  - Dangling pointers -- leave pointer to recycled space
    - stack frame popped or recycled heap item
  - Dereference nil pointers or other illegal address
  - Unreachable garbage
  - in C: p+1 different from (int)p + 1

Program Units

- Separate segments of code allowing separate declarations of variables
  - Ex.: procedures, functions, methods, blocks
  - During execution represented by unit instance
    - fixed code segment
    - activation record with "fixed" ways of accessing items

Activation Record Structure

- Return address
- Access info on parameters (*how?*)
- Space for local vbles
- *How get access to non-local variables?*

Invoking Function

- Make parameters available to callee
  - E.g., put on stack or in registers
- Save state of caller (registers, prog. counter)
- Ensure callee knows where to return
- Enter callee at first instruction

Returning from Function

- If function, leave result in accessible location
  - e.g., register or top of stack
- Get return address and transfer execution back
- Caller restores state

Parameter Passing

- Call-by-reference (FORTRAN, Pascal, C++)
  - pass address (l-value) of parameter
- Call-by-copying (Algol 60, Pascal, C, C++)
  - pass (r-)value of parameter
  - options: in, out, in-out
- Call-by-name (Algol 60)
  - pass actual expression (as "thunk") - not macro!
  - re-evaluate at each access
  - lazy gives efficient implementation if no side effects

Call-by-name

```
procedure swap(a, b : integer);
  var temp : integer;
  begin
   temp := a;
   a := b;
   b := temp
  end;
```

- Won't always work!
- swap(i, z[i]) with i = 1, z[1] = 3, z[3] = 17
- Can't write swap that always works!

What about Java?

- Conceptually call-by-sharing
- Implemented as call-by-value of a reference