6.035: Semantics

More Closures and Records
(with a few slides from last time for recap)
IMP Grammar

\[ E \rightarrow n \mid x \mid -E \mid True \mid False \]
\[ \mid E + E \mid E - E \mid E < E \mid E == E \]
\[ \mid E \times E \mid E / E \mid !E \]
\[ \mid E && E \mid E || E \]

\[ S \rightarrow \text{var } x = E \mid x = E \]
\[ \mid \text{var } x = \text{fun}(x)\{ S* \} \mid x = \text{fun}(x)\{ S* \} \]
\[ \mid x(E) \]
\[ \mid \text{if } (E) \text{BS else BS} \]
\[ \mid \text{BS} \]
\[ \mid \text{while } (E) \text{BS} \]

BS \rightarrow \{ S* \}

(create closure)

(call closure)

Var cannot be introduced in a scope without initialization
Inference Rules: Declaration

\[ \text{var } x = E \]

\[ h(a_f) = \sigma \quad \neg (x \in \text{dom}(\sigma)) \quad (a, h, e) \to v \]

\[ \neg (a_x \in \text{dom}(h)) \quad \sigma[x : a_x] = \sigma' \quad h[a_f: \sigma', a_x: v] = h' \]

\[ (a_f, h, \text{var } x = e) \to h' \]
Inference Rules: Assignment

\[ x = E \]

\[
\frac{(a, h, e) \rightarrow v \quad \text{update}(a, h, x, v, h')}{(a, h, x = e) \rightarrow h'}
\]
Update

\[ \text{update}(a, h, x, v, h') = \]

\[ h(a_f) = \sigma \quad x \in \text{dom}(\sigma) \quad \neg (a_x \in \text{dom}(h)) \quad \sigma[x : a_x] = \sigma' \quad h[a_f : \sigma', \ a_x : v] = h' \]

\[ \text{update}(a_f, h, x, v, h') \]

\[ h(a_f) = \sigma \quad \neg (x \in \text{dom}(\sigma)) \quad \text{update}(\sigma(\rho), h, x, v, h') \]

\[ \text{update}(a_f, h, x, v, h') \]

• Recursively search the stack for \( x \), and update \( x \), return new heap
Inference Rules: Block Scope

Block Scope \{ S^* \}:

\[
\begin{align*}
\sigma = \{ \rho : a_f \} & \quad \neg (a_\sigma \in \text{dom}(h)) \quad h[a_\sigma : \sigma] = h' \quad (a_\sigma, h', S^*) \rightarrow h'' \\
(a_f, h, bs \ S^*) \rightarrow h''
\end{align*}
\]

Sequential Composition $S^*$:

\[
(a, h, S) \rightarrow h' \quad (a, h', S_r) \rightarrow h''
\]

\[
(a, S :: S_r) \rightarrow h''
\]
Extending IMP with Closures

• **Syntax:** create and call closures

• **Representation:** closures are values and frames a long-lived

• **Semantics:** creating and calling closures
Inference Rules: Closure Creation (Declaration)

Declaration \texttt{var } x = \texttt{fun}(x)\{ S* \}:

\[
\begin{align*}
  h(a_f) &= \sigma \\
  \neg(x \in \text{dom}(\sigma)) \\
  \neg(a_x \in \text{dom}(h)) \\
  \sigma[x : a_x] &= \sigma' \\
  h[a_f : \sigma', a_x : f] &= h' \\
  (a_f, h, \text{var } x = \texttt{fun}(x)\{ S* \}) &\rightarrow h'
\end{align*}
\]

- Executing a closure: capture frame pointer and store closure in heap
Managing Scope (Revisited)

```
1: var f = 0;
2: {
3:   var x = 1;
4:   f = fun(y) {
5:     print(x);
6:     x = x + 1;
7:   };
8: }
9: f();
10: f();
```
Inference Rules: Closure Execution

Closure Call: \( x(e) \)

\[(a_f, h, x) \rightarrow (a_c, x_p, S^*)\]  \(x\) Evaluates to a closure

\[(a_f, h, e) \rightarrow v\]  \(E\) evaluates to a value

\(\neg(a_\sigma \in \text{dom}(h))\)  \(\sigma = \{\rho : a_c, x_p : v\}\)  \(h' = h[a_\sigma : \sigma]\)  Create a new frame

\[(a_\sigma, h', S^*) \rightarrow h''\]  Evaluate body of closure

\[(a_f, h, x(e)) \rightarrow h''\]
IMP vs. MITScript vs. JS vs. Python

• IMP:
  • Blocks introduce scopes
  • Var declaration adds variable to the current scope when executed
  • Any reachable scope can be modified

• JS
  • Blocks do not introduce scope, they are merely cosmetic
  • Var declarations add variables to the current scope if they are present in the function regardless of if or when they execute
  • Any reachable scope can be modified
  • If a variable is in a scope but is not initialized, its value defaults to undefined.
Block scope in JS

Block Scope \( \{ S^* \} \):

\[
\sigma = \{ p : a_f \} \quad \neg (a_\sigma \in \text{dom}(h)) \quad h[a_\sigma : \sigma] = h' \quad (a_\sigma, h', S^*) \to h'' \quad (a_f, h, bs \ S^*) \to h''
\]

Sequential Composition \( S^* \):

\[
(a, h, [\square]) \to h \quad (a, h, S) \to h' \quad (a, h', S_r) \to h'' \quad (a, S :: S_r) \to h''
\]

Body executes in same stack frame as before.

No need for new stack frame.
Examples

JS:
var x = 7;
g = function(){
  var x;
  x = 8;
  h = function(){
    console.log("Value x=" + x);
    x = x + 1;
  }
  h();
  h();
  return h;
}

h();
h();
return h;
}

console.log(x)

hh=g()

hh()

console.log(x)
IMP vs. MITScript vs. JS vs. Python

• IMP:
  • Blocks introduce scopes
  • Var declaration adds variable to the current scope when executed
  • Any reachable scope can be modified

• JS
  • Blocks do not introduce scope, they are merely cosmetic
  • Var declarations add variables to the current scope if they are present in the function regardless of if or when they execute
  • Any reachable scope can be modified
  • If a variable is in a scope but not initialized, its value defaults to undefined.

• Python
  • There are no blocks
  • There are no var declarations; an assignment introduces a variable to the scope, even if assignment is not executed
  • Only the current scope and the global scope can be modified
  • If a variable is in scope but not initialized, access produces an error

• MITScript
  • Blocks do not introduce scope, they are merely cosmetic
  • There are no var declarations; an assignment introduces a variable to the scope, even if assignment is not executed
  • Only the current scope and the global scope can be modified
  • If a variable is in a scope but is not initialized, its value defaults to undefined.
Function call in MITScript

\[ \Gamma = a_g; \Gamma'; a_i \]
\[ (\Gamma, h, e_b) \rightarrow (h_1, Function(a_j, fun(x) S)) \]
\[ (\Gamma, h_1, e_p) \rightarrow (h_2, v) \]
\[ a' = freshAddress(h_2) \]
\[ h_3 = h_2[a': newStackFrame(a_j, fun(x)S)] \]
\[ h_4 = update_3(h_3, a', x, v) \]
\[ h_5 = h_4[ret: None] \]
\[ \Gamma' = \Gamma; a' \]
\[ (\Gamma', h_5, S) \rightarrow h_6 \]

\[ (\Gamma, h, e_b(e_p)) \rightarrow (h_6, h(\text{ret})) \]
Examples

Python:
x = 7
def g():
    x = 8
    def h():
        print(x)
        x = x + 1
    h()
    h()
    return h

h()
h()
return h

print(x)

hh=g()

hh()

print(x)
Records and Objects

• Evaluation Relation

\[(a, h, e) \rightarrow v\] \hspace{1cm} \[(a, h, s) \rightarrow h\]
Records and Objects

• Evaluation Relation

\[(a, h, e) \rightarrow (h, v) \quad (a, h, s) \rightarrow h\]

• Expressions also modify the heap
  • record creation needs to store record in the heap
Need to adjust previous rules

- All other rules need to account for this
  - It is trivial in most cases

\[
\begin{align*}
\text{Integer Constant} & \quad n = n_r \\
(a, h, n) & \rightarrow (h, n_r)
\end{align*}
\]

\[
\begin{align*}
\text{Unary Minus} & \quad (\gamma, h, e) \rightarrow (h', v) \quad \text{type}(v) = \text{int} \quad \text{int}(v) = n_r \\
(\gamma, h, -e) & \rightarrow (h', n_r)
\end{align*}
\]
Records

• Record constructor
  • A key design decision is that evaluation returns an address rather than the map
  • Order of evaluation is prescribed by the rule

\[
(a, h, e_0) \rightarrow (h_0, v_0) \quad \ldots \quad (a, h_{i-1}, e_i) \rightarrow (h_i, v_i)
\]

\[
\neg (a_m \in \text{dom}(h)) \quad m = \{x_i : v_i\}_{i \leq n} \quad h_n[a_m : m] = h'
\]

\[
(a_f, h, \{x_i : e_i\}_{i \leq n}) \rightarrow (h', a_m)
\]
Record Update

\[(a_f, h, t) \rightarrow (h', a_t) \quad h''(a_t) = m\]

\[(a_f, h', e) \rightarrow (h'', v) \quad m = \{x_i : v_i\}_{i \leq N} \quad m' = m[x : v] \quad h_3 = h''[a_t : m']\]

\[(a_f, h, t. x = e) \rightarrow h_3\]
The End