Review: Allocated Memory

- When accessing a variable $v$ inside the environment of method $f$ we get its address using:
  \[ \gamma_f(v) = \max_x : (v, \text{id}, x) \in \gamma_f \]
- In addition we can get the value of the variable as:
  \[ \sigma_f(v) = \mu(\gamma_f(v)) \]
- Thus, in this example, we have:
  \[ \gamma_f(i) = 100 \]
  \[ \sigma_f(i) = \mu(\gamma_f(i)) = \mu(100) = 3 \]

Environment and Memory

- Given the ability to distinguish local variables and global variables the environment $\gamma_f$ of a method $f$ is simply the union of its parameters and local variables with the existing environment $\gamma$:
  \[ \gamma_f = \gamma \cup f.params \cup f.locals \]

Review: Local Memory

- When get the address of a variable using the maximum:
  \[ \gamma_f(v) = \max_{x} : (v, \text{id}, x) \in \gamma_f \]
- This is to distinguish local variables from global variables that they mask.
- For example, suppose we had the following at the start of method $f$:
  \[ \gamma_f = \{ (y, 0), (x, 100), (y, 101) \} \]
  \[ \mu = \{ (0, \text{undef}), \ldots, (100, 5), (101, 1), \ldots, (a - 1, \text{undef}), (a, \text{unused}), (a + 1, \text{unused}), \ldots, (n, \text{unused}) \} \]
- Here, two $y$ variables are a local @ 101 and non-local one @ 0 and so:
  \[ \gamma_f(y) = 101 \]
  \[ \sigma_f(y) = \mu(\gamma_f(y)) = \mu(101) = 1 \]
Program Meaning

As before, meaning of a program is the state given by:

\[ M : \text{Program} \times \text{State} \rightarrow \text{State} \]

\[ M(p, \sigma) = M(p.\text{main}(), \text{allocate}(p.\text{globals}, \sigma)) \]

The initial state of a program is always empty:

\[ \sigma_0 = \gamma_0 \times \mu(0) \times 0 \]

\[ = \{ \} \times \{(0, \text{unused}), (1, \text{unused}), \ldots, (n, \text{unused})\} \times 0 \]

Environment and Memory

The memory component of a state is used to track actual values of variables, and so uses the over-riding union to track this.

For example, if we assign value 15 to the variable \( y \) in the method \( f \) of the prior example we have:

\[ \gamma_f = \{ (y, 0), (x, 100), (y, 101) \} \]

\[ \mu = \{ (0, 10), \ldots, (100, 5), (101, 1), \ldots, (a - 1, \text{undef}), (a, \text{unused}), (a + 1, \text{unused}), \ldots, (n, \text{unused}) \} \]

\[ \mu' = \mu \cup \{ (y, 15) \} \]

\[ = \mu \cup \{ (101, 15) \} \]

\[ \mu = \{ (0, 10), \ldots, (100, 5), (101, 15), \ldots, (a - 1, \text{undef}), (a, \text{unused}), (a + 1, \text{unused}), \ldots, (n, \text{unused}) \} \]

Updating the State: Start

We start our program in the initial empty state:

\[ \sigma_0 = \{ \} \times \{(0, \text{unused}), (1, \text{unused}), \ldots, (n, \text{unused})\} \times 0 \]

As before, we begin by allocating space for global variables:

\[ \sigma_1 = \text{allocate}(x, y, \sigma_0) \]

\[ = \{(x, 0), (y, 1)\} \times \{(0, \text{undef}), (1, \text{undef}), (2, \text{undef}), \ldots, (n, \text{undef})\} \times 2 \]

We can then do any assignments we need for those variables:

\[ \sigma_2 = \sigma_1 \cup \{ (\gamma_1(x), 0), (\gamma_1(y), 0) \} \]

\[ = \sigma_1 \cup \{ (0, 0), (1, 0) \} \]

\[ = \{(x, 0), (y, 1)\} \times \{(0, 0), (1, 0), (2, \text{undef}), \ldots, (n, \text{undef})\} \times 2 \]

Allocating and Deallocating Memory

- **allocate** operator—adds (identifier-address) and (address-value) pairs to the current state:

  \[ \sigma = \gamma \times \mu \times a \]

  allocate \( (v_1, v_2, \ldots, v_k, \sigma) \) = \( \gamma' \times \mu' \times a' \), where:

  \[ \gamma' = \gamma \cup \{ (v_i.\text{id}, a_i) \} \times \{ (v_i.\text{id}, a_i + 1), \ldots, (v_i.\text{id}, a_i + (k - 1)) \} \]

  \[ \mu' = \mu \cup \{ (a, \text{unused}), (a + 1, \text{unused}), \ldots, (a + (k - 1), \text{unused}) \} \]

  \[ a' = a + k \]

- **deallocate** operator—removes variables from stack and frees memory:

  \[ \sigma = \gamma \times \mu \times a \]

  deallocate \( (v_1, v_2, \ldots, v_k, \sigma) \) = \( \gamma' \times \mu' \times a' \), where:

  \[ \gamma' = \gamma \setminus \{ (v_i.\text{id}, a_i) \} \times \{ (v_i.\text{id}, a_i + 1), \ldots, (v_i.\text{id}, a_i + (k - 1)) \} \]

  \[ \mu' = \mu \setminus \{ (a, \text{unused}), (a + 1, \text{unused}), \ldots, (a + (k - 1), \text{unused}) \} \]

  \[ a' = a - k \]
#### Updating the State: main()

- After the global variables are dealt with we can evaluate the `main()` method, starting at line 14:
  \[ \sigma_2 = \{ (x, 0), (y, 1) \} \times \{ (0, 0), (1, 0), (2 \text{ unused}), \ldots, (n, \text{ unused}) \} \times 2 \]
- We begin by allocating space for local variables of `main()`:
  \[ \sigma_3 = \text{allocate}(z, c, \sigma_2) \]
  \[ = \{ (x, 0), (y, 1), (z, 2), (c, 3) \} \times \mu_3 \times 4 \]
  where: \( \mu_3 = \{ (0, 0), (1, 0), (2, \text{ unused}), (3, \text{ unused}), (4, \text{ unused}), \ldots, (n, \text{ unused}) \} \)
- Line 15 of `main()` then does an assignment as follows:
  \[ \sigma_4 = \sigma_3 \uplus \{ (\gamma(2), 1) \} \]
  \[ = \{ x, 1 \} \]
  \[ = \{ (x, 0), (y, 1), (z, 2), (c, 3) \} \times \mu_4 \times 4 \]
  where: \( \mu_4 = \{ (0, 0), (1, 0), (2, 1), (3, \text{ unused}), (4, \text{ unused}), \ldots, (n, \text{ unused}) \} \)

#### Calling a void Method

- Line 16 of `main()` calls `void A()`:
  \[ M(\lambda(z), A, \sigma_4) = \text{removeframe}(\lambda, \text{params}, \text{locals}, \text{locals}) \]
  \[ M(\lambda, \text{BodyValue}(\lambda, \text{params}, \text{locals}), \text{locals}) \]
  \[ = \text{removeframe}([w], [1]) \]
  \[ M(\lambda, \text{BodyValue}([w], [1], [\text{addframe}(\text{locals})])) \]

The call to `addframe` has the effect of hiding the local variables of `main()` while calling allocate to set aside new space for the parameters/locals of `A()`:
\[ \sigma_5 = \text{addframe}([w], [1], \text{locals}) \]
\[ = \text{allocate}(\phi, 1, \text{locals}) \]
\[ = \{ (x, 0), (y, 1), (w, 4), (4, 5) \} \times \mu_5 \times 6 \]
where: \( \mu_5 = \{ (0, 0), (1, 0), (2, 1), (3, \text{ unused}), (4, \text{ unused}), (5, \text{ unused}), (6, \text{ unused}), \ldots, (n, \text{ unused}) \} \)

#### The Meaning of a Method Call

- We can formalize the notion of a method call, using the concept of a stack frame, as follows:
  \[ M(c, f, \sigma) = \text{removeframe}(f, \text{params, f, locals}, M(\text{f.body, ByValue}(f, \text{params}, f, \text{locals}, \sigma))) \]

- **Remove stack frame after evaluating method (i.e., once it returns)**
- **Assign method parameters by value**
- **Add stack frame for the method being called**

#### Calling a void Method

- We have added the stack frame for method `A()`:
  \[ M(\lambda(z), A, \sigma_4) = \text{removeframe}([w], [1], M(\lambda, \text{BodyValue}([w], [1], \sigma_4))) \]
  \[ \sigma_5 = \{ (x, 0), (y, 1), (w, 4), (4, 5) \} \times \mu_5 \times 6 \]
where: \( \mu_5 = \{ (0, 0), (1, 0), (2, 1), (3, \text{ unused}), (4, \text{ unused}), (5, \text{ unused}), \}
\( (6, \text{ unused}), \ldots, (n, \text{ unused}) \} \)

Next, a call to `ByValue` copies the values of method call parameters to the formal parameters of the method itself:
\[ \sigma_6 = \text{ByValue}([w], [2], \sigma_5) \]
\[ \sigma_7 = \{ (x, 0), (y, 1), (w, 4), (4, 5) \} \times \mu_6 \times 6 \]
where: \( \mu_6 = \{ (0, 0), (1, 0), (2, 1), (3, \text{ unused}), (4, \text{ unused}), (5, \text{ unused}), \}
\( (6, \text{ unused}), \ldots, (n, \text{ unused}) \} \)

---

Friday, 12 Apr 2019  
Programming Languages (CS 421/521)
Parameter Passing by Value

- In this example, ByValue copies values of the local variable `z` of `main()` to the input parameter `w` of `A()`:

\[
\sigma_\theta = \text{ByValue}(\{w\}, \{z\}, \sigma_v)
\]

where:

\[
\mu_v = \mu_s \cup \{\gamma(w) = w + 1\}
\]

\[
\mu_s \cup \{\{4, \mu_s(2)\}\}
\]

\[
\mu_s \cup \{(4, 1)\}
\]

\[
\{(0, 0), (1, 0), (2, 1), (3, \text{undef}), (4, 1), (5, \text{undef}), (6, \text{undef}), \ldots, (n, \text{undef})\}
\]

Getting the necessary addresses here involves two different environments, since variable `z` is not visible from inside `A()`:

\[
\gamma_z \text{ from } \text{main}() \\
\gamma_z \text{ from } A()
\]

The runtime stack will still store the information about the local environment of `main()`, even if that information is not visible to the new method.

- Parameter Passing by Reference

A by-value pass copies from old address `(w@2)` to new address `(w@4)` on the stack:

\[
\sigma_\theta = \text{ByValue}(\{w\}, \{z\}, \sigma_v)
\]

where:

\[
\mu_v = \mu_s \cup \{\gamma(w) = w + 1\}
\]

\[
\mu_s \cup \{(4, \mu_s(2)\}\}
\]

\[
\mu_s \cup \{(4, 1)\}
\]

\[
\{(0, 0), (1, 0), (2, 1), (3, \text{undef}), (4, 1), (5, \text{undef}), (6, \text{undef}), \ldots, (n, \text{undef})\}
\]

A by-reference pass would re-use the address `(w@2)` for the parameter `w`, giving the method direct access to value, even without being able to use local variable `z` in `A()`:

\[
\sigma_\theta = \text{ByReference}(\{w\}, \{z\}, \text{addframe}(\{w\}, \{1\}, \sigma_v))
\]

where:

\[
\mu_v = \mu_s \cup \{(4, \mu_s(2)\}\}
\]

\[
\mu_s \cup \{(4, 1)\}
\]

\[
\{(0, 0), (1, 0), (2, 1), (3, \text{undef}), (4, \text{undef}), (5, \text{undef}), \ldots, (n, \text{undef})\}
\]

The code uses less space on the stack, since address `2` is being shared by both `z` and `w`.

Method Execution

- We can now evaluate the body of method `A()` itself:

\[
M(A(z), A, \sigma_v) = \text{removeframe}(\{w\}, \{(z)\})\cdot M(A, \sigma_v)
\]

\[
\sigma_\theta = \{(x, 0), (y, 1), (w, 4), (1, 5)\} \times \mu_v \times 6
\]

where:

\[
\mu_v = \{(0, 0), (1, 0), (2, 1), (3, \text{undef}), (4, 1), (5, \text{undef}), (6, \text{undef}), \ldots, (n, \text{undef})\}
\]

This simply does the required arithmetic and assignments, as covered by the semantics for these operations:

\[
\sigma_T = \{(x, 0), (y, 1), (w, 4), (1, 5)\} \times \mu_T \times 6
\]

where:

\[
\mu_T = \mu_v \cup \{(y, 1) = w + 1\}
\]

\[
\mu_v \cup \{(5, 2) \cup \{1, 2\}\}
\]

\[
\{(0, 0), (1, 2), (2, 1), (3, \text{undef}), (4, 1), (5, 2), (6, \text{undef}), \ldots, (n, \text{undef})\}
\]

\[\text{See lectures 15 \& 16}\]

Method Return

- Finally, method `A()` returns control back to line 16 of `main()`:

\[
M(A(z), A, \sigma_v) = \text{removeframe}(\{w\}, \{(z)\})
\]

\[
\sigma_T = \{(x, 0), (y, 1), (w, 4), (1, 5)\} \times \mu_T \times 6
\]

where:

\[
\mu_T = \{(0, 0), (1, 2), (2, 1), (3, \text{undef}), (4, 1), (5, 2), (6, \text{undef}), \ldots, (n, \text{undef})\}
\]

The call to `removeframe` will deallocate memory set aside for `A()` and restore local state for `main()`:

\[
\sigma_s = \text{removeframe}(\{w\}, \{(z)\})
\]

\[
= \text{deallocate}(x, y, z, \mu_T \cup \text{main.locals})
\]

\[
= \{(x, 0), (y, 1), (w, 4), (z, 3)\} \times \mu_s \times 4
\]

where:

\[
\mu_s = \{(0, 0), (1, 2), (2, 1), (3, \text{undef}), (4, \text{undef}), \ldots, (n, \text{undef})\}
\]
This Week

- **Topic:** Semantics of functions/methods
- **Reading:** Text, 9.1–9.3
- **Meet:** usual schedule
- **Homework 03:** due Friday, 12 April, 5:00 PM

- **Office Hours:** Wing 210
  - Monday, 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday, 2:00 PM – 3:00 PM
  - Friday: 9:00 AM – 10:30 AM

Friday, 12 April 2019