6.035: Semantics

Closures
What is a Closure?

Python

```python
f = lambda x, y : x + y
```

C++

```cpp
auto f = [](int x, int y) {
    return x + y;
};
```

Java

```java
BinaryOperator<int> f = (int x, int y) -> {
    return x + y;
};
```

Javascript

```javascript
var f = function(x, y) {
    return x + y;
}

var f = (x, y) => {
    return x + y;
}
```
What is a Closure?

- A function and a scope (in which to run the function)
- First-class: a closure is a value
- Higher-Order: functions may take closures as arguments

```javascript
var f = fun(x) {
    print(x);
};
f(1)
```

Output: 1

```javascript
var x = 1;
var f = fun(y) {
    print(x);
};
f(2);
```

Output: 1

```javascript
var x = 1;
var f = fun() {
    x = 2
};
var g = fun(x) {
    x();
}
g(f);
print(x);
```

Outputs: 2
Why?
Challenge: Managing Scopes

A closure may refer to value of variable in from a context that has been popped from the stack

Scope’s frame may live longer than the single execution of the scope’s code

Strategy: closure contains a pointer to the frame in which it was created

```
var f = 0;
{
    var x = 1;
    f = fun(y) {
        print(x);
        x = x + 1;
    };
}
f();
f();
f();
```

Output:
1
2
3
Extending IMP with Closures

• Syntax: create and call closures

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

• Semantics: creating and calling closures
Extending IMP with Closures

• Syntax: create and call closures

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

• Semantics: creating and calling closures
Add Closures: Grammar

$$E \rightarrow n \mid x \mid \neg E \mid True \mid False$$
$$\mid E + E \mid E - E \mid E < E \mid E == E$$
$$\mid E \times E \mid E / E \mid !E$$
$$\mid E \&\& E \mid E \mid E$$

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

$$BS \rightarrow \{ S^* \}$$
(create closure)

(call closure)
Extending IMP with Closures

- Syntax: create and call closures

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

- Semantics: creating and calling closures
Linked Stacks (Boardwork)

1: var x = 1;
2: {
3:   var x = 2;
4:   {
5:     x = 3;
6:   }
7: }

Frames, Heaps, and **Linked Stacks**

- **Domain of addresses**: $A$
  - Location of an integer value in memory

- **Domain of parent fields**: $P = \{ \rho \}$
  - Field to access parent of frame

- **Domain of frames**: $\sigma \in \Sigma = (X \cup P) \to A$
  - A *frame* $\sigma$ is an element of the domain $\Sigma$, which is set of all (partial) functions that map a variable $x$ from the domain of all variables $X$ to an address

- **Domain of closures**: $f \in F = A \times X \times S^*$
  - A closure is a tuple consisting of an frame address, variable, and sequence of statements

- **Domain of heaps**: $h \in H = A \to V$ where $V = \mathbb{N} \cup \mathbb{B} \cup \Sigma \cup F$
  - A *heap* $h$ is an element of the domain $H$, which is set of all (partial) functions that map an address to a value (integer or boolean)
Evaluation Relations (with heaps and stacks)

• Define the semantics of each term in our language \((E, B, \text{and} \ S)\) with an evaluation relation:

\[
\begin{align*}
(\gamma, h, e) & \rightarrow v \\
(\gamma, h, s) & \rightarrow (\gamma, h) \\
(\gamma, h, bs) & \rightarrow (\gamma, h)
\end{align*}
\]

• Meaning: given a stack, and heap, the term evaluates to a result
Evaluation Relations (with linked stacks)

• Define the semantics of each term in our language ($E$, $B$, and $S$) with an evaluation relation:

  $$(a, h, e) \rightarrow v \quad (a, h, s) \rightarrow h \quad (a, h, bs) \rightarrow h$$

• Meaning: given a frame pointer, and heap, the term evaluates to a result
Expressions: Inference Rules

- **Integer Constant**: 
  \[
  n = n_r \\
  (\gamma, h, n) \rightarrow n_r
  \]

- **Variable Reference**: 
  \[
  \text{lookup}(\gamma, x) = \sigma \
  \sigma(x) = a \
  h(a) = v_r \\
  (\gamma, h, x) \rightarrow v_r
  \]

- **Unary Minus**: 
  \[
  (\gamma, h, e) \rightarrow v \
  \text{type}(v) = \text{int} \
  - \text{int}(v) = n_r \\
  (\gamma, h, -e) \rightarrow n_r
  \]
Lookup

- $lookup(a, x) = \sigma$ : lookup variable binding by traversing the stack

\[
\begin{align*}
\neg(a = 0) & \quad h(a) = \sigma & \quad x \in \text{dom}(\sigma) \\
\hline
lookup(a, x) = \sigma
\end{align*}
\]

\[
\begin{align*}
\neg(a = 0) & \quad h(a) = \sigma & \quad \neg(x \in \text{dom}(\sigma)) & \quad lookup(\sigma(\rho), x) = \sigma' \\
\hline
lookup(a, x) = \sigma'
\end{align*}
\]
# Expressions: Inference Rules

<table>
<thead>
<tr>
<th>Category</th>
<th>Rule</th>
<th>Context</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Constant</td>
<td>$n = n_r$</td>
<td>$(\gamma, h, n) \rightarrow n_r$</td>
<td>int (v) = n_r</td>
</tr>
<tr>
<td>Variable Reference</td>
<td>$\text{lookup}(\gamma, x) = \sigma \quad \sigma(x) = a \quad h(a) = v_r$</td>
<td>$(\gamma, h, x) \rightarrow v_r$</td>
<td>int (v) = v_r</td>
</tr>
<tr>
<td>Unary Minus</td>
<td>$(\gamma, h, e) \rightarrow v \quad \text{type}(v) = \text{int} \quad \text{int}(v) = n_r$</td>
<td>$(\gamma, h, -e) \rightarrow n_r$</td>
<td>int (v) = n_r</td>
</tr>
<tr>
<td></td>
<td>$\text{lookup}(a_f, x) = \sigma \quad \sigma(x) = a_x \quad h(a_x) = v_r$</td>
<td>$(a_f, h, x) \rightarrow v_r$</td>
<td>int (v) = v_r</td>
</tr>
<tr>
<td></td>
<td>$(a, h, e) \rightarrow v \quad \text{type}(v) = \text{int} \quad \text{int}(v) = n_r$</td>
<td>$(a, h, -e) \rightarrow n_r$</td>
<td>int (v) = n_r</td>
</tr>
</tbody>
</table>
Inference Rules: Declaration

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

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

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

\[ (a_f, h, \text{var } x = e) \rightarrow h' \]
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

• $update(a, h, x, v, h') =$

$h(a_f) = \sigma \quad x \in dom(\sigma) \quad \neg(a_x \in dom(h)) \quad \sigma[x : a_x] = \sigma' \quad h[a_f : \sigma', a : v] = h'$

$update(a_f, h, x, v, h')$

$h(a_f) = \sigma \quad \neg(x \in dom(\sigma)) \quad update(\sigma(\rho), h, x, v, h')$

$update(a_f, h, x, v, h')$

• 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 \( S^* \):

\[
\begin{align*}
(a, h, [\[]) &\rightarrow h \\
(a, h, S) &\rightarrow h' \\
(a, h', S_r) &\rightarrow h'' \\
(a, S :: S_r) &\rightarrow h''
\end{align*}
\]
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 $\text{var } x = \text{fun}(x)\{ S^* \}$:

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

\[
(a_f, h, \text{var } x = \text{fun}(x_p)\{ S^* \}) \rightarrow h'
\]

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

```
var f = 0;
{
    var x = 1;
    f = fun(y) {
        print(x);
        x = x + 1;
    };
}

f();
f();
```

<table>
<thead>
<tr>
<th>Line</th>
<th>FP (a)</th>
<th>Heap (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
<td>{ 100 : 0, 200: {p : 0, f : 100}}</td>
</tr>
<tr>
<td>3</td>
<td>208</td>
<td>{ 100 : 0, 108 : 1, 200 : {p : 0, f : 100}, 208 : {p : 200, x : 108}}</td>
</tr>
<tr>
<td>7</td>
<td>208</td>
<td>{ 100 : 0, 108 : 1, 116 : (208, y, {print(x); x = x + 1}), 200 : {p : 0, f : 116}, 208 : {p : 200, x : 108}}</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
<td>{ 100 : 0, 108 : 1, 116 : (208, y, {print(x); x = x + 1}), 200 : {p : 0, f : 116}, 208 : {p : 200, x : 108}}</td>
</tr>
<tr>
<td>9</td>
<td>200</td>
<td>{ 100 : 0, 108 : 1, 116 : (208, y, {print(x); x = x + 1}), 124: 2, 200 : {p : 0, f : 116}, 208 : {p : 200, x : 124}}</td>
</tr>
<tr>
<td>10</td>
<td>200</td>
<td>{ 100 : 0, 108 : 1, 116 : (208, y, {print(x); x = x + 1}), 124: 2, 132:3 200 : {p : 0, f : 116}, 208 : {p : 208, x : 132}}</td>
</tr>
</tbody>
</table>
Inference Rules: Closure Execution

Closure Call: $x(e)$

\[(a_f, h, x) \rightarrow (a_c, x_p, S^*)\] \hspace{1cm} x Evaluates to a closure

\[(a_f, h, e) \rightarrow v\] \hspace{1cm} E evaluates to a value

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

\[(a_\sigma, h', S^*) \rightarrow h''\] \hspace{1cm} Evaluate body of closure

\[(a_f, h, x(e)) \rightarrow h''\]
The End