Semantics of Call and Return

- **Meaning rule 01** for method calls: A `Call c` to method `f` has the following steps:
  1. Make a stack frame containing parameters and locals of `f`.
  2. Evaluate `Call c`'s arguments and assign values to each of the corresponding parameters.
  3. If `f` is non-void, add a result variable (int with `f`'s name).
  4. Push the stack frame onto the run-time stack.
  5. Interpret the statements in `f`'s body.
  6. Pop the stack frame from the stack.
  7. If `f` is non-void, return the value of the result variable to the Expression where call `Call c` appears.

Non-void Methods

- **Meaning rule 02** for non-void method calls: The meaning of a Return statement is the result of assigning the given Expression to the method's result variable.
- **Meaning rule 03** for non-void method calls: The meaning of a Call Expression is the value of the result variable at the point of the method's Return statement.
- **Note**: our language distinguishes between a Call Expression and a Call Statement:
  1. A Call Expression appears on the right-hand side of an Assignment; the method called must be non-void.
  2. A Call Statement appears by itself with no Assignment; the method called must be void.

Example Program Trace

<table>
<thead>
<tr>
<th>Call</th>
<th>Return</th>
<th>Visible State</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td></td>
<td>&lt;x, 0&gt;, &lt;y, 0&gt;, &lt;a, un.&gt;, &lt;b, un.&gt;, &lt;c, un.&gt;</td>
</tr>
<tr>
<td>addTo</td>
<td></td>
<td>&lt;x, 0&gt;, &lt;y, 0&gt;, &lt;i, 1&gt;, &lt;j, 2&gt;</td>
</tr>
<tr>
<td>addTo</td>
<td></td>
<td>&lt;x, 1&gt;, &lt;y, 2&gt;, &lt;i, 1&gt;, &lt;j, 2&gt;</td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td>&lt;x, 1&gt;, &lt;y, 2&gt;, &lt;sum, un.&gt;</td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td>&lt;x, 1&gt;, &lt;y, 2&gt;, &lt;sum, 3&gt;</td>
</tr>
<tr>
<td>main</td>
<td></td>
<td>&lt;x, 1&gt;, &lt;y, 2&gt;, &lt;a, 1&gt;, &lt;b, 2&gt;, &lt;c, 3&gt;</td>
</tr>
</tbody>
</table>

Note: the `sum()` method has a return variable (also named `sum`).
Formal Semantics: Typing

Once we introduce methods to our language, our semantics will need to start tracking the types of various entities.

Even in our simple language where each primitive is of int type, we want to formally distinguish void and non-void methods and calls.

A semantics that can deal with this distinction is easy to extend to situations where we have other sorts of non-integer types.

Formal Semantics: Type Maps

A type map takes an identifier to a type, or combination of types.

For global variables, map \( tm_G \) is a set of <identifier, type> pairs:

\[
\text{tm}_G = \{(x, \text{int}), (y, \text{int})\}
\]

For set of all methods, map \( tm_F \) is a set of triples, giving each function's name, return type, and set of <parameter, type> pairs:

\[
\text{tm}_F = \{\langle A, \text{void}, \{(w, \text{int})\}\rangle, \langle B, \text{int}, \{}\rangle, \\
\langle \text{main}, \text{void}, \{}\rangle\}\}
\]

Formal Semantics: Typing Function

Type maps for each method are generated by a typing function:

\[
\text{typing} : \text{Declarations} \times \text{Methods} \times \text{Method} \rightarrow \text{Type Map}
\]

\[
\text{typing}(G, F, f) = (\text{tm}_G \cup \text{tm}_F) \cup \text{tm}_f, \text{ where:}
\]

\[
\text{tm}_G = \bigcup_i \{G_i, \text{id, } G_i, \text{type}\}
\]

\[
\text{tm}_F = \bigcup_i \{F_i, \text{id, } F_i, \text{type, } F_i, \text{params}\}
\]

\[
\text{tm}_f = f.\text{params} \cup f.\text{locals}
\]

\[
\text{f.params} = \bigcup_i \{\text{params, } id, \text{params, } type\}
\]

\[
\text{f.locals} = \bigcup_i \{\text{local, } id, \text{local, } type\}
\]
Formal Semantics: Type Maps

> Remember that we are using an over-riding union (⋓): local variables mask global ones that have the same identifier:

\[
\begin{align*}
\text{tm}_0 &= \{ (x, \text{int}), (y, \text{int}) \} \\
\text{tm}_y &= \{ (A, \text{void}, \{ (w, \text{int}) \}), (B, \text{int}, \{ \}), (\text{main}, \text{void}, \{ \}) \}
\end{align*}
\]

> In method B(), the local variable y replaces the global one in the type map.

Allocated Memory

> The state of a method will contain:
1. Its own variables and their addresses on the stack
2. Memory allocations: all active static and stack addresses \([0, \ldots, (a - 1)]\) and any other free addresses still available for use by the stack \([a, \ldots, n]\):
   - Memory is allocated in contiguous blocks
   - Active addresses that are not yet used represent variables that are declared, but not assigned yet; these are labeled undef
   - Free addresses represent stack space we are not currently using for anything; these are labeled unused

Denotational Semantics of Methods

> We can now formalize the meaning rules for methods, involving stack frames, to give a semantics based upon:
1. State: \(\sigma\) (sigma)
2. Environment: \(\gamma\) (gamma)
3. Memory: \(\mu\) (mu)

> The state of a method \(f\) is a triple:

\[
\sigma_f = \gamma_f \times \mu \times \alpha
\]

> The environment \(\gamma_f\) gives (identifier-address) pairs for all the identifiers that are on the run-time stack

> Address \(\alpha\) is top of run-time stack (not yet allocated)

Allocated Memory

> E.g.: method \(f\) has parameters \((i == 3), (j == -2)\), and a local variable \(k\) that is declared, with no value

> Supposing that the three variables are assigned memory locations \([100, 101, 102]\) on the stack respectively, local environment and overall memory might look like the following:

\[
\begin{align*}
\gamma_f &= \{(4, 100), (5, 101), (6, 102)\} \\
\mu &= \{0, \text{undef}, \ldots, (100, 3), (101, -2), (102, \text{undef}), (103, \text{undef}), \ldots, (a - 1, \text{undef}), (a, \text{undef}), (a + 1, \text{undef}), \ldots, (n, \text{undef})\}
\end{align*}
\]

> Here, the addresses \([100, 101, 102]\) are part of the stack frame for \(f\) and we have:
1. Locations before that are for global variables and those that belong to previous method calls (stack frames) still in use
2. Locations after that are parts of later method calls (stack frames), also still in use
Allocated Memory

\( \gamma_f = \{ (i, 100), (j, 101), (k, 102) \} \)
\( \mu = \{ (0, \text{undef}), \ldots, (100, 3), (101, -2), (102, \text{undef}), (103, \text{undef}), \ldots, (a - 1, \text{undef}), (a, \text{unused}), (a + 1, \text{unused}), \ldots, (n, \text{unused}) \} \)

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

Local Memory

- When get the address of a variable using the maximum:
  \[ \gamma_f(v) = \max_x \langle v.id, x \rangle \in \gamma_f \]
- This is to distinguish local variables from global variables that they mask
- For example, suppose we had the following:
  \( \gamma_f = \{ (y, 0), (x, 100), (y, 101) \} \)
  \( \mu = \{ (0, 10), \ldots, (100, 5), (101, 1), \ldots, (a - 1, \text{undef}), (a, \text{unused}), (a + 1, \text{unused}), \ldots, (n, \text{unused}) \} \)
- Here, two \( y \) variables are a local @ 101 and non-local one @ 0 and so:
  \[ \gamma_f(y) = 101 \]
  \[ \sigma_f(y) = \mu(\gamma_f(y)) = \mu(101) = 1 \]