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 $(e, env) \Rightarrow v$?
- If $E$ and $env$ “correspond”, then expect $v : T$

Typed PCF

- $T ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T$
- Provide identifiers w/type when introduced.
- $e ::= x \mid n \mid \text{true} \mid \text{false} \mid \text{succ} \mid \text{pred} \mid \text{iszero} \mid$
  if $e$ then $e$ else $e \mid (\text{fn } x : T \Rightarrow e) \mid (e \ e) \mid$
  rec $(x : T) \Rightarrow e$  \textit{Ignore recursion for now!}

Type-checking Rules

$E$ is type environment: identifiers $\rightarrow$ types

$E \vdash n : \text{Int}$, if $n$ is an integer
$E \vdash \text{true} : \text{Bool}$, $E \vdash \text{false} : \text{Bool}$
$E \vdash \text{succ} : \text{Int} \rightarrow \text{Int}$, $E \vdash \text{pred} : \text{Int} \rightarrow \text{Int}$
$E \vdash \text{iszero} : \text{Int} \rightarrow \text{Bool}$
$E \vdash x : E(x)$
More Type-Checking Rules

\[
\begin{align*}
& \text{E } \vdash \text{e: } \text{Bool}, \, \text{E } \vdash \text{e1: } \text{T}, \, \text{E } \vdash \text{e2: } \text{T} \\
& \text{E } \vdash \text{if } \text{e } \text{then } \text{e1 } \text{else } \text{e2: } \text{T} \\
& \text{E } \vdash \text{f: } \text{T } \rightarrow \text{U}, \, \text{E } \vdash \text{x: } \text{T} \\
& \text{E } \vdash (\text{f } \text{x}) : \text{U} \\
& \text{E } \vdash (\text{fn} (\text{x: } \text{T}) => \text{body}) : \text{T } \rightarrow \text{U}
\end{align*}
\]

Computation Rules

\[
\begin{align*}
& \text{Computation Rules} \\
& \text{(id, } \text{env) } \Rightarrow \text{env(id)} \quad \text{(n, } \text{env) } \Rightarrow \text{n, for n an int} \\
& \text{(true, } \text{env) } \Rightarrow \text{true} \quad \text{(false, } \text{env) } \Rightarrow \text{false} \\
& \text{(succ, } \text{env) } \Rightarrow \text{succ} \quad \text{(pred, } \text{env) } \Rightarrow \text{pred} \\
& \text{(iszero, } \text{env) } \Rightarrow \text{iszero} \\
& \text{(b, } \text{env) } \Rightarrow \text{true, (e1, } \text{env) } \Rightarrow \text{v} \\
& \text{(if } \text{b then e1 else e2, } \text{env) } \Rightarrow \text{v} \quad \text{....}
\end{align*}
\]

Typing Values

\[
\begin{align*}
& \vdash \text{n : Int, for n an integer} \\
& \vdash \text{true : Bool} \\
& \vdash \text{false : Bool} \\
& \vdash \text{succ : Int } \rightarrow \text{Int} \\
& \vdash \text{pred : Int } \rightarrow \text{Int} \\
& \vdash \text{iszero : Int } \rightarrow \text{Bool}
\end{align*}
\]
Environment Compatibility

\[ \begin{align*}
E' \vdash fn (x:T) &\Rightarrow e: T \rightarrow U \\
\vdash \gamma < fn (x:T) &\Rightarrow e, env >: T \rightarrow U \\
\text{for E' s.t E' and env are compatible}
\end{align*} \]

E and env are compatible iff
domain(E) = domain(env), and
for all x in domain(env), \( \vdash env(x): E(x) \)

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 \( \vdash v: T \).

**Proof:** By induction on proof of \( E \vdash e: T \)

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

Proof

**Conditional:** S’pose \( E \vdash if b then e1 else e2: T \) because
\( E \vdash b: \text{boolean}, \ E \vdash e1: T \), 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 \( \vdash v: T \),
which is all we need.

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

Skip rest
Attributes of Variable

- Scope
- Lifetime
- Location
- Value

Done!

Lifetime

- FORTRAN - all allocated statically - $\infty$
- 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

```plaintext
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