
© Copyright 2009 Andrew J. Craik
1 Introduction

In this technical report we present the abstract syntax and small step operational semantics of a language which captures the salient features of our extension of C with our ownership and effect annotations.

2 Abstract Syntax

Before we can begin to enumerate type rules or operational semantics, it is necessary to define the syntax of the language being discussed which we do in this section.

To make the syntax rules easier to read, we define the different symbols which
will be used in the rules:

\[
\begin{align*}
C & \quad \text{name of a class} \\
\Delta & \quad \text{method frame} \\
e & \quad \text{expression} \\
f & \quad \text{name of a field} \\
\Gamma & \quad \text{ownership checking environment} \\
K, I, J & \quad \text{actual context} \\
L & \quad \text{class definition} \\
M & \quad \text{method definition} \\
m & \quad \text{method name} \\
P & \quad \text{program} \\
\varphi & \quad \text{read \\& write effect set} \\
Q & \quad \text{context constraint} \\
T & \quad \text{type} \\
s & \quad \text{statement} \\
X & \quad \text{formal context parameters} \\
x & \quad \text{local variable or parameter name}
\end{align*}
\]

A program \( P \) is defined to be a set of classes \( L \) and static boot-strapping expression \( e \):

\[
P ::= L e
\]

User defined types and methods with context parameters can have constraints which restrict the actual contexts that can be supplied as value for the context parameters. There are four forms of these constraints:

\[
Q ::= \text{where } X_1 \neq X_2 \\
\quad \text{where } X_1 < X_2 \\
\quad \text{where } X_1 > X_2
\]

The definition of a class \( C_1 \) with formal context parameters \( \overline{X}_1 \) which optionally extends a class \( C_2 \) consists of a set of sub-contexts \( \overline{X}_2 \), a set of fields \( f \) with types \( T \) and a set of method declarations \( M \):

\[
L ::= \text{class } C_1 [\overline{X}_1] \text{ extends } C_2 \ Q \{ \text{subcontexts } \overline{X}_2; T f; M \}
\]

A type \( T \) contains the name of a class \( C \) and a set of actual context parameters \( \overline{K} \):

\[
T ::= C|\overline{K}|
\]

The declaration of a method with return type \( T \) named \( m \) with formal context parameters \( \overline{X} \) taking parameters \( \overline{x} \) of types \( T \) with maximum read effects of \( T \)
and maximum write effects of $\mathcal{J}$ where the contexts must satisfy the constraints $\mathcal{Q}$ listed if any:

$$M ::= Tm[\mathcal{X}] (\mathcal{T}_x) \text{reads}(\mathcal{I}) \text{writes}(\mathcal{J}) \mathcal{Q}$$

Expressions evaluate to values and consist of

$$e ::= e.m | \mathcal{K}|(\mathcal{T}) | e.f $$

$$| \text{new} \mathcal{C} | \mathcal{K} | \text{this} $$

$$| x | \text{null}$$

A statement consists of an expression, assignment, sequence of statements, a return, a foreach loop, and a local variable definition:

$$s ::= ; | e; $$

$$| e_1 = e_2 | \{\mathcal{S}\} $$

$$| \text{return} e; | \text{foreach} (\mathcal{T}_x \text{in} e) \{\mathcal{S}\} $$

$$| x = e; $$

Actual context parameters can be:

$$K, I, J ::= X|\text{this}|\text{this}.X|\text{world}$$

$\varphi$ is a tuple of read effects $\mathcal{I}$ and write effects $\mathcal{J}$:

$$\varphi ::= (\mathcal{I}, \mathcal{J})$$

Type checking takes place in an environment $\Gamma$ which holds mappings from variables to types as well as domination relationships between contexts:

$$\Gamma \in \{x \rightarrow T, \text{variable}$$

$$K \preceq K', \text{domination}$$

$$K \text{ valid contexts}\}$$

Lastly, we track the current method being typed, as specified by its name and parameters, in a method frame $\Delta$:

$$\Delta ::= \langle m, \mathcal{T} \rangle$$

$$| \emptyset$$
2.1 Helper Functions

There are a number of helper functions which we use to lookup information about methods, fields, and classes. The method function returns the return type, read and write effects, and formal context arguments of a method \(m\) in class \(C\) with arguments of types \(T\):

\[
\begin{align*}
\text{class } C\sqbracket{X_1} & \ldots \{ \ldots \text{method } m\sqbracket{X_2}\text{(T)} \} \text{reads } \langle \mathcal{I} \rangle \text{writes } \langle \mathcal{J} \rangle \text{where } \mathcal{Q} \ldots \\
\varphi & = \langle \mathcal{I}, \mathcal{J} \rangle \\
\text{method}(C\mid K_1\mid, m, T) & = \langle [K_1/X_1]T, [K_1/X_1]\varphi, X, Q \rangle
\end{align*}
\]

\[
\begin{array}{l}
\text{class } C \ldots \text{extends class } C'\{ \ldots M \ldots \} \\
\text{class } C'\sqbracket{X} \quad m(T) \notin M
\end{array}
\]

\[
\text{method}(C\mid K\mid, m, T) = \text{method}(C'\mid K_{1..}|X|\mid, m, T)
\]

\[
\begin{array}{l}
\text{class } C\{ \ldots M \ldots \} \\
m \notin M
\end{array}
\]

\[
\text{method}(C\mid K\mid, m, T) = \varnothing
\]

The field method returns the type of a field \(f\) in a class \(C\):

\[
\begin{align*}
\text{class } C\sqbracket{X} & \ldots \{ \ldots \text{field } f\ldots \} \\
\text{field}(C\mid K\mid, f) & = [K/X]T
\end{align*}
\]

\[
\begin{array}{l}
\text{class } C \ldots \text{extends class } C'\{ \ldots J \ldots \} \\
\text{class } C'\sqbracket{X} \quad f \notin J
\end{array}
\]

\[
\text{field}(C\mid K\mid, f) = \text{field}(C'\mid K_{1..}|X|\mid, f)
\]

\[
\begin{array}{l}
\text{class } C\{ \ldots J \ldots \} \\
f \notin J
\end{array}
\]

\[
\text{field}(C\mid K\mid, f) = \varnothing
\]

The subcontexts function returns the declared sub-contexts of the \textit{this} context in class \(C\):

\[
\text{subcontexts}(C) = \{ \ldots \}
\]
class C . . . extends C′ { ... subcontexts X . . . }

\[ \overline{X} = X \cup \text{subcontexts}(C') \]

subcontexts(C) = \overline{X}'

The owner function returns the owner context parameter for a type:

\[ \text{owner}(C|K|) = K_1 \]

Note also that we can obtain the type associated with a given context via an environment lookup as follows:

\[ \Gamma(K) = T \]
\[ \Gamma \vdash K : T \]

Lastly, we need to be able to validate context constraints:

\[
\begin{align*}
\text{validateConstraint}(\Gamma, Q) & = \forall q \in Q \text{validateConstraint}(\Gamma, q) \\
\text{validateConstraint}(\Gamma, K_1 \# K_2, \Gamma) & = K_1 \# K_2 \in \Gamma \\
\text{validateConstraint}(\Gamma, K_1 > K_2, \Gamma) & = K_1 > K_2 \in \Gamma \\
\text{validateConstraint}(\Gamma, K_1 < K_2, \Gamma) & = K_1 < K_2 \in \Gamma
\end{align*}
\]

3 Type Rules

In the following subsections the standard format of the typing statements will be:

\[ \Gamma ; K ; \Delta \vdash e : ? T \]

This statement is read as the expression \( e \) evaluates to type \( T \) with context side-effects \( \varphi \) under typing environment \( \Gamma \) with current context \( K \) and current method frame \( \Delta \). We now present the rules for the remainder of the syntactic constructs in our language in a top down manner.

3.1 Programs

To type a program we validate all of the classes defined in it and then type the bootstrap code and compute the program’s return type and effects based on it:

\[ \vdash \mathcal{L} \quad \emptyset ; \text{world} ; \emptyset \vdash e : ? T \]

\[ \overline{Le} : \varphi T \]
3.2 Class Declarations

To type check a class we must first ensure that the class it extends, if any, is valid. We must then ensure that the declared methods are valid, that fields are not overridden and that the declared formal context parameters only append additional parameters to the list declared by the super class.

\[
\Gamma = this \preceq X_1, this : C_1\langle this, X_2, |X|\rangle,
X, Q
\]
\[
\Gamma; this \vdash M \quad \Gamma; this ; \vdash X', T
\]
\[
\Gamma; this; \varnothing \vdash Q
\]
\[
\vdash \text{class } C_1\langle X \rangle \text{ extends } C_2\langle X' \rangle \text{ where } Q\{\text{subcontexts } X''; TM; M\}
\]

3.3 Method Definition

To validate a method definition, we first type its constituent statements in the current evaluation environment with the formal parameters bound to their type to determine the effect of executing the method body. The computed effects must be the same or smaller than the effects declared on the signature. Further, the declared effects must be the same or smaller than those of the method being overridden, if any. Lastly, the method must include its parent’s formal context parameters, but may optionally add its own parameters as well (validated by the \(\forall i \in 1..|X'| \ X'_i = X_1\) below).

\[
\Gamma; x : T, X; K; \langle m, T \rangle \vdash \{s\} : \langle I', J' \rangle \varnothing \quad \Gamma; K; \langle m, T \rangle \vdash I, J
\]
\[
\text{method}(\Gamma(\text{super}), m) = Tm[|X'|](T_x)\text{reads}(I')\text{writes}(J')
\]
\[
\Rightarrow T \preceq I \preceq T' \wedge J \preceq J' \preceq T' \wedge \forall i \in 1..|X'| X'_i = X_i \quad \Gamma; K; \langle m, T \rangle \vdash Q
\]
\[
\Gamma; K; \varnothing \vdash Tm[|X|](T_x)\text{reads}(I)\text{writes}(J)\text{where } Q\{s\}
\]

\[
\Gamma; x : T; K; \langle m, T \rangle \vdash \{s\} : \langle I', J' \rangle \varnothing \quad \Gamma; K; \langle m, T \rangle \vdash I, J
\]
\[
\text{super} = \varnothing \lor \text{method}(\Gamma(\text{super}), m) = \varnothing
\]
\[
\Rightarrow T \preceq I \wedge J \preceq J \quad \Gamma; K; \langle m, T \rangle \vdash Q
\]
\[
\Gamma; K; \varnothing \vdash Tm[|X|](T_x)\text{reads}(I)\text{writes}(J)\text{where } Q\{s\}
\]
3.4 Loops

The foreach loop considered earlier in this paper can be typed in this system. We require the collection in the loop to have a next() method which returns an object with a type which is included in the declared element type:

\[
\Gamma; K; \Delta \vdash e : \varphi' \quad \text{class } C[X]\ldots
\]

\[
\text{method}(C[K], \text{next}, \emptyset) = \langle T', \varphi'', \emptyset \rangle
\]

\[
\varphi = \varphi' \cup \varphi'' \quad \Gamma, x \rightarrow T; K; \Delta \vdash \{\pi\} : \varphi'' \quad \emptyset
\]

\[
\Gamma; K; \Delta \vdash \text{foreach}(T x \in e)\{\pi\} : \varphi \quad \emptyset
\]

3.5 Statement Blocks and Expressions

To type a block of statements we simply type each of the statements; this no result type because statements only produce side-effects:

\[
\forall s_i \in \pi \Gamma; K; \Delta \vdash s_i : \varphi_i \land \varphi = \bigcup_{s_i} \varphi_i
\]

\[
\Gamma; K; \Delta \vdash \{\pi\} : \varphi \quad \emptyset
\]

When typing an expression as a statement, we discard the result type:

\[
\Gamma; K; \Delta \vdash e : T
\]

\[
\Gamma; K; \Delta \vdash e : \varphi \quad \emptyset
\]

3.6 Return Statement

To type a return statement, we must ensure that the type of the expression to be returned is a valid subtype of the current method’s return type. Finally, the effect of evaluating the return is the effect of evaluating the expression to be returned.

\[
\Gamma \vdash K : C[K] \quad \Gamma; K; \langle m, T \rangle \vdash e : T'
\]

\[
\text{method}(C[K], m, T) = \langle T, \varphi', \emptyset \rangle
\]

\[
\Gamma; K; \langle m, T \rangle \vdash \text{return } e : \varphi \quad \emptyset
\]
3.7 Method Invocation

To type a method invocation we first compute the type and effect of evaluating the expression $e$. We can then compute the types and effects of computing the method’s actual parameters. We then lookup the size of the method’s context parameter list and ensure a valid actual context parameter has been supplied for each. The effect of the invocation is the union of these read-effects and write-effects combined with the method’s declared effects raised to the current context after substituting actual contexts for formal context parameters.

\[
\begin{align*}
\Gamma; K; \Delta &\vdash e : \phi \ C \mid K_2 \quad \Gamma; K; \Delta \vdash \tau : \mathcal{T} \\
\text{method}(C \mid K_2, m, T) &= \langle T, \varphi, \bar{X}_1, \bar{Q} \rangle \\
|\bar{X}_1| &= |\bar{K}_1| \quad \Gamma; K; \Delta \vdash K_1 \\
\varphi &= \varphi' \cup \varphi \cup [\bar{K}_1/\bar{X}_1] \varphi \\
\text{validateConstraint}(\Gamma[K_1/X_1]\bar{Q}) \\
\Gamma; K; \Delta &\vdash \text{this}\.m \mid K_1 \mid (\bar{e}) : \phi \ T
\end{align*}
\]

3.8 Object Instantiation

Calling a constructor is largely the same as calling a method except that for simplicity there are no formal context parameters to bind and there is no receiver computation required. Note that the type of the object being created is validated to ensure that the correct number of context parameters are supplied.

\[
\begin{align*}
\Gamma; K; \Delta &\vdash C \mid \bar{K} \\
\text{class} C[\bar{X}] \ldots \bar{Q} \ldots \\
\Gamma; K; \Delta &\vdash \text{new} C \mid \bar{K} \mid \phi' \ C \mid \bar{K} \\
\Gamma; K; \Delta &\vdash C \mid \bar{K}
\end{align*}
\]
3.9 Reading Local Variables & Parameters

Reading a local variable or parameter does not cause a context side-effect; a context side-effect occurs when the contents of an object are read.

\[
\frac{\Gamma(x) = T}{\Gamma; \vdash x : \langle \emptyset, \emptyset \rangle T}
\]

Reading the local self-reference variable this has no side-effect for the same reasons:

\[
\frac{\Gamma(\text{this}) = T}{\Gamma; \vdash \text{this} : \langle \emptyset, \emptyset \rangle T}
\]

3.10 Writing Local Variables & Parameters

Writing to a local variable or parameter does not cause a side-effect since the stack is not aliased. To type a local assignment, we type the expression and ensure the computed type is a subtype of that declared by the variable or parameter.

\[
\frac{\Gamma; K; \Delta \vdash e : \varphi T' \quad \Gamma(x) = T}{\Gamma; K; \Delta \vdash x = e : \varphi \emptyset}
\]

3.11 Reading Fields

When reading a field, we must first compute the type of the object to which the field belongs. The effect of the statement will then be the total read and write effects of evaluating the object reference expression as well as a read of the context or sub-context in which the field is located.

\[
\frac{\Gamma; K; \Delta \vdash e : \varphi' T' \mid \text{field}(C, f) = T \quad \varphi = \varphi' \cup \langle K_1, \emptyset \rangle}{\Gamma; K; \Delta \vdash \text{this}.f : \varphi T}
\]
3.12 Writing Fields

To compute the effect of writing to a field we must compute the types and effects of evaluating the object reference expression and the new value for the field. These effects are then raised to the current context and the owner of the field’s object is added to the write effects:

\[
\begin{align*}
\Gamma; K; \Delta \vdash e : \varphi &\quad |\quad C | K | \quad \text{field}(C, f) = T \\
\varphi &= \varphi' \cup \langle \text{owner}(T), \varnothing \rangle \\
\hline 
\Gamma; K; \Delta \vdash e.f : \varphi' T
\end{align*}
\]

3.13 Validating Context Constraints

We validate context constraints by checking that the contexts are valid and that the constraint does not violate any existing constraints:

\[
\begin{align*}
\Gamma; K; \Delta \vdash K_1 &\quad |\quad \Gamma; K; \Delta \vdash K_2 \\
K_1 < K_2 \notin \Gamma &\quad |\quad K_1 > K_2 \notin \Gamma \\
\hline 
\Gamma; K; \Delta \vdash K_1 \neq K_2
\end{align*}
\]

\[
\begin{align*}
\Gamma; K; \Delta \vdash K_1 &\quad |\quad \Gamma; K; \Delta \vdash K_2 \\
K_1 \neq K_2 \notin \Gamma &\quad |\quad K_1 > K_2 \notin \Gamma \\
\hline 
\Gamma; K; \Delta \vdash K_1 < K_2
\end{align*}
\]

\[
\begin{align*}
\Gamma; K; \Delta \vdash K_1 &\quad |\quad \Gamma; K; \Delta \vdash K_2 \\
K_1 < K_2 \notin \Gamma &\quad |\quad K_1 \neq K_2 \notin \Gamma \\
\hline 
\Gamma; K; \Delta \vdash K_1 > K_2
\end{align*}
\]
3.14 Validate Contexts

Lastly, we present rules for validating contexts and types. For a context to be valid, it must be in the set of currently visible contexts:

\[
\Gamma \vdash K : C \mid K \quad \text{method}(C \mid K \mid m, T) = \langle \lambda \cdot \cdot , X, \cdot \rangle \\
K' \in \{\text{this, world}\} \cup X
\]

\[
\Gamma; K; \langle m, T \rangle \vdash K'
\]

\[
\Gamma \vdash K : C \mid K' \in K \cup \{K, \text{world}\}
\]

\[
\Gamma; K; \vdash K'
\]

\[
\Gamma \vdash K : C \mid X \in \text{subcontexts}(C)
\]

\[
\Gamma; K; \vdash K'
\]

Domination relationships are either stored in the environment, a produce of owner ordering, transitivity, \textit{world} being the top context, or self domination:

\[
\Gamma \vdash K \preceq K' \in \Gamma \\
\Gamma \vdash K \preceq K' \\
\Gamma \vdash K \preceq \text{world} \\
\Gamma \vdash K \preceq K'' \\
\Gamma \vdash K \preceq K'
\]

To validate a type we ensure the number of actual context parameters matches the number of formal context parameters and that the supplied contexts are valid:

\[
\text{class } C[\overline{X}] \ldots \overline{Q} \ldots | \overline{X} = |\overline{K}| \quad \Gamma; K; \Delta \vdash \overline{X} \\
\Gamma; K; \Delta \vdash \overline{Q}
\]

\[
\Gamma; K; \Delta \vdash C[\overline{K}]
\]

We make sub-typing transitive:

\[
\vdash T < T''. \quad \vdash T'' < T' \implies \vdash T < T'
\]
We make sub-typing reflexive:

\[ \vdash T <: T \]

We provide standard subsumption:

\[ \frac{e : T' \quad T' <: T}{e : T} \]

We permit type coercion through sub-typing:

\[ \frac{\text{class } C_1[\overline{X_1}] \text{ extends } C_2 \ldots \quad \text{class } C_2[\overline{X_2}] \ldots \quad \vdash C_2 | \overline{K_1}, \overline{X_2} | <: D | \overline{K'} |}{\vdash C_1 | \overline{K} | <: D | \overline{K'} |} \]

4 Dynamic Semantics

In this section we present the small step operation semantics for the language presented in the previous section.

4.1 Operational Semantics Syntax

In the small step operation semantics presented later in this section we will be using the following abstract syntax:

- \( l \) typed location
- \( c \) context name
- \( K ::= c \mapsto l \) actual contexts
- \( o ::= \{ f \mapsto l, K \} \) objects
- \( H ::= l \mapsto o \) heap
- \( S ::= \Delta \) stack
- \( \Delta ::= \{ x \mapsto l \} \) stack frame

In addition to the above syntax, we import the abstract syntax of the language itself in the presentation of the type rules. Note that the standard expression used in the development of the operational semantics is of the form:

\[ H, S: e \rightarrow H', S': e' \]
The above statement is read as the expression $e$ evaluated with heap $H$ and stack $S$ reduces to another expression $e'$ with a heap $H'$ and a new stack $S'$.

We also supply a helper function which looks up a context name in the heap and stack to return a location:

$$\text{lookup}(H, S, c) = H(c)$$

$$S = S', \Delta \quad e \in \Delta$$

$$\text{lookup}(H, S, c) = \Delta(c)$$

$$\text{lookup}(H, S, c) = \emptyset$$

### 4.2 Small Step Operational Semantics

We now present the small step operational semantics for our language in a bottom up manner. These rules codify the operation of the language.

#### 4.2.1 Reading Parameters & Variables

To read a parameter passed into the current method, we look it up in the current stack frame:

$$S = S'; \Delta$$

$$H; S; x \rightarrow H; S; \Delta(x)$$

#### 4.2.2 Reading this

To read the `this` variable, we simply lookup its value in the stack:

$$S = S'; \Delta$$

$$H; S; \text{this} \rightarrow H; S; \Delta(\text{this})$$
4.2.3 Reading Fields

To select the value of a field we must first compute the target of the lookup. Once we have the lookup target, we simply lookup the value for the field in the heap using the computed target:

\[
H; S; l.f \rightarrow H; S; l(f)
\]

\[
H; S; e \rightarrow H'; S'; e'
\]

\[
H; S; e.f \rightarrow H'; S'; e'.f
\]

4.2.4 Writing Parameters & Variables

To assign a value to a variable or parameter we must first reduce the right-hand side to produce the value to be stored. We then store the value into the appropriate location in the stack.

\[
S = S''; \Delta' \Delta' = \Delta \Delta'(x) = l \quad S' = S''; \Delta'
\]

\[
H; S; x = l; \rightarrow H; S'
\]

\[
H; S; e \rightarrow H'; S'; e'
\]

\[
H; S; x = e \rightarrow H'; S'; x = e'
\]

4.2.5 Writing Fields

To assign a value to a field we first reduce the right-hand side to produce the value to be stored. We then reduce the target of the field select. Finally, we store the value in the appropriate field in the heap and return:

\[
H' = H[l \mapsto H(l)[f \mapsto l']]
\]

\[
H; S; l.f = l' \rightarrow H'; S
\]

\[
H; S; e \rightarrow H'; S'; e'
\]

\[
H; S; e.f = l \rightarrow H'; S'; e'.f = l
\]

\[
H; S; e_2 \rightarrow H'; S'; e'_2
\]

\[
H; S; e_1.f = e_2 \rightarrow H'; S'; e_1.f = e'_2
\]
4.2.6 Local Variable Declarations

For a local variable declaration we first compute the value of the right-hand side. Once we have computed the value, we set the value in the current stack:

\[
\frac{}{H; S; x = l \rightarrow H; S; \{x \mapsto l\}}
\]

\[
\frac{H; S; e \rightarrow H'; S'; e'}{H; S; x = e \rightarrow H'; S'; x = e'}
\]

4.2.7 Reducing Block of Statements

The statements in a statement block are sequentially reduced in the order supplied. Note that we short-circuit evaluation if a return value is produced early (the third rule):

\[
\frac{}{H; S; \{\overline{s}\} \rightarrow H; S; \overline{s}}
\]

\[
\frac{H; S; s \rightarrow H'; S'}{H; S; s; \overline{s} \rightarrow H'; S'; \overline{s}}
\]

\[
\frac{H; S; s \rightarrow H'; S'; l}{H; S; s; \overline{s} \rightarrow H'; S'; \overline{l}}
\]

4.2.8 Loops

Reducing a loop in the absence of primitive Boolean values requires that we terminate the loop when the next() method returns an object representing null; not unlike the handling of null in Ruby.

\[
\frac{}{H; S; l.\text{next}(\ ) \rightarrow H'; S'; \text{null}}
\]

\[
\frac{}{H; S; \text{foreach}(x \text{ in } l) \rightarrow H'; S'}
\]
4.2.9 Method Invocation

To invoke a method we first compute the target of the call and then we compute the arguments. Once we have completed this, we push a new stack frame with this bound to the target of the call, the actual contexts supplied as part of the method call, and the methods formal parameters bound to its actual computed parameters, and then we evaluate the body of the method.

\[
\frac{\vdash \overline{l} : T \quad \vdash \overline{t} : T \quad \text{class } C \ldots \{ \ldots m \ldots (T\overline{x})\{\overline{v}\} \ldots \}}{K = \overline{v} \mapsto \text{lookup}(H, S, c) \quad S' = S, \{\text{this } \mapsto l, \text{this } \mapsto \overline{l}, K, \forall i \in 1..|\overline{l}| \ x_i \mapsto l_i\}} {H; S; l.m \mid c \mid (\overline{t}) \rightarrow H; S'; \overline{v}}
\]

\[
\frac{H; S; e \rightarrow H'; S'; \overline{e}' \quad H; S; \overline{t}; \overline{e} \rightarrow H'; S'; \overline{e}' \quad \overline{t} \rightarrow H'; S'; \overline{e}'.m \mid c \mid (\overline{v})}{H; S; e \rightarrow H'; S'; \overline{e}'}
\]

4.2.10 Return Statements

When we encounter a return statement we compute the value to be returned. Once that is completed we pop the stack frame and resume the previous method with the computed value.

\[
S = S', \Delta \\
\frac{H; S; \text{return } l \rightarrow H'; S'; l}{H; S; \text{return } l \rightarrow H'; S'; l}
\]
\[
\frac{H; S; e \rightarrow H'; S'; e'}{H; S; \text{return } e \rightarrow H'; S'; \text{return } e'}
\]

### 4.2.11 Object Creation

Initializing a new object is the same as invoking a method except that we must allocate the new object on the heap and initialize its fields before evaluating the constructor body:

\[
\begin{align*}
\text{class } C \ldots \{ \ldots \mathcal{J} \ldots C(Tx) \ldots \{ x \} \ldots \} & \quad H' = H, \mathcal{J} \mapsto \{ x \mapsto \emptyset \} \\
S' = S, \{ \text{this} \mapsto \mathcal{L}_t, \text{this} \mapsto \mathcal{L}_t, x \mapsto \text{lookup}(H, S, c) \} & \quad H; S; \text{new } C \mid c \mid \rightarrow H'; S'
\end{align*}
\]

### 4.2.12 Program

Finally, to reduce a program we reduce the bootstrap code to the final return value:

\[
\begin{align*}
H; S; \mathcal{L}_l & \rightarrow l \\
H = \{ \} & \quad S = \{ \} \quad H; S; e \rightarrow H'; S'; e' \\
\emptyset; \emptyset; \mathcal{L}_e & \rightarrow H'; S'; \mathcal{L}_e' \\
H; S; e \rightarrow H'; S'; e' & \quad H; S; \mathcal{L}_e \rightarrow H'; S'; \mathcal{L}_e'
\end{align*}
\]

### 4.2.13 Rules to Enforce Well-Formed Heaps

We now present two final rules which ensure the heap is well formed along with the locations contained therein:

### 4.2.14 The rootwalk helper method

In the following sections we will show the semantics of how to check the relationship between two contexts. In this development of the language semantics
we will use a the most simplistic pointer chasing algorithm which minimizes memory usage and algorithmic complexity at the expense of runtime performance. Alternative implementations of this method are possible and one of these is discussed in our paper where we optimize for performance not memory usage[ref]. To facilitate the use of this pointer chasing algorithm we now define the rootwalk method which allows us to generate the list of nodes between a node and the world root.

\[
\begin{align*}
  o[K_1] &= \text{world} \\
  \text{rootwalk}(o,H) &= \emptyset
\end{align*}
\]

\[
\begin{align*}
  o[K_1] &= l \\
  \text{rootwalk}(o,H) &= \{l\} \cup \text{rootwalk}(H[l],H)
\end{align*}
\]

4.2.15 Context Equality

For a context \(o_1\) to be equal to another context \(o_2\) then all of the ancestors of \(o_1\) must also be ancestors of \(o_2\).

\[
\begin{align*}
  \text{rootwalk}(o_1,H) \neq \text{rootwalk}(o_2,H) \\
  H; S; o_1 = o_2 \rightarrow \text{false}
\end{align*}
\]

\[
\begin{align*}
  \text{rootwalk}(o_1,H) = \text{rootwalk}(o_2,H) \\
  H; S; o_1 = o_2 \rightarrow \text{true}
\end{align*}
\]

4.2.16 Testing for Context Ancestor Relationships

For a context \(o_1\) is an ancestor of \(o_2\) then all of the ancestors of \(o_1\) must also be ancestors of \(o_2\).

\[
\begin{align*}
  \text{rootwalk}(o_1,H) \nsubseteq \text{rootwalk}(o_2,H) \\
  H; S; o_1 > o_2 \rightarrow \text{false}
\end{align*}
\]

\[
\begin{align*}
  \text{rootwalk}(o_1,H) \subseteq \text{rootwalk}(o_2,H) \\
  H; S; o_1 > o_2 \rightarrow \text{true}
\end{align*}
\]
4.2.17 Testing for Context Descent Relationships

For a context \( o_1 \) is a descendent of \( o_2 \) then all of the ancestors of \( o_2 \) must also be ancestors of \( o_1 \).

\[
\text{rootwalk}(o_2, H) \not\subseteq \text{rootwalk}(o_1, H) \\
\text{H; S; } o_1 < o_2 \rightarrow \text{false}
\]

\[
\text{rootwalk}(o_2, H) \subseteq \text{rootwalk}(o_1, H) \\
\text{H; S; } o_1 < o_2 \rightarrow \text{true}
\]

4.2.18 Testing for Context Disjointness

If two contexts are disjoint, their lists of parents must not have a subset relationship. If such a subset relationship exists, then some form of parent-child relationship exists which means the contexts are not disjoint.

\[
\text{rootwalk}(o_1, H) \subseteq \text{rootwalk}(o_2, H) \lor \text{rootwalk}(o_2, H) \subseteq \text{rootwalk}(o_1, H) \\
\text{H; S; } o_1 \# o_2 \rightarrow \text{false}
\]

\[
\text{rootwalk}(o_1, H) \not\subseteq \text{rootwalk}(o_2, H) \land \text{rootwalk}(o_2, H) \not\subseteq \text{rootwalk}(o_1, H) \\
\text{H; S; } o_1 \# o_2 \rightarrow \text{true}
\]