1.9 Language Semantics

The following sections contain the operational semantics of ROOPL++, as specified by syntax-directed inference rules.

1.9.1 Preliminaries

We define l to be a location. We define a location for integer variables to bind to a single location in program memory and a vector of memory locations for object and array variables, where the vector is the size of the object or array. A memory location is in the set of non-negative integers, $\mathbb{N}_0$. An environment γ is a partial function mapping variables to memory locations. A store μ is a partial function mapping memory locations to values. An object is a tuple of a class name and an environment mapping fields to memory locations. A value is either an integer, an object, a location or a vector of locations.

Applications of environments γ and stores μ are analogous to the type environment Γ, defined in section 1.8.1.

$$
\begin{aligned}
l \in \text{Locs} &= \mathbb{N}_0 \\
\gamma \in \text{Envs} &= \text{VarIDs} \rightharpoonup \text{Locs} \\
\mu \in \text{Stores} &= \text{Locs} \rightharpoonup \text{Values} \\
\text{Objects} &= \Big\{ \langle c_f,\ \gamma_f \rangle \mid c_f \in \text{ClassIDs} \ \wedge \ \gamma_f \in \text{Envs} \Big\} \\
v \in \text{Values} &= \mathbb{Z} \cup \text{Objects} \ \cup \text{Locs} \cup [\text{Locs}]
\end{aligned}
$$

Figure 1.21: Semantic values, originally from [1]

1.9.2 Expressions

The judgment:

$$\langle \gamma, \mu \rangle \vdash_{expr} \ e \Rightarrow v$$

defines the meaning of expressions. We say that under environment γ and store μ, expression e evaluates to value v.

$$\frac{}{\langle \gamma, \mu \rangle \vdash_{expr} \ n \Rightarrow \bar{n}}\ \textsc{Con} \qquad \frac{}{\langle \gamma, \mu \rangle \vdash_{expr} \ x \Rightarrow \mu\Big(\gamma(x)\Big)}\ \textsc{Var} \qquad \frac{}{\langle \gamma, \mu \rangle \vdash_{expr} \ \texttt{nil} \Rightarrow 0}\ \textsc{Nil}$$

$$\frac{\langle \gamma, \mu \rangle \vdash_{expr} \ e_1 \Rightarrow v_1 \qquad \langle \gamma, \mu \rangle \vdash_{expr} \ e_2 \Rightarrow v_2 \qquad [\![\otimes]\!](v_1, v_2) = v}{\langle \gamma, \mu \rangle \vdash_{expr} \ e_1 \otimes e_2 \Rightarrow v}\ \textsc{BinOp}$$

Figure 1.22: Semantic inference rules for expressions, originally from [1]

As shown in figure 1.22, expression evaluation has no effects on the store. Logical values are represented by *truthy* and *falsy* values of any non-zero value and zero respectively. The evaluation of binary operators is presented in figure 1.24.

$$\frac{\langle \gamma, \mu \rangle \vdash_{expr} \; e \Rightarrow v \qquad \gamma(x) = l \qquad \mu(l)[v] = l' \qquad \mu(l') = w}{\langle \gamma, \mu \rangle \vdash_{expr} \; x[e] \Rightarrow w} \textsc{ArrElem}$$

Figure 1.23: Extension to the semantic inference rules for expression in RoopL++

For RoopL++, we extend the expression ruleset with a single rule for array element variables shown in figure 1.23. As with the expressions inference rules in RoopL, this extension has no effect on the store.

$$\begin{aligned}
[\![\texttt{+}]\!](v_1, v_2) &= v_1 + v_2 & [\![\texttt{\%}]\!](v_1, v_2) &= v_1 \bmod v_2 \\
[\![\texttt{-}]\!](v_1, v_2) &= v_1 - v_2 & [\![\texttt{\&}]\!](v_1, v_2) &= v_1 \wedge v_2 \quad , \textit{bitwise} \\
[\![\texttt{*}]\!](v_1, v_2) &= v_1 \times v_2 & [\![\texttt{|}]\!](v_1, v_2) &= v_1 \vee v_2 \quad , \textit{bitwise} \\
[\![\texttt{/}]\!](v_1, v_2) &= v_1 \div v_2 & [\![\texttt{\^{}}]\!](v_1, v_2) &= v_1 \oplus v_2 \\
[\![\texttt{\&\&}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = 0 \vee v_2 = 0 \\ 1 & \text{otherwise} \end{cases} & [\![\texttt{<=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \leq v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{||}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 = 0 \\ 1 & \text{otherwise} \end{cases} & [\![\texttt{>=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \geq v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{<}]\!](v_1, v_2) &= \begin{cases} 1 & \text{if } v_1 < v_2 \\ 0 & \text{otherwise} \end{cases} & [\![\texttt{=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{>}]\!](v_1, v_2) &= \begin{cases} 1 & \text{if } v_1 > v_2 \\ 0 & \text{otherwise} \end{cases} & [\![\texttt{!=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \neq v_2 \\ 1 & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 1.24: Definition of binary expression operator evaluation, originally from [1]

1.9.3 Statements

The judgment

$$\gamma \vdash^{\Gamma}_{stmt} \; s : \mu \rightleftharpoons \mu'$$

defines the meaning of statements. We say that under environment γ, statement s with class map Γ reversibly transforms store μ to store μ'. Figure 1.25a, 1.25b and 1.25c defines the operational semantics of RoopL++.

The following semantic rules have been simplified from the original RoopL semantics [1] to better accommodate the extended language.

The inference rule Skip defines the operational semantics of **skip** statements and has no effects on the store μ.

$$\frac{}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{skip} : \mu \rightleftharpoons \mu} \textsc{Skip}$$

$$\frac{\gamma \vdash^{\Gamma}_{stmt} \ s_1 : \mu \rightleftharpoons \mu' \qquad \gamma \vdash^{\Gamma}_{stmt} \ s_2 : \mu' \rightleftharpoons \mu''}{\gamma \vdash^{\Gamma}_{stmt} \ s_1 \ s_2 : \mu \rightleftharpoons \mu''} \textsc{Seq}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e \Rightarrow v \qquad [\![\odot]\!]\Big(\mu\Big(\gamma(x)\Big), v\Big) = v'}{\gamma \vdash^{\Gamma}_{stmt} \ x \odot = e \ : \ \mu \rightleftharpoons \mu[\gamma(x) \mapsto v']} \textsc{AssVar}$$

$$\frac{\mu\Big(\gamma(x_1)\Big) = v_1 \qquad \mu\Big(\gamma(x_2)\Big) = v_2}{\gamma \vdash^{\Gamma}_{stmt} \ x_1 \ \textbf{<=>} \ x_2 \ : \ \mu \rightleftharpoons \mu[\gamma(x_1) \mapsto v_2, \ \gamma(x_2) \mapsto v_1]} \textsc{SwpVar}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{expr} \ e_1 \not\Rightarrow 0 \qquad \gamma \vdash^{\Gamma}_{stmt} \ s_1 : \mu \rightleftharpoons \mu' \qquad \gamma \vdash^{\Gamma}_{loop} \ (e_1, s_1, s_2, e_2) : \mu' \rightleftharpoons \mu''}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{from} \ e_1 \ \textbf{do} \ s_1 \ \textbf{loop} \ s_2 \ \textbf{until} \ e_2 \ : \ \mu \rightleftharpoons \mu''} \textsc{LoopMain}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{expr} \ e_2 \not\Rightarrow 0}{\gamma \vdash^{\Gamma}_{loop} \ (e_1, s_1, s_2, e_2) \ : \ \mu \rightleftharpoons \mu} \textsc{LoopBase}$$

$$\frac{\begin{array}{cc} \langle \gamma, \mu \rangle \vdash^{\Gamma}_{expr} e_2 \Rightarrow 0 & \gamma \vdash^{\Gamma}_{stmt} s_2 : \mu \rightleftharpoons \mu' \\ \langle \gamma, \mu' \rangle \vdash^{\Gamma}_{expr} e_1 \Rightarrow 0 \qquad \gamma \vdash^{\Gamma}_{stmt} s_1 : \mu' \rightleftharpoons \mu'' & \gamma \vdash^{\Gamma}_{loop} (e_1, s_1, s_2, e_2) : \mu'' \rightleftharpoons \mu''' \end{array}}{\gamma \vdash^{\Gamma}_{loop} \ (e_1, s_1, s_2, e_2) \ : \ \mu \rightleftharpoons \mu'''} \textsc{LoopRec}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{expr} e_1 \not\Rightarrow 0 \qquad \gamma \vdash^{\Gamma}_{stmt} s_1 : \mu \rightleftharpoons \mu' \qquad \langle \gamma, \mu' \rangle \vdash^{\Gamma}_{expr} e_2 \not\Rightarrow 0}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{if} \ e_1 \ \textbf{then} \ s_1 \ \textbf{else} \ s_2 \ \textbf{fi} \ e_2 \ : \ \mu \rightleftharpoons \mu'} \textsc{IfTrue}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{expr} e_1 \Rightarrow 0 \qquad \gamma \vdash^{\Gamma}_{stmt} s_1 : \mu \rightleftharpoons \mu' \qquad \langle \gamma, \mu' \rangle \vdash^{\Gamma}_{expr} e_2 \Rightarrow 0}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{if} \ e_1 \ \textbf{then} \ s_1 \ \textbf{else} \ s_2 \ \textbf{fi} \ e_2 \ : \ \mu \rightleftharpoons \mu'} \textsc{IfFalse}$$

Figure 1.25a: Semantic inference rules for statements, modified from [1]

Rule Seq defines statement sequences where the store potentially is updated between each statement execution.

Rule AssVar defines reversible assignment in which variable identifier x under environment γ is mapped to the value v' resulting in an updated store μ'. For variable swapping SwpVar defines how value mappings between two variables are exchanged in the updated store.

For loops and conditionals, Rules LoopMain, LoopBase and LoopRec define the meaning of loop statements and IfTrue and IfFalse, similarly to the operational semantics of Janus, as

$$\gamma(this) = l \quad \mu(l) = l' \quad \mu(l') = \Big\langle c, (l_1,\ \ldots, l_m) \Big\rangle \quad \gamma(y_i) = l'_i$$

$$\Gamma(c) = \Big\langle \overbrace{(x_1,\ \ldots, x_m)}^{fields}, \overbrace{(\ldots, \textbf{method}\ q(t_1 z_1,\ \ldots, t_l z_k)\ s,\ \ldots)}^{methods} \Big\rangle$$

$$\frac{\gamma' = [this \mapsto l, x_1 \mapsto l_1,\ \ldots, l_m \mapsto v_m, z_1 \mapsto l'_1, \ldots, z_k \mapsto l'_k] \qquad \gamma' \vdash^{\Gamma}_{stmt}\ s\ : \mu \rightleftharpoons \mu'}{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{call}\ q(y_1,\ ...,\ y_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{Call}$$

$$\frac{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{call}\ q(y_1,\ ...,\ y_n)\ :\ \mu' \rightleftharpoons \mu}{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{uncall}\ q(y_1,\ ...,\ y_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{Uncall}$$

$$\gamma(x_0) = l \quad \mu(l) = l' \quad \mu(l') = \Big\langle c, (l_1,\ \ldots, l_m) \Big\rangle \quad \gamma(y_i) = l'_i$$

$$\Gamma(c) = \Big\langle \overbrace{(x_1,\ \ldots, x_m)}^{fields}, \overbrace{(\ldots, \textbf{method}\ q(t_1 z_1,\ \ldots, t_l z_k)\ s,\ \ldots)}^{methods} \Big\rangle$$

$$\frac{\gamma' = [this \mapsto l, x_1 \mapsto l_1,\ \ldots, l_m \mapsto v_m, z_1 \mapsto l'_1, \ldots, z_k \mapsto l'_k] \qquad \gamma' \vdash^{\Gamma}_{stmt}\ s\ : \mu \rightleftharpoons \mu'}{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{call}\ x_0 :: q(y_1,\ ...,\ y_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{CallObj}$$

$$\frac{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{call}\ x_0 :: q(y_1,\ ...,\ y_n)\ :\ \mu' \rightleftharpoons \mu}{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{uncall}\ x_0 :: q(y_1,\ ...,\ y_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{ObjUncall}$$

$$\Gamma(c) = \Big\langle \overbrace{(x_1,\ \ldots, x_m)}^{fields}, methods \Big\rangle \quad \gamma' = \gamma[x \mapsto 0] \quad l_0 \notin \text{dom}(\mu)\ \ldots\ l_m \notin \text{dom}(\mu)$$

$$\mu' = \mu\Big[\gamma'(x) \mapsto l_0, l_0 \mapsto \Big\langle c, (l_1,\ \ldots, 1_m) \Big\rangle, l_1 \mapsto 0,\ \ldots, l_m \mapsto 0\Big]$$

$$\frac{\gamma' \vdash^{\Gamma}_{stmt}\ s\ :\ \mu' \rightleftharpoons \mu'' \qquad \mu'' = \mu' \setminus \{\gamma'(x), l_0,\ \ldots, l_m\}}{\gamma \vdash^{\Gamma}_{stmt}\ \textbf{construct}\ c\ x \quad s \quad \textbf{destruct}\ x\ :\ \mu \rightleftharpoons \mu''}\ \textsc{ObjBlock}$$

Figure 1.25b: Semantic inference rules for statements, modified from [1] (cont)

presented in [3]. LoopMain is entered if e_1 is true and each iteration enters LoopRec until e_2 is false, in which case LoopBase is executed. Similarly, if e_1 and e_2 are true, rule IfTrue is entered, executing the then-branch of the conditional. If e_1 and e_2 are false, the IfFalse rule is executed and the else-branch is executed.

Rules Call, Uncall, CallObj and UncallObj respectively define local and non-local method invocations. For local methods, method q in current class c should be of arity n matching the number of arguments. The updated store μ' is obtained after statement body execution in the object environment. As local uncalling is the inverse of local calling, the direction of execution is simply reversed, and as such the input store a **call** statement serves as the output store of the **uncall** statement, similarly to techniques presented in [4, 3].

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_1 \Rightarrow v_1 \quad \langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_2 \Rightarrow v_2 \quad \gamma(x) = l \quad \mu(l)[v_1] = l' \quad \mu(l') = w \quad [\![\odot]\!](w, v_2) = w'}{\gamma \vdash^{\Gamma}_{stmt} \ x[e_1] \odot\!= e_2 \ : \ \mu \rightleftharpoons \mu[l' \mapsto w']} \textsc{AssArrElemVar}$$

$$\frac{\Gamma(c) = \Big\langle \overbrace{(x_1, \ \ldots, x_m)}^{fields}, methods \Big\rangle \qquad \gamma(x) = l \qquad l_0 \notin \ \mathrm{dom}(\mu) \ \ldots \ l_m \notin \ \mathrm{dom}(\mu)}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c \ x \ : \ \mu[l \mapsto 0] \rightleftharpoons \mu \left[l \mapsto l_0, l_0 \mapsto \Big\langle c, (l_1, \ \ldots, 1_m) \Big\rangle, l_1 \mapsto 0, \ \ldots, l_m \mapsto 0 \right]} \textsc{ObjNew}$$

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c \ x \ : \ \mu' \rightleftharpoons \mu}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ c \ x \ : \ \mu \rightleftharpoons \mu'} \textsc{ObjDelete}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e \Rightarrow n \qquad \gamma(x) = l \qquad \mu(l) = 0 \qquad l' \notin \ \mathrm{dom}(\mu)}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{new} \ a[e] \ x \ : \ \mu \rightleftharpoons \mu[l \mapsto l', l' \mapsto 0^n]} \textsc{ArrNew}$$

$$\frac{\dfrac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ a[e] \ x \ : \ \mu' \rightleftharpoons \mu}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ a[e] \ x \ : \ \mu \rightleftharpoons \mu'} \textsc{ArrDelete} \quad \gamma(x) = l \qquad \gamma(x') = l' \qquad \mu(l) = v}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{copy} \ c \ x \ x' \ : \ \mu[l' \mapsto 0] \leftrightharpoons \mu[l' \mapsto v]} \textsc{Copy}$$

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{copy} \ c \ x \ x' \ : \ \mu' \leftrightharpoons \mu}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{uncopy} \ c \ x \ x' \ : \ \mu \rightleftharpoons \mu'} \textsc{Uncopy}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_1 \Rightarrow v_1 \qquad \langle \gamma, \mu' \rangle \vdash^{\Gamma}_{stmt} \ e_2 \Rightarrow v_2 \qquad r \notin \ \mathrm{dom}(\mu) \qquad \mu' = \mu[r \mapsto v_1] \qquad \gamma[x \mapsto r] \vdash^{\Gamma}_{stmt} \ s : \mu' \rightleftharpoons \mu'' \qquad \mu'' = \mu'[r \mapsto v_2] \qquad \mu''' = \mu'' \setminus \{r\}}{\gamma \vdash^{\Gamma}_{stmt} \ \textbf{local} \ c \ x = e_1 \quad s \quad \textbf{delocal} \ x = e_2 \ : \ \mu \rightleftharpoons \mu'''} \textsc{LocalBlock}$$

Figure 1.25c: Extension to the semantic inference rules for statements in Roopl++

The statically scoped object blocks are defined in rule ObjBlock. The operation semantics of these blocks are similar to **local**-blocks from Janus. We add the reference x to a new environment and afterwards map location of x to the object tuple at location l_0, containing the locations of all object fields, all of which, along with l_0 must be unused in μ. The result store μ'' is obtained after executing the body statement s in store μ' mapping x to object reference at l_0, as long as all object fields are zero-cleared in μ'' afterwards. If any of these conditions fail, the object block statement is undefined.

Figure 1.25c shows the extensions to the semantics of Roopl with rules for **new**/**delete** and **copy**/**uncopy** statements, array element assignment and local blocks.

Rule AssArrElemVar defines reversible assignment to array elements. After evaluating expressions e_1 to v_1 and e_2 to v_2, the value at the location of variable $x[v_1]$ under environment γ is mapped to the value v_3 resulting in an updated store μ'.

Dynamic object construction and destruction is defined by rules ObjNew and ObjDelete. For construction, x must be bound to a location l. We then make location l point to a new pair consisting of the class name and a vector of m new locations mapping object fields to locations. For destruction, x is still bound to l return l to a null pointer. As with object blocks, it is the program itself responsible for zero-clearing object fields before destruction. If the object fields are not zero-cleared, the ObjDelete statement is undefined.

Array construction and destruction is very similar to object construction and destruction. The major difference is we bind the location to a vector of size equal to the evaluated expression result. For deletion, we return the location of x to a null pointer and remove the binding to the vector from the store.

Object and array referencing is defined by rules Copy and Uncopy. A reference is created and a new store μ' obtained by mapping x' to the reference r which x current maps to, if c matches the tuple mapped to the location l. A reference is removed and a new store μ' obtained if x and x' maps to the same reference r and x' then is removed from the store.

Local blocks are as previously mentioned, semantically similar to object blocks, where the memory location of variable x must be unused in the store μ. The updated store μ' contains location r mapped to the evaluated value of e_1, v_1 and the reference r, which x is bound to. The result store after body statement execution, μ'' must have r mapped to the expression value of e_2, v_2. Before the local block terminates, a third store update is executed, clearing the used memory locations, such that and r are mapped to zero and become unused again.

1.9.4 Programs

The judgment

$$\vdash_{prog} p \Rightarrow \sigma$$

defines the meaning of programs. The class p containing the main method is instantiated and the main function is executed with the partial function σ as the result, mapping variable identifiers to values, correlating to the class fields of the main class.

$$\frac{\begin{array}{c}\Gamma = \text{gen}(c_1,\ ...,\ c_n) \quad \Gamma(c_1) = \Big(\overbrace{\{\langle t_1, f_1\rangle,\ ...,\ \langle t_n, f_n\rangle\}}^{fields},\ methods\Big) \\ \Big(\textbf{method main ()}\ s\Big) \in methods \quad \gamma = [f_1 \mapsto 1,\ ...,\ f_i \mapsto i] \\ \mu = [1 \mapsto 0,\ ...,\ i \mapsto 0,\ this \mapsto i+1, i+1 \mapsto \langle c_1, \gamma\rangle] \qquad \gamma \vdash^{\Gamma}_{stmt}\ s\ :\ \mu \rightleftharpoons \mu'\end{array}}{\vdash_{prog}\ c_1\ \cdots\ c_n \Rightarrow (\gamma, \mu')}\ \textsc{Main}$$

Figure 1.26: Semantic inference rules for programs, originally from [1]

As with Roopl programs, the fields of the main method in the main class c are bound in a new environment, starting at memory address 1, as 0 is reserved for **nil**. The fields are zero-initialized

in the new store μ and address $i+1$ which maps to the new instance of c. After body execution, store μ' is obtained.

1.10 Program Inversion

In order to truly show that ROOPL++ in fact is a reversible language, we must demonstrate and prove local inversion of statements is possible, such that any program written in ROOPL++, regardless of context, can be executed in reverse. Haulund presented a statement inverter for ROOPL in [1], which maps statements to their inverse counterparts. Figure 1.27 shows the statement inverter, extended with the new ROOPL++ statements for construction/destruction and referencing copying/copy removal.

$$
\begin{aligned}
&\mathcal{I}[\![\textbf{skip}]\!] = \textbf{skip} && \mathcal{I}[\![s_1\ s_2]\!] = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!] \\
&\mathcal{I}[\![x\ \texttt{+=}\ e]\!] = x\ \texttt{-=}\ e && \mathcal{I}[\![x\ \texttt{-=}\ e]\!] = x\ \texttt{+=}\ e \\
&\mathcal{I}[\![x\ \texttt{\^{}=}\ e]\!] = x\ \texttt{\^{}=}\ e && \mathcal{I}[\![x\ \texttt{<=>}\ e]\!] = x\ \texttt{<=>}\ e \\
&\mathcal{I}[\![x[e_1]\ \texttt{+=}\ e_2]\!] = x[e_1]\ \texttt{-=}\ e_2 && \mathcal{I}[\![x[e_1]\ \texttt{-=}\ e_2]\!] = x[e_1]\ \texttt{+=}\ e_2 \\
&\mathcal{I}[\![x[e_1]\ \texttt{\^{}=}\ e_2]\!] = x[e_1]\ \texttt{\^{}=}\ e_2 && \mathcal{I}[\![x[e_1]\ \texttt{<=>}\ e_2]\!] = x[e_1]\ \texttt{<=>}\ e_2 \\
&\mathcal{I}[\![\textbf{new}\ c\ x]\!] = \textbf{delete}\ c\ x && \mathcal{I}[\![\textbf{copy}\ c\ x\ x']\!] = \textbf{uncopy}\ c\ x\ x' \\
&\mathcal{I}[\![\textbf{delete}\ c\ x]\!] = \textbf{new}\ c\ x && \mathcal{I}[\![\textbf{uncopy}\ c\ x\ x']\!] = \textbf{copy}\ c\ x\ x' \\
&\mathcal{I}[\![\textbf{call}\ q(\ldots)]\!] = \textbf{uncall}\ q(\ldots) && \mathcal{I}[\![\textbf{call}\ x :: q(\ldots)]\!] = \textbf{uncall}\ x :: q(\ldots) \\
&\mathcal{I}[\![\textbf{uncall}\ q(\ldots)]\!] = \textbf{call}\ q(\ldots) && \mathcal{I}[\![\textbf{uncall}\ x :: q(\ldots)]\!] = \textbf{call}\ x :: q(\ldots) \\
&\mathcal{I}[\![\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ e_2]\!] && = \textbf{if}\ e_1\ \textbf{then}\ \mathcal{I}[\![s_1]\!]\ \textbf{else}\ \mathcal{I}[\![s_2]\!]\ \textbf{fi}\ e_2 \\
&\mathcal{I}[\![\textbf{from}\ e_1\ \textbf{do}\ s_1\ \textbf{loop}\ s_2\ \textbf{until}\ e_2]\!] && = \textbf{from}\ e_1\ \textbf{do}\ \mathcal{I}[\![s_1]\!]\ \textbf{loop}\ \mathcal{I}[\![s_2]\!]\ \textbf{until}\ e_2 \\
&\mathcal{I}[\![\textbf{construct}\ c\ x\quad s\quad \textbf{destruct}\ x]\!] && = \textbf{construct}\ c\ x\quad \mathcal{I}[\![s]\!]\quad \textbf{destruct}\ x \\
&\mathcal{I}[\![\textbf{local}\ t\ x\ = e\quad s\quad \textbf{delocal}\ t\ x\ = e]\!] && = \textbf{local}\ t\ x = e\quad \mathcal{I}[\![s]\!]\quad \textbf{delocal}\ t\ x\ = e
\end{aligned}
$$

Figure 1.27: ROOPL++ statement inverter, extended from [1]

Program inversion is conducted by recursive descent over components and statements. A proposed extension to the statement inverter for whole-program inversion is retained in the ROOPL++ statement inverter. The extension covers a case that reveals itself during method calling. As a method call is equivalent to an uncall with the inverse method we simply change calls to uncalls during inversion, the inversion of the method body cancels out. The proposed extension, presented in [4, 1], simply avoids inversion of calls and uncalls, as shown in figure 1.28.

1.10.1 Invertibility of Statements

While the invertibility of statements remains untouched by the extensions made in ROOPL++, the following proof, originally presented in [1], has been included for completeness.