A Model of Garbage Collection for OO Languages

Rob Hunter and Shriram Krishnamurthi

Brown University
Providence, RI
Contact: `rob@cs.brown.edu`

1 Abstract

Traditional programming language models ignore garbage collection. When semanticists leave it as an implementation detail separate from the model, any properties they prove are less meaningful because the model does not accurately describe the implemented system. Some language models do incorporate garbage collection, but none of these models handle side effects, objects, or sub-typing, which are each realistic language features.

We present the first semantics for an object-oriented language that explicitly models garbage collection. We prove that the model is sound and that inserting garbage collection steps into evaluation preserves the original semantics. Our work provides a framework for future research on semantic foundations for systems that need to make high-level guarantees in the presence of complex memory management operations.

2 Introduction

Assume we are given a proof of the correctness of a programming language *model*. This proof does not guarantee that the *implementation* is correct. Having a model is still beneficial, of course, because it can give system designers and users confidence in the correctness of the implementation. As a model more closely approximates the actual implementation, confidence in the implementation increases.

Moving a model closer to the implementation, however, generally introduces both complexity and inflexibility. As a result, semanticists usually opt for a simple model that can cover a wide range of implementations. The most disturbing consequence of this choice, however, is that a program, well-typed with respect to a simple language model, may still "go wrong." That is, if a programmer incorrectly implements a portion of the language that the *model* designers considered an implementation detail, the model and implementation will be inconsistent. A richer model that explicitly handled this detail would likely expose the problem when a semanticist attempted, for example, a soundness proof.

ClassicJava [4] is a semantics that captures the object-oriented complexities of Java in the relatively simple style of the well-studied semantics and type systems of Scheme and ML ([3, 5, 10]). Many modern languages, Java included, manage memory automatically. Unfortunately, neither ClassicJava, nor any other object-oriented models, incorporate garbage collection.

2.1 A Motivating Example

Reiss, et. al. [9] present a Java-based system that supports dynamic upgrades. As motivation for our work, we now briefly describe their system. Suppose we would like to upgrade a Java class while a program is running. In particular, for some class A, we want to provide a new implementation, class A'. Thus, for example, if the program executes the code `x = new A` after the upgrade, then `x` should be bound to an instance of A', and not A.

Any dynamic upgrade system needs a mechanism for handling "old" instances of upgraded classes. Consider an instance a of class A. After the upgrade to A', the code of the original class A should never be executed. But what if, after the upgrade, the program calls method `someMethod()` on a? Reiss, et. al. [9] handle this at upgrade time by patching the dynamic dispatch table (vtable) of class A to trap all method calls. The patched vtable is filled with stub methods which, when invoked, check to see if an upgraded version of the class exists. If not, then a special upgrade method is called on the current instance. This upgrade method converts old instances to upgraded instances. When the upgrade is complete, the stub executes the upgraded method code on the upgraded instance.

In the previous example, calling `someMethod()` on a results in the execution of a stub version of `someMethod`. This stub calls the upgrade method on instance a, which produces a', a proper instance of class A'. A pointer to a' is stored so that future `a.someMethod()` calls will not require a call to the upgrade method. As a last step, the stub invokes `someMethod()` on a'.

The system, as described, is inefficient. After the first call to an old object (during which we upgrade it to a new version), each time we access the object the user pays a run-time penalty due to the level of indirection through the vtable stubs. That is, a call to an old instance results in the execution of a stub function, which in turn executes the proper method.

To fix this inefficiency, the *garbage collector* collapses these indirections and updates pointers from old objects to their upgraded instances. In more general terms, the implementers of this system have augmented the standard Java garbage collector with extra, non-trivial operations. In particular, in a system like this, garbage collection is no longer an implementation detail—it is an integral part of the system's behavior. Therefore, any model of this system that omits garbage collection is unacceptable.

This paper does not provide a semantics for this dynamic upgrade system, but it does provide an object-oriented model that makes garbage collection explicit, which is the necessary first step in writing such a semantics.

2.2 Overview

We will refer to our extension of ClassicJava as ClassicJava$^{\text{gc}}$. In Section 4 we describe ClassicJava$^{\text{gc}}$. In Section 5 we show that our garbage collection rule is correct. The type soundness proof and the definition of ClassicJava are included in the Appendix.

3 Related Work

We base our semantics on ClassicJava [4], which does not model garbage collection. As a starting point for our formalization of garbage collection, we use the calculi of Morrisett, et. al. [7]. Their work, however, models collection in a lambda-calculus language with explicit heap allocation. They do not consider state, objects, or sub-typing. We extend their work to model a realistic object-oriented language. Morrisett and Harper [8] also extend Morrisett, et. al. [7]. They make certain details explicit which allows them to formalize, for example, tail-call elimination. Their work focuses on a language with parametric polymorphism, and they provide a

P	$=$	$\mathit{defn}^* \; e$
defn	$=$	**class** c **extends** c **implements** i^* $\{\mathit{field}^* \; \mathit{meth}^*\}$ $\mid$
		interface i **extends** i^* $\{\mathit{meth}^*\}$
field	$=$	$t \; \mathit{fd}$
meth	$=$	$t \; \mathit{md} \; (\mathit{arg}^*) \; \{\mathit{body}\}$
arg	$=$	$t \; \mathit{var}$
body	$=$	e $\mid$ **abstract**
e	$=$	**new** $c \mid \mathit{var} \mid \mathsf{null} \mid e\underline{\;:\;c}.\mathit{fd} \mid e\underline{\;:\;c}.\mathit{fd} = e$
		$e.\mathit{md}\;(e^*) \mid$ $\textbf{super}\underline{\;\equiv\;\textbf{this}\;:\;c}.\mathit{md}\;(e^*)$
		view $t \; e$ $\mid$ **add1** $e \mid n$
var	$=$	var $\mid$ **this**
c	$=$	a class name or *object*
i	$=$	an interface name or Empty
fd	$=$	a field name
md	$=$	a method name
t	$=$	$c \mid i \mid \mathsf{nat}$
n	$=$	a natural number

Table 1: The syntax of ClassicJava$^{\mathrm{gc}}$

model for tag-free garbage collection in this polymorphic context. They do not consider state, objects, or sub-typing.

In Section 2.1, we presented a dynamic upgrading system as motivation for our work. Hicks [6] describes a different system for dynamic upgrades, which is lower-level than the system of Reiss et. al. [9]. In particular, Hicks' system does not have a notion of objects, nor does he use garbage collection as a means to efficient upgrading. Hence, while he does provide a formal model of his system, his model does not overlap with our work.

4 ClassicJava$^{\mathrm{gc}}$

In describing ClassicJava$^{\mathrm{gc}}$, we will assume an understanding of ClassicJava. For those unfamiliar with ClassicJava, its typing and reduction rules are included in the Appendix. The set of programs in ClassicJava$^{\mathrm{gc}}$ is defined by the syntax in Table 1. A program P is a set of class and interface definitions followed by an expression. The typing rules of Tables A5 and A6 (and the rules which we define in this section) assign types to a subset of syntactically correct programs.

A number of predicates and relations ensure that the classes and interfaces have certain structural properties before assigning a type to the program (see Tables A1, A2, A3 and A4). For example, P is not typable if $\textsc{FieldOncePerClass}(P)$ does not hold. Similarly, consider a field-access expression of the form $e.\mathit{fd}$. The $[\mathsf{get}, \vdash_e]$ typing rule (see Table A5) will not assign a type t to this expression unless the relation $\langle c.\mathit{fd}, t\rangle \; \boxed{\textsc{FieldInClass}_P} \; t'$ holds, and e has type t'. In words, this relation means that a field, with name fd and type t, is declared in class c and is available in class t' through inheritance. In this paper, all relations are written as $a \; \boxed{\textsc{Relation}_P} \; b$, which means the relation $\boxed{\textsc{Relation}_P}$, which is parameterized over a program P, relates a and b.

ClassicJava$^{\mathrm{gc}}$ is a reduction semantics. Evaluation is the application of rewriting steps (reduction rules). Evaluation halts when we reach a value or an error.[1] The reduction rules are shown in Table A7. Since

[1]Evaluation also halts if it gets stuck (i.e., if no reduction rule applies even though the current expression is neither a value

(standard variables)	var	$\in$	Var	
(object locations)	$object$	$\in$	Loc	
(values)	v	$\in$	Val	$::= n \mid object \mid \mathsf{null}$
(errors)	$error$	$\in$	$Error$	$::=$ error: bad cast | error: deref null

Table 2: Notations of ClassicJava$^{\mathrm{gc}}$

programs create new objects during evaluation, evaluation is the rewriting of a tuple $\langle e, \mathcal{S}\rangle$, where e is the expression and $\mathcal{S}$ is the *store* which maps object names (locations) to object values.

In addition to assigning types to expressions, the typing rules also annotate expressions with extra information. These annotations are called *elaborations*. Consider the consequent of the [get, $\vdash_e$] judgment, which is $P, \Gamma \vdash_{\mathsf{e}} e.fd \Longrightarrow \underline{e' : c}.fd : t$. Note that $\Longrightarrow$ is the elaboration symbol, and that the underlined text is the result of elaboration. In this example, we annotate the get-expression with c, which is the class in which the programmer declared field fd.

Once type checking is complete, we are left with elaborated expressions. To prove correctness properties, we need to be able to apply typing rules during evaluation. Because the evaluation rules operate on elaborated expressions, we need new typing rules. Consequently, we define (just as in [4]), for each $\vdash_{\mathsf{e}}$ and $\vdash_{\mathsf{s}}$ rule, a corresponding $\vdash_{\underline{\mathsf{e}}}$ or $\vdash_{\underline{\mathsf{s}}}$ rule. These new rules are exactly the same, except that they apply to elaborated expressions.

The $\xrightarrow{new}$ reduction rule reduces $\langle \mathsf{E}[\mathbf{new}\ c], \mathcal{S}\rangle$ to $\langle \mathsf{E}[object], \mathcal{S}[object \mapsto \langle c, \mathcal{F}\rangle]\rangle$. In the resulting tuple, *object* is a previously unused (memory) location, and $\langle c, \mathcal{F}\rangle$ is the representation of an object (defined precisely in Table A7). Note that the store, $\mathcal{S}$, was augmented with a new location-value binding. At any point in evaluation, this binding may become *garbage*. Roughly, an object is garbage if it is not needed during the rest of evaluation. Garbage collection, for our purposes, will be the removal of bindings from the store. As in [7], we introduce a *garbage collection reduction rule*, which we can apply at any stage of evaluation.

Intuitively, if removing a particular binding from the store will not introduce any free locations[2] with respect to the current expression and store, then we can remove that binding. We formalize this intuition with the $\xrightarrow{fl}$ rule below. This rule depends on the definition of FL (see Table 3), which formally captures the notion of a free location. The symbol $\uplus$ denotes union, with the restriction that the two argument sets are disjoint.

$$\langle e, \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{fl} \langle e, S_1\rangle \qquad [\xrightarrow{fl}]$$
$$\text{where } FL(e, S_1) = \emptyset$$

Type soundness is not sufficient to show that the garbage collection rule is correct. We will need a way to check that the return values across several evaluations of the same program are equivalent, independent of how garbage collection is interleaved. Our equivalence metric needs to respect the intensional nature of garbage collection. We cannot use location names as a test for equality, since locations differ across different runs. We cannot use structural equality, because this would fail to capture object identity and sharing relationships.

To address this difficulty, we follow the lead of Morrisett, et. al. [7], and require that all programs return a natural number. Our measurement of equality (and hence, measurement of the correctness of a garbage collection scheme) is simple—does a program *with* garbage collection return the same number as the program *without* it?

Note that this is not a restriction on the expressiveness of programs. If we really want to return an object

nor an error). However, we will show that this is impossible.

[2] By *free location* we mean an *object* that may be reachable during evaluation, and yet is unbound in the store.

$$\begin{aligned}
FL\,(i) &= \emptyset \\
FL\,(object) &= \{object\} \\
FL\,(\mathbf{new}\ c) &= \emptyset \\
FL\,(\mathsf{null}) &= \emptyset \\
FL\,(e\underline{\ :c'\ }.fd) &= FL\,(e) \\
FL\,(e_1\underline{\ :c\ }.fd = e_2) &= FL\,(e_1) \cup FL\,(e_2) \\
FL\,(e.md(e_1,...,e_n)) &= FL\,(e) \cup FL\,(e_1) \cup ... \cup FL\,(e_n) \\
FL\,(\mathbf{super}\underline{\equiv\ *\ :\ c}.md(e_1,...,e_n)) &= FL\,(e_1) \cup ... \cup FL\,(e_n) \\
FL\,(\mathbf{view}\ t\ e) &= FL\,(e) \\
FL\,(\mathbf{add1}\ e) &= FL\,(e) \\
\\
FL\,(object_1 = \langle c_1, \mathcal{F}_1\rangle, ..., object_n = \langle c_n, \mathcal{F}_n\rangle) &= [FL\,(\langle c_1, \mathcal{F}_1\rangle)\ \cup\ ...\ \cup\ FL\,(\langle c_n, \mathcal{F}_n\rangle)] \setminus \{object_1, ...object_n\} \\
FL\,(\langle c, \mathcal{F}\rangle) &= \cup_{(c'.fd=v)\in\mathcal{F}} FL\,(v) \\
FL\,(e, S) &= (FL\,(e) \cup\ FL\,(S)) \setminus dom\,(S)
\end{aligned}$$

Table 3: Definition of the *FL* function. $*$ above means either **this** or *object*.

(as we were able to in ClassicJava) we can encode the creation of the object in the number we return. For example, consider a class c which has f nat fields, each of size b bits. We can represent any specific instance of c with an $f \times b$ bit number.

The [prog, $\vdash_p$] typing rule requires that the type of all programs be numbers. We add numbers to our syntax, and we assert the existence of a type nat with the typing rule [nat-type, $\vdash_t$]. We also need the typing rule [nat, $\vdash_e$] which declares that numbers are of type nat:

$$\begin{array}{ll}
P \vdash_{\mathsf{t}} \mathsf{nat} & [\mathsf{nat\text{-}type}, \vdash_{\mathsf{t}}] \\
P, \Gamma \vdash_{\mathsf{e}} n \Longrightarrow n : \mathsf{nat} & [\mathsf{nat}, \vdash_{\mathsf{e}}]
\end{array}$$

We need a way of manipulating natural numbers, and thus we provide a simple **add1** function, which has the following typing and reduction rules:

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Longrightarrow e' : \mathsf{nat}}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{add1}\ e \Longrightarrow \mathbf{add1}\ e' : \mathsf{nat}} \qquad [\mathsf{add1}, \vdash_{\mathsf{e}}]$$

$$P \vdash \langle \mathsf{E}\,[\mathbf{add1}\ n]\,, \mathcal{S}\rangle \longrightarrow \langle \mathsf{E}\,[n+1]\,, \mathcal{S}\rangle \qquad [\xrightarrow{add1}]$$

5 Correctness of ClassicJava$^{\mathbf{gc}}$

We formulate a type soundness theorem which shows that well-typed programs do not get stuck. The proof, which uses standard progress and subject reduction lemmas, is in the Appendix.

Theorem 1 (Type Soundness) *If $\vdash_{\mathsf{p}} P \Longrightarrow P' : \mathsf{nat}$ and $P' = defn_1\ ...\ defn_n\ e$, then either*

1. *Evaluation diverges; or*
2. *Evaluation results in an* error *configuration; or*

3. $P' \vdash \langle e, \emptyset \rangle \longrightarrow^* \langle \mathsf{null}, \mathcal{S} \rangle$*; or*

4. $P' \vdash \langle e, \emptyset \rangle \longrightarrow^* \langle n, \mathcal{S} \rangle$

5.1 The Garbage Collection Rule is Correct

To establish correctness, we need to show that a particular program returns the same number, regardless of how $\xrightarrow{fl}$ is interleaved in evaluation. Specifically, we will use *Kleene Equivalence* to measure semantic preservation.

In the following sections, G is some set of reduction rules. R represents the set of reduction rules from Table A7 (R does *not* contain $\xrightarrow{fl}$). $R + fl$ is the set of *all* reduction rules. We use r to mean an element of R. ST is an evaluation state of the form $\langle e, \mathcal{S} \rangle$.

Definition 1 (Full Evaluation) *$ST \Downarrow_G ST'$ means $ST \longrightarrow^*_G ST'$ and there is no $\xrightarrow{r}$ rule in G such that $ST' \longrightarrow ST''$, for some ST''.*[3]

Definition 2 (Kleene Equivalence)
$(ST_1, G_1) \cong (ST_2, G_2)$ means $ST_1 \Downarrow_{G_1} \langle n, \mathcal{S} \rangle$ iff $ST_2 \Downarrow_{G_2} \langle n, \mathcal{S}' \rangle$.

By showing that we can "postpone" applications of the $\xrightarrow{fl}$ rule to the end of evaluation, we conclude that adding the $\xrightarrow{fl}$ reduction rule does not interfere with evaluation.

Theorem 2 (Correctness) *For all programs P, for all evaluation states ST (where $\vdash_{\mathsf{p}} P \Longrightarrow P' : \mathsf{nat}$, $P' = defn^* e$, and $ST = \langle e, \mathcal{S} \rangle$, for some $\mathcal{S}$), $(ST, R) \cong (ST, R + fl)$.*

Proof: Fix some P and ST which satisfy the conditions of the theorem statement. Assume $ST \Downarrow_R \langle n, \mathcal{S} \rangle$. By not performing any $\xrightarrow{fl}$ steps, we can perform this same evaluation under $R + fl$. Thus, $ST \Downarrow_{R+fl} \langle n, \mathcal{S} \rangle$. Conversely, assume $ST \Downarrow_{R+fl} \langle n, \mathcal{S} \rangle$. We can write out the full evaluation as follows:

$$ST \xrightarrow{R+fl} ST_1 \xrightarrow{R+fl} ST_2 \xrightarrow{R+fl} \ldots \xrightarrow{R+fl} ST_{n-1} \xrightarrow{R+fl} \langle n, \mathcal{S} \rangle$$

We can rearrange this evaluation using Lemma 1:

$$ST \xrightarrow{r_1} ST'_1 \xrightarrow{r_2} \ldots \xrightarrow{r_k} ST'_k \xrightarrow{fl} \ldots \xrightarrow{fl} ST'_{n-1} \xrightarrow{fl} \langle n, \mathcal{S} \rangle$$

Since an $\xrightarrow{fl}$ step never changes the expression and can only remove bindings, it must be the case that $ST'_k = \langle n, S \uplus S' \rangle$, for some S'. Therefore, $ST \Downarrow_R \langle n, S \uplus S' \rangle$. ■

Lemma 1 (Repeated Postponement) *If $ST_0 \xrightarrow{R+fl} ST_1 \xrightarrow{R+fl} \ldots \xrightarrow{R+fl} ST_n$, then $ST_0 \xrightarrow{r_1} ST'_1 \xrightarrow{r_2} \ldots \xrightarrow{r_k} ST'_k \xrightarrow{fl} ST'_{k+1} \xrightarrow{fl} \ldots \xrightarrow{fl} ST'_{n-1} \xrightarrow{fl} ST_n$.*

Proof: We induct on the number of rewrite rules in this evaluation. If no evaluations are performed, then the lemma is vacuously true. Now assume this holds for $n - 1$ rewriting steps. Applying the assumption, reorder these evaluations so that all the R rules occur first. If the nth rewrite rule is an $\xrightarrow{fl}$ rule, we are done. If the nth rule is an R rule, we can write

[3]If we had used $\xrightarrow{r+fl}$ in place of $\xrightarrow{r}$ in this definition, then full evaluation would be unsatisfiable since $\xrightarrow{fl}$ can *always* be applied.

$ST_0 \xrightarrow{r_1} ST_1' \xrightarrow{r_2} ... \xrightarrow{r_k} ST_k' \xrightarrow{fl} ST_{k+1}' \xrightarrow{fl} ... \xrightarrow{fl} ST_{n-2}' \xrightarrow{fl} ST_{n-1} \xrightarrow{r_n} ST_n$

By repeatedly applying Lemma 2, we can rewrite the evaluation in the following way:

$ST_0 \xrightarrow{r_1} ST_1' \xrightarrow{r_2} ... \xrightarrow{r_k} ST_k' \xrightarrow{r_n} ST_{k+1}'' \xrightarrow{fl} ... \xrightarrow{fl} ST_{n-2}'' \xrightarrow{fl} ST_{n-1}' \xrightarrow{fl} ST_n.$ ■

Lemma 2 (Postponement) *If* $ST_1 \xrightarrow{fl} ST_2 \xrightarrow{r} ST_3$*, then* $\exists\ ST_2'$ *s.t.* $ST_1 \xrightarrow{r} ST_2' \xrightarrow{fl} ST_3$.

Proof: We case on the possible forms of ST_1:

Case [*new*]:

Assume $\langle \mathsf{E}[\mathbf{new}\ c], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{fl} \langle \mathsf{E}[\mathbf{new}\ c], S_1\rangle \xrightarrow{new} \langle \mathsf{E}[object], S_1 \uplus \{object \mapsto \langle c, \mathcal{F}\rangle\}\rangle$, where $FL(\mathsf{E}[\mathbf{new}\ c], S_1) = \emptyset$, $\mathcal{F} = \{c'.fd \mapsto \mathsf{null} \mid ...\}$, and $object \notin dom(S_1)$.

By definition, $\langle \mathsf{E}[\mathbf{new}\ c], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{new} \langle \mathsf{E}[object'], \mathcal{S}_1 \uplus \mathcal{S}_2 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\}\rangle \equiv ST_2'$ where $object' \notin \mathcal{S}_1 \uplus \mathcal{S}_2$ and $\langle c, \mathcal{F}\rangle$ is as before.

$FL(\mathsf{E}[object'], S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})$

$= (FL(\mathsf{E}[object']) \cup FL(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})) \setminus dom(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})$

$= [FL(\mathsf{E}[\mathbf{new}\ c]) \cup FL(object') \cup FL(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})] \setminus [FL(\mathbf{new}\ c) \cup dom(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})]$, by Lemma 4.

$= [(FL(\mathsf{E}[\mathbf{new}\ c]) \cup FL(S_1)) \setminus dom(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})]$
$\cup [(FL(object') \cup FL(\langle c, \mathcal{F}\rangle)) \setminus dom(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})]$

$= [FL(object') \cup FL(\langle c, \mathcal{F}\rangle)] \setminus dom(S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\})$ (by the assumption)

$= \emptyset$ (since, by the construction of $object'$, $FL(\langle c, \mathcal{F}\rangle) = \emptyset$).

Thus, $ST_2' \xrightarrow{fl} \langle \mathsf{E}[object'], S_1 \uplus \{object' \mapsto \langle c, \mathcal{F}\rangle\}\rangle \equiv ST_X$. Since $object \notin \mathcal{S}_1$ and $object' \notin \mathcal{S}_1$, ST_X is related by alpha to $\langle \mathsf{E}[object], S_1 \uplus \{object \mapsto \langle c, \mathcal{F}\rangle\}\rangle$. We conclude that $ST_2' \xrightarrow{fl} \langle \mathsf{E}[object], S_1 \uplus \{object \mapsto \langle c, \mathcal{F}\rangle\}\rangle$.

Case [*get*]:

Assume $\langle \mathsf{E}[object\underline{\ :c'\ }.fd], S_1 \uplus \mathcal{S}_2\rangle \xrightarrow{fl} \langle \mathsf{E}[object\underline{\ :c'\ }.fd], S_1\rangle \xrightarrow{get} \langle \mathsf{E}[v], S_1\rangle$ where $FL(\mathsf{E}[object\underline{\ :c'\ }.fd], S_1) = \emptyset$, $S_1(object) = \langle c, \mathcal{F}\rangle$, and $\mathcal{F}(c'.fd) = v$. Since $object \in dom(S_1)$, $\langle \mathsf{E}[object\underline{\ :c'\ }.fd], S_1 \uplus \mathcal{S}_2\rangle \xrightarrow{get} \langle \mathsf{E}[v], S_1 \uplus \mathcal{S}_2\rangle \equiv ST_2'$.

$(FL(\mathsf{E}[v]) \cup FL(S_1)) \setminus dom(S_1)$

$= [FL(\mathsf{E}[object\underline{\ :c'\ }.fd]) \cup FL(v) \cup FL(S_1)] \setminus (dom(S_1) \cup FL(object\underline{\ :c'\ }.fd))$ (by Lemma 4)

$= [(FL(\mathsf{E}[object\underline{\ :c'\ }.fd]) \cup FL(S_1)) \setminus (dom(S_1) \cup FL(object\underline{\ :c'\ }.fd))] \cup [FL(v) \setminus (dom(S_1) \cup FL(object\underline{\ :c'\ }.fd))]$

$= FL(v) \setminus (dom(S_1) \cup FL(object\underline{\ :c'\ }.fd))$, by the assumption.

Combining the fact that $FL(v) \subseteq FL(\langle c, \mathcal{F}\rangle)$ with Lemma 5, we conclude that the previous line equals $\emptyset$. Thus, $ST_2' \xrightarrow{fl} \langle \mathsf{E}[v], S_1\rangle$

Case $[\mathit{set}]$:

Assume $\langle \mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{\mathit{fl}} \langle \mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v], S_1\rangle \xrightarrow{\mathit{set}} \langle \mathsf{E}[v], S_1'\rangle$,
where $\mathit{FL}(\mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v], S_1) = \emptyset$, $S_1(\mathit{object}) = \langle c, \mathcal{F}\rangle$, and $S_1' = S_1[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]$.

Since $\mathit{object} \in \mathit{dom}(S_1)$, $\langle \mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v], S_1 \uplus S_2\rangle \xrightarrow{\mathit{set}} \langle \mathsf{E}[v], S_1' \uplus S_2\rangle \equiv ST_2'$.

$(\mathit{FL}(\mathsf{E}[v]) \cup \mathit{FL}(S_1')) \setminus \mathit{dom}(S_1')$

$= [\mathit{FL}(\mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v]) \cup \mathit{FL}(v) \cup \mathit{FL}(S_1')] \setminus (\mathit{dom}(S_1') \cup \mathit{FL}(\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v))$ (by Lemma 4)

$= [(\mathit{FL}(\mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v]) \cup \mathit{FL}(S_1')) \setminus (\mathit{dom}(S_1') \cup \mathit{FL}(\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v))]$
$\quad \cup\ [\mathit{FL}(v) \setminus (\mathit{dom}(S_1') \cup \mathit{FL}(\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v))]$ $\qquad (\star)$

Using Lemma 3 and the fact that $[\mathit{FL}(\mathsf{E}[\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v], S_1) = \emptyset]$, we get that $\mathit{FL}(v) \subseteq \mathit{dom}(S_1)$

$\implies \mathit{FL}(\langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle) \setminus \mathit{dom}(S_1') = \emptyset$

$\implies \mathit{FL}(S_1') = \emptyset$

$\implies (\star) = \mathit{FL}(v) \setminus (\mathit{dom}(S_1') \cup \mathit{FL}(\mathit{object}\underline{\ :c'\ }.\mathit{fd} = v))$

$= \emptyset$.

Thus, $ST_2' \xrightarrow{\mathit{fl}} \langle \mathsf{E}[v], S_1'\rangle$.

Case $[\mathit{call}]$:

Assume $\langle \mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{\mathit{fl}} \langle \mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)], S_1\rangle$
$\xrightarrow{\mathit{call}} \langle \mathsf{E}[e[\mathbf{this} \leftarrow \mathit{object}, \mathit{var}_1 \leftarrow v_1, ..., \mathit{var}_n \leftarrow v_n]], S_1\rangle$, where $\mathit{FL}(\mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)], S_1) = \emptyset$, $S_1(\mathit{object}) = \langle c, \mathcal{F}\rangle$, and $\langle \mathit{md}, T, \mathit{var}_1\ ...\ \mathit{var}_n, e\rangle$ $\boxed{\textsc{MethInClass}_P}$ c.

Since $\mathit{object} \in \mathit{dom}(S_1)$, $\langle \mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle$
$\xrightarrow{\mathit{call}} \langle \mathsf{E}[e[\mathbf{this} \leftarrow \mathit{object}, \mathit{var}_1 \leftarrow v_1, ..., \mathit{var}_n \leftarrow v_n]], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \equiv ST_2'$.

$\mathit{FL}(\mathsf{E}[e[\mathbf{this} \leftarrow \mathit{object}, \mathit{var}_1 \leftarrow v_1, ..., \mathit{var}_n \leftarrow v_n]], S_1)$

$= [\mathit{FL}(\mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)]) \cup \mathit{FL}(e[...]) \cup \mathit{FL}(S_1)] \setminus [\mathit{dom}(S_1) \cup \mathit{FL}(\mathit{object}.\mathit{md}(v_1, ..., v_n))]$
(by Lemma 4)

$= (\mathit{FL}(\mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)]) \cup \mathit{FL}(S_1)) \setminus [\mathit{dom}(S_1) \cup \mathit{FL}(\mathit{object}.\mathit{md}(v_1, ..., v_n))]$
$\cup [\mathit{FL}(e[...]) \setminus (\mathit{dom}(S_1) \cup \mathit{FL}(\mathit{object}.\mathit{md}(v_1, ..., v_n)))]$

$= [\mathit{FL}(e[...])] \setminus [\mathit{dom}(S_1) \cup \mathit{FL}(\mathit{object}) \cup \mathit{FL}(v_1) \cup ... \cup \mathit{FL}(v_n)]$

$= [\mathit{FL}(v_1) \cup ... \cup \mathit{FL}(v_n)] \setminus [\mathit{dom}(S_1) \cup \mathit{FL}(\mathit{object}) \cup \mathit{FL}(v_1) \cup ... \cup \mathit{FL}(v_n)]$ (by Lemma 6)

$= \emptyset$.

Thus, $ST_2' \xrightarrow{\mathit{fl}} \langle \mathsf{E}[e[\mathbf{this} \leftarrow \mathit{object}, \mathit{var}_1 \leftarrow v_1, ..., \mathit{var}_n \leftarrow v_n]], S_1\rangle$.

Case $[\mathit{super}]$:

The proof for this case is almost exactly the same as the proof for $[\mathit{call}]$.

Case $[cast]$:

Assume $\langle \mathsf{E}[\textbf{view}\ t\ object], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{fl} \langle \mathsf{E}[\textbf{view}\ t\ object], \mathcal{S}_1\rangle \xrightarrow{cast} \langle \mathsf{E}[object], \mathcal{S}_1\rangle$,
where $\mathcal{S}_1(object) = \langle c, \mathcal{F}\rangle$ and $c\ \boxed{\text{SubType}P}\ t$.

$\langle \mathsf{E}[\textbf{view}\ t\ e], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{cast} \langle \mathsf{E}[object], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle$, where $(\mathcal{S}_1 \uplus \mathcal{S}_2)(object) = \langle c, \mathcal{F}\rangle$ and $c\ \boxed{\text{SubType}P}\ t$.

Combining the fact that $FL(\textbf{view}\ t\ object) = FL(object)$ with Lemma 4, we get that $FL(\mathsf{E}[object], \mathcal{S}_1) = \emptyset$.
Therefore, $\langle \mathsf{E}[object], \mathcal{S}_1 \uplus \mathcal{S}_2\rangle \xrightarrow{fl} \langle \mathsf{E}[object], \mathcal{S}_1\rangle$.

■

Lemma 3 (FL and Evaluation Context)
$FL(E[e]) = \bigcup_n FL(n)$, where n ranges over the nodes of the syntax tree $\mathsf{E}[e]$.

Proof: Show by inducting on the height of the syntax tree. ■

Lemma 4 (FL Substitution)
$FL(E[e]) = [FL(E[e']) \cup FL(e)] \setminus FL(e')$

Proof: Follows immediately from Lemma 3. ■

Lemma 5 (Enclosed Fields) *If $S = \{..., object \mapsto \langle c, \mathcal{F}\rangle, ...\}, FL(S) = \emptyset$, and some set $A \subseteq FL(\langle c, \mathcal{F}\rangle)$ then $A \subseteq dom(S)$.*

Proof: $FL(S) = \emptyset \implies FL(\langle c, \mathcal{F}\rangle) \subseteq dom(S) \implies A \subseteq dom(S)$. ■

Lemma 6 (FL and Methods)
If $P, t_0 \vdash_m t\ md(t_1\ var_1, ..., t_n\ var_n)\{e\}$, then $FL(e[v_1/var_1, ..., v_n/var_n]) \subseteq FL(v_1) \cup ... \cup FL(v_n)$.

Proof: The body e can contain no object locations, and therefore, all free locations must be introduced by the substitution. ■

5.2 The Correctness of Diverging Programs

Non-terminating programs are useful only to the extent that they can communicate with the user or another program. For example, an operating system, the canonical infinite loop, is useless if it does not interact with the user.

In this section we sketch how our existing theoretical machinery might be used to derive correctness properties for non-terminating programs. We assume the existence of a special instance of a class which appears in all programs at the same location, *output*. The instance has a field of type `nat` named *datum*. We further assume that *object* is never garbage collected. Intuitively, the program passes a message to the user by mutating the field *datum* in *output*.

Consider two runs of the same program in which only the second run performs garbage collection. Intuitively, we would like to verify that the *datum* fields of both runs are always the same. Since our only notion of time is through reduction rules, we provide the following definition which allows us to speak about a particular point of evaluation across runs even when we interleave garbage collection:

Definition 3 (Intermediate Evaluation) *$ST \Downarrow_G^i \langle e, \mathcal{S}\rangle$ means that the program state, after i reduction steps, is $\langle e, \mathcal{S}\rangle$. Each rule in G counts as 1 reduction step, except for $\xrightarrow{fl}$, which counts as 0 steps.*

Given the object *output* and the previous definition, we can concisely formulate a correctness property which applies to all well-typed programs:

Property 1 *For all programs P, (where $\vdash_p P \Longrightarrow P' : \mathsf{nat}$, $P' = defn^* e_0$, and $ST_0 = \langle e_0, \emptyset\rangle$), if $ST_0 \Downarrow_R^i \langle e, \mathcal{S}_i\rangle$ and $ST_0 \Downarrow_{R+fl}^i \langle e', \mathcal{S}'_j\rangle$, then $\mathcal{F}(c.datum) = \mathcal{F}'(c.datum)$, where $\mathcal{S}_i(output) = \langle c, \mathcal{F}\rangle$ and $\mathcal{S}'_j(output) = \langle c, \mathcal{F}'\rangle$.*

Proof: (Sketch) We assume $ST_0 \xrightarrow{r_1} ST_1 \xrightarrow{r_2} ... \xrightarrow{r_i} ST_i$ and $ST_0 \xrightarrow{R+fl} ST'_1 \xrightarrow{R+fl} ... \xrightarrow{R+fl} ST'_j$, where $j \geq i$. Applying Lemma 1 and alpha-renaming, we conclude $ST_0 \xrightarrow{r_1} ST_1 \xrightarrow{r_2} ... \xrightarrow{r_i} ST_i \xrightarrow{fl} ST''_{i+1} \xrightarrow{fl} ... \xrightarrow{fl} ST''_{j-1} \xrightarrow{fl} ST'_j$. Assume that $\mathcal{F}(c.data) \neq \mathcal{F}(c.data)$. Only $\xrightarrow{set}$ (an R rule) can change the value of a field, and thus the first time the fields carry different values must occur with one of the rules $r_1, ..., r_i$. But we have already shown that the first $i+1$ states of both evaluations are equivalent up to alpha-renaming, and so this is a contradiction. ∎

6 Conclusion

Our work was inspired by an implementation of dynamic upgrading [9]. Its authors claim that their system preserves both performance and correctness. We wanted to prove the latter. To do so, however, we needed to be able to model their implementation (which significantly modifies the garbage collector to preserve performance).

Unfortunately, there is a mismatch between the previously existing garbage collection models [7, 8] and the dynamic upgrading system. The former are variants of the lambda calculus; the latter is an enhanced form of Java. Hence, it is not possible to directly adapt these existing models to yield a close approximation of the upgrading system.

Therefore, we designed a model of Java with explicit support for garbage collection. We present the model, prove it type sound, and then prove that our addition of the garbage collection reduction rule is also sound. In particular, we show that this rule does not affect the return values of programs. Finally, we propose a correctness property which applies to non-terminating programs.

We still face interesting and significant open problems. First, our garbage collection reduction rule is atomic, and thus our model is inadequate for systems with concurrent garbage collection (for a recent example, see [2]). Second, the next Java standard will include parametric polymorphism to complement the subtype polymorphism already present in the language. This standard is based on the work of Bracha, et. al. on Generic Java [1]. As run-time systems implement this extension, we will have to extend our proofs to handle this richer type context.

References

[1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to Java programming. *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1998.

[2] P. Chang. *Scalable Real-time Parallel Garbage Collection for Symmetric Multiprocessors.* PhD thesis, Carnegie Mellon University, 2001.

[3] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[4] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University, June 1999. Previous version appeared in *ACM Symposium on Principles of Programming Languages*, 1998.

[5] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 1998.

[6] M. Hicks. *Dynamic Software Updating.* PhD thesis, University of Pennsylvania, 2001.

[7] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. *ACM Conference on Functional Programming and Computer Architecture*, 1995.

[8] G. Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. *Higher-Order Operational Techniques in Semantics*, pages 175–226, 1998.

[9] D. Reiss, G. Cooper, and S. Krishnamurthi. Efficient run-time support for type safe dynamic class upgrades. Technical Report CS-01-08, Brown University, 2001.

[10] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1991.

Appendix

In this section, we prove that ClassicJava^{gc} is sound. For completeness, in the tables that follow, we also present the typing and reduction rules of the original ClassicJava.

Since evaluation may introduce *object*s as expressions, we need to be able to type check these new expressions in order for the soundness proof to work. Thus, as is standard in the literature, we introduce a new typing rule[4]:

$$\frac{object \in dom\,(\Gamma)}{P, \Gamma \vdash_{\underline{\mathsf{e}}} object : \Gamma(object)}\ [\mathsf{obj}]$$

As in Flatt, et. al. [4], we replace [wcast, $\vdash_e$] and [ncast, $\vdash_e$] with a new casting relation. This relation allows us to assign types to all intermediate expressions of the reduction:

$$\frac{P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t'}{P, \Gamma \vdash_{\underline{\mathsf{e}}} \textbf{view}\ t\ e : t}\ [\mathsf{cast}, \vdash_{\underline{\mathsf{e}}}]$$

We also need to ensure that the typing environment and store each self-consistent and consistent with each other. We use the following condition to express this property:

Definition 4 (Environment-Store Consistency)
$P, \Gamma \vdash_\sigma \mathcal{S} \Leftrightarrow$
$[\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$
Σ_1: $\Longrightarrow \Gamma(object) = c$
Σ_2: *and* $dom\,(\mathcal{F}) = \Big\{c_1.fd \mid \langle c_1.fd, c_2\rangle\ \boxed{\text{FieldInClass}_P}\ c\Big\}$
Σ_3: *and* $rng\,(\mathcal{F}) \subseteq dom\,(\mathcal{S})\ \cup\ \{\mathit{null}\}\ \cup\ Nat$
Σ_4: *and* $\Big(\mathcal{F}(c_1.fd) = object'\ and\ \langle c_1.fd, c_2\rangle\ \boxed{\text{FieldInClass}_P}\ c\Big)$
$\Longrightarrow \Big(\big(\mathcal{S}(object') = \langle c', \mathcal{F}'\rangle\big) \Longrightarrow c'\ \boxed{\text{SubType}_P}\ c_2\Big)$
Σ_5: *and* $\Big(\mathcal{F}(c_1.fd) = n\ and\ \langle c_1.fd, c_2\rangle\ \boxed{\text{FieldInClass}_P}\ c\Big)$
$\Longrightarrow (c_2 = \mathit{nat})]$
Σ_6: *and* $object \in dom\,(\Gamma) \Longrightarrow object \in dom\,(\mathcal{S})$
Σ_7: *and* $dom\,(\mathcal{S}) \subseteq dom\,(\Gamma)$

Note that the **this** in a source code **super** expression may become *object*, for some *object*, during evaluation.[5]. As in [4], we need to ensure that the following property of **super** expressions always holds:

Definition 5 (Well-Formed Super Calls)
SuperOK(e) $\Leftrightarrow$ *For all* $\textbf{super}\underline{\equiv\ e_0\ :\ c\ }.md(e_1, ..., e_n)$ *in* e, *either* $e_0 = \textbf{this}$ *or* $e_0 = object$, *for some object.*

We are now ready to prove type soundness. For convenience, we duplicate the theorem statement here:

Theorem 1 (Type Soundness) *If* $\vdash_{\mathsf{p}} P \Longrightarrow P' : \mathit{nat}$ *and* $P' = defn_1\ ...\ defn_n\ e$, *then either*

1. *Evaluation diverges; or*

[4]This rule would not appear in any implementation of this model. It only exists to make the soundness proof work.
[5]This can happens under substitution, since **this** is a variable.

2. Evaluation results in an error *configuration; or*

3. $P' \vdash \langle e, \emptyset \rangle \longrightarrow^* \langle \mathit{null}, \mathcal{S} \rangle$*; or*

4. $P' \vdash \langle e, \emptyset \rangle \longrightarrow^* \langle n, \mathcal{S} \rangle$

Proof: If evaluation diverges then we are done. Thus, assume evaluation does not diverge. We first show:

If at least s steps of evaluation apply and the resulting state (after s steps) is $\langle e_s, \mathcal{S}_s \rangle$ and *not* an error configuration, then $\exists\ \Gamma$ s.t. $P', \Gamma \vdash_{\underline{\mathsf{e}}} e_s : \mathsf{nat}$ and $P', \Gamma \vdash_\sigma \mathcal{S}_s$ and $\textsc{SuperOK}(e_s)$. $(\star)$

When $s = 0$: From $[\mathsf{prog}, \vdash_{\mathsf{p}}]$ we know $P', \{\} \vdash_{\underline{\mathsf{e}}} e : \mathsf{nat}$, $P', \emptyset \vdash_\sigma \emptyset$, and $\textsc{SuperOK}(e)$ is trivially true. Thus, $(\star)$ is true for $s = 0$. Now assume $(\star)$ holds for $s = k$. By Subject Reduction, then, we know that $(\star)$ holds for $s = k + 1$. Thus $(\star)$ is proved. We know evaluation halts in a finite number s steps. By combining $(\star)$ with Progress, we prove Type Soundness. ■

Lemma 7 (Subject Reduction) *If* $P, \Gamma \vdash_{\underline{\mathsf{e}}}\ e : \mathsf{nat}$ *and* $P, \Gamma \vdash_\sigma S$ *and* $\textsc{SuperOK}(e)$ *and* $\langle e, \mathcal{S} \rangle \longrightarrow \langle e', \mathcal{S}' \rangle$*, then* e' *is an* error *configuration or there exists a* Γ' *s.t.*

1. $P, \Gamma' \vdash_{\underline{\mathsf{e}}} e' : \mathsf{nat}$*;*

2. $P, \Gamma' \vdash_\sigma S'$*; and*

3. $\textsc{SuperOK}(e')$

Proof: We case on every possibility of $\longrightarrow$. If e' is an error configuration, then we're done. Otherwise, we construct a Γ' and show that it satisfies the required properties. For property 3, we point you to Flatt, et. al. [4], except for the $\xrightarrow{fl}$ case, in which it is trivially true.

Case $[\xrightarrow{new}]$: Let $\Gamma' = \Gamma\, [\mathit{object} \mapsto c]$.

1. We know $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[\mathbf{new}\ c] : \mathsf{nat}$. From $\xrightarrow{new}$, $\mathit{object} \notin \mathit{dom}\,(S) \implies$ (by Σ_6) $\mathit{object} \notin \mathit{dom}\,(\Gamma) \implies$ (by Lemma 9) $P, \Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[\mathbf{new}\ c] : \mathsf{nat}$. Since $P, \Gamma' \vdash_{\underline{\mathsf{e}}} \mathbf{new}\ c : c$ and $P, \Gamma' \vdash_{\underline{\mathsf{e}}} \mathit{object} : c$, we get from Lemma 10 that $P, \Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[\mathit{object}] : \mathsf{nat}$.

2. Σ_5 is unaffected by the change to $\mathcal{S}$ and Γ.

Case $[\xrightarrow{get}]$: Let $\Gamma' = \Gamma$.

1. $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[\mathit{object}\underline{\ :c'\ }.\mathit{fd}] : \mathsf{nat}$. Let t be such that $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object}\underline{\ :c'\ }.\mathit{fd} : t$. If $v = \mathsf{null}$ then v can be typed as $t \implies$ (by Lemma 10) $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[v] : \mathsf{nat}$. Now assume v is some object'. We know by inversion of $[\mathsf{get}, \vdash_{\mathsf{e}}]$ that $\langle c'.\mathit{fd}, t \rangle\ \boxed{\textsc{FieldInClass}_P}\ c$, for some c. By Σ_6 and Σ_4, $c''\ \boxed{\textsc{SubType}_P}\ t$, where $\mathcal{S}(\mathit{object}') = \langle c'', \_ \rangle \implies$ (by Σ_1) $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object}' : c'' \implies$ (by Lemma 12) $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[v] : \mathsf{nat}$. Finally, if v is a number then Σ_5 gives us $t = \mathsf{nat}$. Since $P, \Gamma \vdash_{\underline{\mathsf{e}}} v : \mathsf{nat}$, by Lemma 10 we get $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[v] : \mathsf{nat}$.

2. $\mathcal{S}$ and Γ are unchanged.

Case $[\xrightarrow{set}]$: Let $\Gamma' = \Gamma$.

1. Very similar to previous case.

2. Σ_3, Σ_4, and Σ_5 are the properties affected by the field update, but the proof of Σ_4 can be used from [4].
   Σ_3: By definition of v and Lemma 13.
   Σ_5: By inversion of $[\mathsf{set}, \vdash_{\mathsf{e}}]$, $c_2 = \mathsf{nat}$.

Case $[\xrightarrow{call}]$: Let $\Gamma' = \Gamma$.

1. We assume $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathit{object}.\mathit{md}(v_1, ..., v_n)] : \mathsf{nat}$. Thus, we know $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object}.\mathit{md}(v_1, ..., v_n) : t$ for some t. By inversion of $\left[\mathsf{call}, \vdash_{\underline{\mathsf{e}}}\right]$, we get $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object} : t'$ and $\langle \mathit{md}, t_1 \; ... \; t_n \rightarrow \; t, \mathit{var}_1 \; ... \; \mathit{var}_n, e_b \rangle \; \boxed{\text{MethInClass}_P} \; t'$ and $P, \Gamma \vdash_{\underline{\mathsf{e}}} v_i : t_i \implies P, \Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i \implies P, t_0 \vdash_{\underline{\mathsf{m}}} t \; \mathit{md} \; (t_1 \; \mathit{var}_1, ..., t_n \; \mathit{var}_n) \; \{e_b\}$, where t_0 is the defining class of $\mathit{md} \implies$
   $P, [\mathbf{this} : t_0, \mathit{var}_1 : t_1, ..., \mathit{var}_n : t_n] \vdash_{\underline{\mathsf{s}}} e_b : t \implies$ (with an alpha-renaming if necessary)
   $P, \Gamma\,[\mathbf{this} : t_0, \mathit{var}_1 : t_1, ..., \mathit{var}_n : t_n] \; \vdash_{\underline{\mathsf{s}}} e_b : t \implies$ (by inversion of $\left[\mathsf{sub}, \vdash_{\underline{\mathsf{s}}}\right]$)
   $P, \Gamma\,[\mathbf{this} : t_0, \mathit{var}_1 : t_1, ..., \mathit{var}_n : t_n] \; \vdash_{\underline{\mathsf{e}}} e_b : t''$, for some $t'' \; \boxed{\text{SubType}_P} \; t$. From $\boxed{\text{MethInClass}_P}$ we know that $t' \; \boxed{\text{SubType}_P} \; t_0 \implies P, \Gamma \vdash_{\underline{\mathsf{s}}} \mathit{object} : t_0 \implies$ (by Lemma 11) $P, \Gamma \vdash_{\underline{\mathsf{s}}} e_b\,[\mathit{object}/\mathbf{this}, v_1/\mathit{var}_1, ..., v_n/\mathit{var}_n] : t'' \implies P, \Gamma \vdash_{\underline{\mathsf{s}}} e_b\,[...] : t \implies$ (by Lemma 12) $P, \Gamma \vdash_{\underline{\mathsf{s}}} \mathsf{E}\,[e_b\,[...]] : \mathsf{nat} \implies P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[e_b\,[...]] : \mathsf{nat}$.

2. $\mathcal{S}$ and Γ are unchanged.

Case $[\xrightarrow{super}]$: The proof is very similar to the proof for $\mathit{object}.\mathit{md}(v_1, ..., v_n)$.

Case $[\xrightarrow{cast}]$: Let $\Gamma' = \Gamma$.

1. We assume $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[\mathbf{view}\; t'\,\mathit{object}] : \mathsf{nat} \implies \mathbf{view}\; t'\,\mathit{object} : t'$. By the side-conditions of $[\mathsf{cast}]$, we know that $\mathcal{S}(\mathit{object}) = \langle c, \_ \rangle$ and $c \; \boxed{\text{SubType}_P} \; t' \implies P, \Gamma \vdash_{\underline{\mathsf{s}}} \mathit{object} : t' \implies$ (by Lemma 12) $P, \Gamma \vdash_{\underline{\mathsf{s}}} \mathsf{E}\,[\mathit{object}] : \mathsf{nat}$.

2. $\mathcal{S}$ and Γ are unchanged.

Case $[\xrightarrow{add1}]$: Let $\Gamma' = \Gamma$.

1. We assume $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[\mathbf{add1}\; v] : \mathsf{nat}$. By inversion of $\left[\mathsf{add1}, \vdash_{\underline{\mathsf{e}}}\right]$, we get that $P, \Gamma \vdash_{\underline{\mathsf{e}}} v : \mathsf{nat}$. By Lemma 10, since $v + 1$ is also of type nat, $P, \Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}\,[v + 1] : \mathsf{nat}$.

2. $\mathcal{S}$ and Γ are unchanged.

Case $[\xrightarrow{fl}]$: Let $\Gamma' = \Gamma \backslash \{\mathit{object} \mapsto c \mid \mathit{object} \in \mathit{dom}\,(S_2) \;\; \mathit{and} \;\; \Gamma(\mathit{object}) = c\}$.[6] $P \vdash \langle e, \mathcal{S}_1 \uplus \mathcal{S}_2 \rangle \longrightarrow \langle e, \mathcal{S}_1 \rangle$.

1. We assume $P, \Gamma \vdash_{\underline{\mathsf{e}}} e : \mathsf{nat}$. Let $G \equiv \Gamma \backslash \Gamma'$. Assume that rule $\left[\mathsf{obj}, \vdash_{\underline{\mathsf{e}}}\right]$ with some $\mathit{object} \in \mathit{dom}\,(G)$ is used in the typing derivation of e. Since, in a derivation, a typing environment is never extended or reduced with respect to Loc's, we know that $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object} : c$, for some $c \implies \mathit{object} \in \mathit{dom}\,(\Gamma)$. Thus, $P, \Gamma \vdash_{\sigma} (\mathcal{S}_1 \uplus \mathcal{S}_2), e \implies$ (by Σ_6) $\mathit{object} \in \mathit{dom}\,(\mathcal{S}_1 \uplus \mathcal{S}_2) \implies$ (by choice of object) $\mathit{object} \in \mathit{dom}\,(\mathcal{S}_2)$. Also, object in the typing derivation of $e \implies \mathit{object} \in \mathit{FL}\,(e) \implies$ (since $\mathit{object} \in \mathit{dom}\,(\mathcal{S}_2)$, and hence $\mathit{object} \notin \mathit{dom}\,(\mathcal{S}_1)$) $\mathit{FL}\,(e, \mathcal{S}_1) \neq \emptyset$, which is a contradiction by the side-condition of the $\xrightarrow{fl}$ reduction rule. Thus, $\mathit{object} \in \mathit{dom}\,(G)$ cannot exist in the typing derivation of e with $\Gamma \implies$ the same typing derivation can be used for Γ'.

2. For $\Sigma_1 ... \Sigma_5$, assume $\mathcal{S}_1(\mathit{object}) = \langle c, \mathcal{F} \rangle$.
   Σ_1: Satisfied since $\mathcal{S}_1 \subseteq \mathcal{S}_1 \uplus \mathcal{S}_2$ and $P, \Gamma \vdash_{\sigma} \mathcal{S}_1 \uplus \mathcal{S}_2$
   Σ_2: Same argument as Σ_1.

[6] Γ' is a subset of the bindings in Γ. We define Γ' as such so that environment-store consistency is preserved across an $\xrightarrow{fl}$ application.

Σ_3: We have only removed bindings with this step. So we only need to consider the possibility that we removed something in the $\mathit{rng}\,(\mathcal{F})$. Precisely, assume some $\mathit{object}_2 \in \mathit{dom}\,(\mathcal{S}_2)$ such that $\mathcal{F}(c_1.\mathit{fd}) = \mathit{object}_2$ for some c_1 and fd. We know $\mathit{object}_2 \in \mathit{FL}\,(\langle c, \mathcal{F}\rangle) \subseteq \mathit{FL}\,(\mathcal{S}_1) \implies \{\mathit{object}_2 \mapsto ...\} \in \mathit{FL}\,(e, \mathcal{S}_1)$ which violates the side-condition of $\xrightarrow{\mathit{fl}}$.

Σ_4: By a similar argument to the one we used for Σ_3, we get $\mathit{object}' \in \mathit{dom}\,(\mathcal{S}_1) \implies$ (since $P, \Gamma \vdash_\sigma \mathcal{S}_1 \uplus \mathcal{S}_2, e$) Σ_4 is satisfied.

Σ_5: Same argument as Σ_1.

Σ_6: Let $\mathit{object} \in \mathit{dom}\,(\Gamma') \implies$ (by $P, \Gamma \vdash_\sigma \mathcal{S}_1 \uplus \mathcal{S}_2, e$) $\Gamma(\mathit{object}) = c$, for some $c \implies$ (by definition of Γ') $\mathit{object} \notin \mathit{dom}\,(\mathcal{S}_2)$.

Σ_7: Assume $\mathcal{S}_1(\mathit{object}) = \langle c, \mathcal{F}\rangle \implies \Gamma(\mathit{object}) = c \implies$ (since $\mathit{object} \in \mathit{dom}\,(\mathcal{S}_1)$) $\mathit{object} \in \mathit{dom}\,(\Gamma')$.

■

Lemma 8 (Progress) *If $P, \Gamma \vdash_{\underline{\mathsf{e}}} e : \mathsf{nat}$ and $P, \Gamma \vdash_\sigma S$ and* $\textsc{SuperOK}(e)$*, then either e is* null *or n or there exists an $\langle e', \mathcal{S}'\rangle$ such that $P \vdash \langle e, \mathcal{S}\rangle \longrightarrow \langle e', \mathcal{S}'\rangle$, where $\longrightarrow$ must not be $\xrightarrow{\mathit{fl}}$.*[7]

Proof: If e is null or e is a number then we are done. If not, then we case on the redex in e.

Case [new c]: The proof follows directly from the [new] reduction rule.

Case [$v\underline{\ : c}.\mathit{fd}$]: By type-checking, $v \notin \mathit{Nat}$. If $v = \mathsf{null}$, $\xrightarrow{\mathit{nget}}$ applies. If $v = \mathit{object}$, then we can show that [get] applies: Type checking (specifically inversion of $\left[\mathsf{get}, \vdash_{\underline{\mathsf{e}}}\right]$) $\implies P, \Gamma \ \vdash_{\underline{\mathsf{e}}} \mathit{object} : t'$, for some type $t' \implies$ (by Lemma 13) $\mathcal{S}(\mathit{object}) = \langle t', \mathcal{F}\rangle$, for some $\mathcal{F}$. By inversion of $\left[\mathsf{get}, \vdash_{\underline{\mathsf{e}}}\right]$ we also know that $\langle c.\mathit{fd}, t\rangle \ \boxed{\textsc{FieldInClass}_P}\ t'$, for some $t \implies$ (by Σ_2) $c.\mathit{fd} \in \mathit{dom}\,(\mathcal{F})$.

Case [$v\underline{\ :\ \ c}.\mathit{fd} = v'$]: Similar to $v\underline{\ : c}.\mathit{fd}$.

Case [$v.\mathit{md}(v_1, ..., v_n)$]: By type-checking, $v \notin \mathit{Nat}$. If v is null, $\xrightarrow{\mathit{ncall}}$ applies. Assume $v = \mathit{object}$. Type-checking $\implies P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object} : t' \implies$ (by Σ_6) $\mathit{object} \in \mathit{dom}\,(\mathcal{S}) \implies \mathcal{S}(\mathit{object}) = \langle t', \mathcal{F}\rangle$, for some $\mathcal{F}$. Type checking $\implies \langle \mathit{md}, T, V, e_b\rangle \ \boxed{\textsc{MethInClass}_P}\ t'$.

Case [super$\underline{\ \equiv\ \ v\ \ :\ \ c}$]: By $\textsc{SuperOK}(e)$, v is some object. Type-checking $\implies \langle \mathit{md}, T, V, e_b\rangle \ \boxed{\textsc{MethInClass}_P}\ c$.

Case [view t v]: By type-checking, $v \notin \mathit{Nat}$. If $v = \mathsf{null}$, $\xrightarrow{\mathit{ncast}}$ applies. Assume $v = \mathit{object}$, and type-checking $\implies P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object} : t' \implies$ (by Σ_6) $\mathcal{S}(\mathit{object}) = \langle t', \mathcal{F}\rangle$ for some $\mathcal{F}$.

Case [add1 v]: Since type-checking guarantees $v \in \mathit{Nat}$, $\xrightarrow{\mathit{add1}}$ applies. ■

Lemma 9 ***(Free)*** *If $P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t$ and $a \notin \mathit{dom}\,(\Gamma)$, then $P, \Gamma\,[a : t'] \vdash_{\underline{\mathsf{e}}} e : t$.*

Proof: From Flatt, et. al. [4], Lemma C.3.1. ■

Lemma 10 ***(Replacement)*** *If $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t, P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t', P, \Gamma \vdash_{\underline{\mathsf{e}}} e' : t'$, then $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e'] : t$.*

[7]Note that e cannot be an error expression since such expressions are not typable.

Proof: From Flatt, et. al. [4], Lemma C.3.2. $\blacksquare$

Lemma 11 ***(Substitution)*** *If* $P, \Gamma\,[var_1 : t_1, ..., var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$ *and* $P, \Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ *for* $i \in [1, n]$*, then* $P, \Gamma \vdash_{\underline{\mathsf{s}}} e\,[v_1/var_1, ..., v_n/var_n] : t$*.*

Proof: From Flatt, et. al. [4], Lemma C.3.3. $\blacksquare$

Lemma 12 ***(Replacement with Subtyping)*** *If* $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t, P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t'$*, and* $P, \Gamma \vdash_{\underline{\mathsf{e}}} e' : t''$ *where* $t'' \boxed{\text{SubType}P} t'$*, then* $P, \Gamma \vdash_{\underline{\mathsf{s}}} \mathsf{E}[e'] : t$*.*

Proof: From Flatt, et. al. [4], Lemma C.3.4. $\blacksquare$

Lemma 13 ***(Consistency Consequence)*** *If* $P, \Gamma \vdash_\sigma \mathcal{S}$ *and* $P, \Gamma \vdash_{\underline{\mathsf{e}}} object : c$*, then* $\mathcal{S}(object) = \langle c, \mathcal{F} \rangle$*, for some* $\mathcal{F}$*.*

Proof: $P, \Gamma \vdash_{\underline{\mathsf{e}}} object : c \implies$ (by inversion of $[\mathsf{obj}, \vdash_{\underline{\mathsf{e}}}]$) $\Gamma(object) = c \implies$ (with $P, \Gamma \vdash_\sigma \mathcal{S}$) $object \in dom\,(\mathcal{S}) \implies \mathcal{S}(object) = \langle c', \mathcal{F} \rangle$, for some c' and $\mathcal{F} \implies$ (by Σ_1) $\Gamma(object) = d \implies c = d$. $\blacksquare$
.

predicate $\textsc{ClassesOnce}(P)$	Each class name is declared only once. **class** c ... **class** c' ... is in $P \implies c \neq c'$
predicate $\textsc{FieldOncePerClass}(P)$	Field names in each class declaration are unique. **class** ... $\{$... fd ... fd' ...$\}$ is in $P \implies fd \neq fd'$
predicate $\textsc{MethodOncePerClass}(P)$	Method names in each class declaration are unique. **class** ... $\{$... md ... md' ...$\}$ is in $P \implies md \neq md'$
predicate $\textsc{InterfacesOnce}(P)$	Each interface is declared only once. **interface** i ... **interface** i' ... is in $P \implies i \neq i'$
predicate $\textsc{InterfacesAbstract}(P)$	Method declarations in an interface are abstract. **interface** ... $\{$... $md(...)\,\{e\}$ ...$\}$ is in $P \implies e$ is **abstract**
predicate $\textsc{MethodArgsDistinct}(P)$	Each method argument name is unique. $md\ (t_1\ var_1\ ...\ t_n\ var_n)\ \{...\}$ is in $P \implies var_1, ..., var_n$ and **this** are all distinct
relation $\boxed{\textsc{SubClassDec}_P}$	Class is declared as an immediate subclass. $c\ \boxed{\textsc{SubClassDec}_P}\ c' \Leftrightarrow$ **class** c **extends** c' ... $\{...\}$ is in P
relation $\boxed{\textsc{ClassFieldDec}_P}$	Field is declared in class. $\langle c.fd, t\rangle\ \boxed{\textsc{ClassFieldDec}_P}\ c \Leftrightarrow$ **class** c... $\{$ ... $t\ fd$ ...$\}$ is in P
relation $\boxed{\textsc{ClassMethDec}_P}$	Method is declared in class. $\langle md, (t_1\ ...\ t_n \rightarrow t), (var_1\ ...\ var_n), e\rangle\ \boxed{\textsc{ClassMethDec}_P}\ c \Leftrightarrow$ **class** c ... $\{$... $t\ md(t_1\ var_1\ ...\ t_n\ var_n)\,\{e\}$ ...$\}$ is in P
relation $\boxed{\textsc{SubInterfaceDec}_P}$	Interface is declared as an immediate sub-interface. $i\ \boxed{\textsc{SubInterfaceDec}_P}\ i' \Leftrightarrow$ **interface** i **extends** ... i' ... $\{...\}$ is in P
relation $\boxed{\textsc{IntMethDec}_P}$	Method is declared in an interface. $\langle md, t_1, t_2, ..., t_n \rightarrow t, var_1, ..., var_n, e\rangle\ \boxed{\textsc{IntMethDec}_P}\ i$ $\Leftrightarrow$ **interface** i ... $\{$... $t\ md\ (t_1\ var_1\ ...\ t_n\ var_n)\ \{e\}$ ...$\}$ is in P
relation $\boxed{\textsc{ImplementsDec}_P}$	Class declares implementation of an interface. $c\ \boxed{\textsc{ImplementsDec}_P}\ i \Leftrightarrow$ **class** c ... **implements**... i ... $\{...\}$ is in P

Table A1: Predicates and relations of ClassicJava

relation	Class is a subclass.
$\boxed{\textsc{SubClass}_P}$	$\boxed{\textsc{SubClass}_P} \equiv$ the transitive, reflexive closure of $\boxed{\textsc{SubClassDec}_P}$
predicate	Classes that are extended are defined.
$\textsc{CompleteClases}(P)$	$\mathit{rng}\left(\boxed{\textsc{SubClassDec}_P}\right) \subseteq \mathit{dom}\left(\boxed{\textsc{SubClassDec}_P}\right) \cup \{\mathit{object}\}$
predicate	Class hierarchy is an order.
$\textsc{WellFoundedClasses}(P)$	$\boxed{\textsc{SubClass}_P}$ is antisymmetric
relation	Interface is a sub-interface.
$\boxed{\textsc{SubInterface}_P}$	$\boxed{\textsc{SubInterface}_P} \equiv$ the transitive, reflexive closure of $\boxed{\textsc{SubInterfaceDec}_P}$
predicate	Extended/implemented interfaces are defined.
$\textsc{CompleteInterfaces}(P)$	$\mathit{rng}\left(\boxed{\textsc{SubInterfaceDec}_P}\right) \cup \mathit{rng}\left(\boxed{\textsc{ImplementsDec}_P}\right)$ $\subseteq \left[\mathit{dom}\left(\boxed{\textsc{SubInterfaceDec}_P}\right) \cup \{\mathsf{Empty}\}\right]$
predicate	Interface hierarchy is an order.
$\textsc{WellFoundedInterfaces}(P)$	$\boxed{\textsc{SubInterface}_P}$ is antisymmetric
relation	Class implements an interface.
$\boxed{\textsc{Implements}_P}$	$c \boxed{\textsc{Implements}_P} i \Leftrightarrow$ $\exists c', i'$ s.t. $c \boxed{\textsc{SubClass}_P} c'$ and $i' \boxed{\textsc{SubInterface}_P} i$ and $c' \boxed{\textsc{ImplementsDec}_P} i'$

Table A2: Predicates and relations of ClassicJava

predicate	Method overriding preserves the type.
$\textsc{ClassMethodsOK}(P)$	$\langle md, T, V, e\rangle \boxed{\textsc{ClassMethDec}_P} c$ and $\langle md, T', V', e'\rangle \boxed{\textsc{ClassMethDec}_P} c'$ $\implies \left(T = T' \text{ or } c \boxed{\textbf{NOT } \textsc{SubClass}_P} c'\right)$
relation	Field is contained in a class.
$\boxed{\textsc{FieldInClass}_P}$	$\langle c'.fd, t\rangle \boxed{\textsc{FieldInClass}_P} c \Leftrightarrow$ $\left[\begin{array}{l}\langle c'.fd, t\rangle \boxed{\textsc{ClassFieldDec}_P} c' \text{ and} \\ c' = \min\left\{c'' \mid c \boxed{\textsc{SubClass}_P} c'' \text{ and } \exists t' \text{ s.t. } \langle c''.fd, t'\rangle \boxed{\textsc{ClassFieldDec}_P} c''\right\}\end{array}\right]$
relation	Method is contained in a class.
$\boxed{\textsc{MethInClass}_P}$	$\langle md, T, V, e\rangle \boxed{\textsc{MethInClass}_P} c \Leftrightarrow$ $\langle md, T, V, e\rangle \boxed{\textsc{ClassMethDec}_P} c'$ and $c' =$ $\min\left\{c'' \mid c \boxed{\textsc{SubClass}_P} c'' \text{ and } \exists e', V' \text{ s.t. } \langle md, T, V', e'\rangle \boxed{\textsc{ClassMethDec}_P} c''\right\}$
predicate	Interface inheritance or re-declaration of methods is consistent.
$\textsc{IntMethodsOK}(P)$	$[\langle md, T, V, \textbf{abstract}\rangle \boxed{\textsc{IntMethDec}_P} i$ and $\langle md, T', V', \textbf{abstract}\rangle \boxed{\textsc{IntMethDec}_P} i']$ $\implies (T = T' \text{ or } \forall i''(i'' \boxed{\textbf{NOT } \textsc{SubInterface}_P} i \text{ or } i'' \boxed{\textbf{NOT } \textsc{SubInterface}_P} i'))$
relation	Method is contained in an interface.
$\boxed{\textsc{MethInInt}_P}$	$\langle md, T, V, \textbf{abstract}\rangle \boxed{\textsc{MethInInt}_P} i \Leftrightarrow$ $i \boxed{\textsc{SubInterface}_P} i'$ and $\langle md, T, V, \textbf{abstract}\rangle \boxed{\textsc{IntMethDec}_P} i'$

Table A3: Predicates and relations of ClassicJava

predicate	Classes supply methods to implement interfaces.
$\textsc{ClassesImplementAll}(P)$	$c \ \boxed{\textsc{ImplementsDec}_P} \ i \implies \forall md, T, V, \left(\begin{array}{l} \langle md, T, V, \textbf{abstract} \rangle \ \boxed{\textsc{MethInInt}_P} \ i \implies \\ \exists e, V' \text{ s.t. } \langle md, T, V', e \rangle \ \boxed{\textsc{MethInClass}_P} \ c \end{array} \right)$
predicate	Class has no **abstract** methods.
$\textsc{NoAbstractMethods}(P,c)$	$\langle md, T, V, e \rangle \ \boxed{\textsc{MethInClass}_P} \ c \implies e \neq \textbf{abstract}$
relation	Type is a subtype.
$\boxed{\textsc{SubType}_P}$	$\boxed{\textsc{SubType}_P} \equiv \boxed{\textsc{SubClass}_P} \cup \boxed{\textsc{SubInterface}_P} \cup \boxed{\textsc{Implements}_P}$
relation	Field or method is in a type.
$\boxed{\textsc{InType}_P}$	$\boxed{\textsc{InType}_P} \equiv \boxed{\textsc{MethInClass}_P} \cup \boxed{\textsc{FieldInClass}_P} \cup \boxed{\textsc{MethInInt}_P}$

Table A4: Predicates and relations of ClassicJava

$\textsc{ClassesOnce}(P)$ $\quad$ $\textsc{InterfacesOnce}(P)$ $\quad$ $\textsc{MethodOncePerClass}(P)$

$\textsc{FieldOncePerClass}(P)$ $\quad$ $\textsc{CompleteClasses}(P)$ $\quad$ $\textsc{WellFoundedClasses}(P)$

$\textsc{CompleteInterfaces}(P)$ $\quad$ $\textsc{WellFoundedInterfaces}(P)$ $\quad$ $\textsc{IntMethodsOK}(P)$

$\textsc{InterfacesAbstract}(P)$ $\quad$ $\textsc{MethodArgsDistinct}(P)$ $\quad$ $\textsc{ClassesImplementAll}(P)$

$$\frac{P \vdash_{\mathsf{d}} \mathit{defn}_j \Longrightarrow \mathit{defn}'_j,\ \text{for } j \in [1,n] \qquad P, [] \vdash_{\mathsf{e}} e \Longrightarrow e' : \mathsf{nat} \qquad \text{where } P = \mathit{defn}_1 \ \ldots\ \mathit{defn}_n\ e}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ \ldots\ \mathit{defn}_n\ e \Longrightarrow \mathit{defn}'_1 \ \ldots\ \mathit{defn}'_n\ e' : \mathsf{nat}} \quad [\mathsf{prog}, \vdash_{\mathsf{p}}]$$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} \mathit{meth}_k \Longrightarrow \mathit{meth}'_k \text{ for each } k \in [1,p]}{\begin{array}{c} P \vdash_{\mathsf{d}} \textbf{class}\ c\ \ldots\ \{t_1\ fd_1\ \ldots\ t_n\ fd_n\ \mathit{meth}_1\ \ldots\ \mathit{meth}_p\} \Longrightarrow \\ \textbf{class}\ c\ \ldots\ \{t_1\ fd_1\ \ldots\ t_n\ fd_n\ \mathit{meth}'_1\ \ldots\ \mathit{meth}'_p\} \end{array}} \quad [\mathsf{defn\text{-}class}, \vdash_{\mathsf{d}}]$$

$$\frac{P, i \vdash_{\mathsf{m}} \mathit{meth}_j \Longrightarrow \mathit{meth}_j \text{ for each } j \in [1,p]}{P \vdash_{\mathsf{d}} \textbf{interface}\ i\ \ldots\{\mathit{meth}_1\ \ldots\ \mathit{meth}_p\} \Longrightarrow \textbf{interface}\ i\ \ldots\{\mathit{meth}_1\ \ldots\ \mathit{meth}_p\}} \quad [\mathsf{defn\text{-}int}, \vdash_{\mathsf{d}}]$$

$$\frac{P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P, [\textbf{this} : t_0, \mathit{var}_1 : t_1, \ldots, \mathit{var}_n : t_n] \vdash_{\mathsf{s}} e \Longrightarrow e' : t}{P, t_0 \vdash_{\mathsf{m}} t\ md(t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\{e\} \Longrightarrow t\ md(t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\{e'\}} \quad [\mathsf{meth}, \vdash_{\mathsf{m}}]$$

$$\frac{P \vdash_{\mathsf{t}} c \qquad \textsc{NoAbstractMethods(P,c)}}{P, \Gamma, \vdash_{\mathsf{e}} \textbf{new}\ c \Longrightarrow \textbf{new}\ c : c} \quad [\mathsf{new}, \vdash_{\mathsf{e}}]$$

$$\frac{\mathit{var} \in \mathit{dom}\,(\Gamma)}{P, \Gamma \vdash_{\mathsf{e}} \mathit{var} \Longrightarrow \mathit{var} : \Gamma(\mathit{var})} \quad [\mathsf{var}, \vdash_{\mathsf{e}}]$$

$$\frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{null} \Longrightarrow \mathsf{null} : t} \quad [\mathsf{null}, \vdash_{\mathsf{e}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Longrightarrow e' : t' \qquad \langle c.fd, t\rangle\ \boxed{\textsc{FieldInClass}_P}\ t'}{P, \Gamma \vdash_{\mathsf{e}} e.fd \Longrightarrow e' \underline{: c}.fd : t} \quad [\mathsf{get}, \vdash_{\mathsf{e}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Longrightarrow e' : t' \qquad \langle c.fd, t\rangle\ \boxed{\textsc{FieldInClass}_P}\ t' \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Longrightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.fd = e_v \Longrightarrow e' \underline{: c}.fd = e'_v : t} \quad [\mathsf{set}, \vdash_{\mathsf{e}}]$$

Table A5: ClassicJava typing rules

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Longrightarrow e' : t' \quad \langle md, T, V, e_b\rangle \ \boxed{\textsc{MethInClass}_P}\ t' \quad P,\Gamma \vdash_{\mathsf{s}} e_j \Longrightarrow e'_j : t_j \text{ for } j \in [1,n]}{P,\Gamma \vdash_{\mathsf{e}} e.md(e_1 \ldots e_n) \Longrightarrow e'.md(e'_1 \ldots e'_n) : t} \quad [\mathsf{call}, \vdash_{\mathsf{e}}]$$

$$\frac{\begin{array}{l} P,\Gamma \vdash_{\mathsf{e}} \mathbf{this} \Longrightarrow \mathbf{this} : c' \quad c' \ \boxed{\textsc{SubClassDec}_P}\ c \\ \langle md, T, V, e_b\rangle \ \boxed{\textsc{MethInClass}_P}\ c \quad e_b \neq \mathbf{abstract} \quad P,\Gamma \vdash_{\mathsf{s}} e_j \Longrightarrow e'_j : t_j \text{ for } j \in [1,n] \end{array}}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{super}.md(e_1 \ldots e_n) \Longrightarrow \mathbf{super} \underline{\equiv \mathbf{this} : c}.md(e'_1 \ldots e'_n) : t} \quad [\mathsf{super}, \vdash_{\mathsf{e}}]$$

$$\frac{P,\Gamma \vdash_{\mathsf{s}} e \Longrightarrow e' : t \quad t \neq \mathsf{nat}}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Longrightarrow e' : t} \quad [\mathsf{wcast}, \vdash_{\mathsf{e}}]$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Longrightarrow e' : t' \quad \left(\begin{array}{l} t \ \boxed{\textsc{SubType}_P}\ t' \quad \text{or} \\ t \in dom\left(\boxed{\textsc{SubInterfaceDec}_P}\right) \text{ or} \\ t' \in dom\left(\boxed{\textsc{SubInterfaceDec}_P}\right) \end{array}\right) \quad t' \neq \mathsf{nat}}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Longrightarrow \mathbf{view}\ t\ e' : t} \quad [\mathsf{ncast}, \vdash_{\mathsf{e}}]$$

$$\frac{P \vdash_{\mathsf{t}} t}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{abstract} \Longrightarrow \mathbf{abstract} : t} \quad [\mathsf{abs}, \vdash_{\mathsf{e}}]$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Longrightarrow e' : t' \quad t' \ \boxed{\textsc{SubType}_P}\ t}{P,\Gamma \vdash_{\mathsf{s}} e \Longrightarrow e' : t} \quad [\mathsf{sub}, \vdash_{\mathsf{s}}]$$

$$\frac{t \in \left[dom\left(\boxed{\textsc{SubClassDec}_P}\right) \cup dom\left(\boxed{\textsc{SubInterfaceDec}_P}\right) \cup \{object, \mathsf{Empty}\}\right]}{P \vdash_{\mathsf{t}} t} \quad [\mathsf{type}, \vdash_{\mathsf{t}}]$$

Table A6: ClassicJava typing rules

$$\begin{aligned}
\mathsf{E} = \quad & [\,] \\
& \mid \mathsf{E}\underline{\ :\ c\ }.fd \\
& \mid \mathsf{E}\underline{\ :\ c\ }.fd = e \mid v\underline{\ :\ c\ }.fd = \mathsf{E} \\
& \mid \mathsf{E}.md(e\ ...) \mid v.md(v\ ...\ \mathsf{E}\ e...) \qquad \text{[evaluation contexts]} \\
& \mid \mathbf{super}\underline{\equiv\ v\ :\ c}.md(v...\mathsf{E}\ e...) \\
& \mid \mathbf{view}\ t\ \mathsf{E} \\
& \mid \mathbf{add1}\ \mathsf{E}
\end{aligned}$$

$P \vdash \langle \mathsf{E}\,[\mathbf{new}\ c]\,, \mathcal{S}\rangle \longrightarrow \langle \mathsf{E}\,[\mathit{object}]\,, \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle$ $[\xrightarrow{new}]$
where $\mathit{object}\ \notin \mathit{dom}\,(\mathcal{S})$ and
$\mathcal{F} = \left\{c'.fd \mapsto \mathsf{null} \mid c\ \boxed{\textsc{SubClass}_P}\ c' \text{ and} \exists t \text{ s.t. } \langle c'.fd, t\rangle\ \boxed{\textsc{ClassFieldDec}_P}\ c'\right\}$

$P \vdash \langle \mathsf{E}\,[\mathit{object}\underline{\ :c'}.fd]\,, \mathcal{S}\rangle \longrightarrow \langle \mathsf{E}\,[v]\,, \mathcal{S}\rangle$ $[\xrightarrow{get}]$
where $S(\mathit{object}) = \langle c, F\rangle$ and $\mathcal{F}(c'.fd) = v$

$P \vdash \langle \mathsf{E}\,[\mathit{object}\underline{\ :c'}.fd = v]\,, \mathcal{S}\rangle \longrightarrow \langle \mathsf{E}\,[v]\,, \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.fd \mapsto v]\rangle]\rangle$ $[\xrightarrow{set}]$
where $S(\mathit{object}) = \langle c, \mathcal{F}\rangle$

$P \vdash \langle \mathsf{E}\,[\mathit{object}.md(v_1, ..., v_n)]\,, \mathcal{S}\rangle \longrightarrow \langle \mathsf{E}\,[e_b\,[\mathbf{this} \leftarrow \mathit{object}, \mathit{var}_1 \leftarrow v_1, ..., \mathit{var}_n \leftarrow v_n]\,, \mathcal{S}]\rangle$ $[\xrightarrow{call}]$
where $S(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $\left[\langle md, T, \mathit{var}_1\ ...\ \mathit{var}_n, e_b\rangle\ \boxed{\textsc{MethInClass}_P}\ c\right]$

$P \vdash \left\langle \mathsf{E}\left[\mathbf{super}\underline{\equiv \mathit{object} : c'}.md(v_1, ..., v_n)\right]\,, \mathcal{S}\right\rangle \longrightarrow$ $[\xrightarrow{super}]$
$\langle \mathsf{E}\,[e_b[\mathbf{this} \leftarrow \mathit{object}, \mathit{var}_1 \leftarrow v_1, ..., \mathit{var}_n \leftarrow v_n]]\,, \mathcal{S}\rangle$
where $\langle md, T, \mathit{var}_1\ ...\ \mathit{var}_n, e_b\rangle\ \boxed{\textsc{MethInClass}_P}\ c'$

$P \vdash \langle \mathsf{E}\,[\mathbf{view}\ t'\mathit{object}]\,, \mathcal{S}\rangle \longrightarrow \langle \mathsf{E}\,[\mathit{object}]\,, \mathcal{S}\rangle$ $[\xrightarrow{cast}]$
where $S(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $c\ \boxed{\textsc{SubType}_P}\ t'$

$P \vdash \langle \mathsf{E}\,[\mathbf{view}\ t\ e]\,, \mathcal{S}\rangle \longrightarrow \langle \mathbf{error:\ bad\ cast}, \mathcal{S}\rangle$ $[\xrightarrow{xcast}]$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ *and* c $\boxed{\textbf{NOT}\ \textsc{SubClass}_P}$ t

$P \vdash \langle \mathsf{E}\,[\mathbf{view}\ t\ \mathsf{null}]\,, \mathcal{S}\rangle \longrightarrow \langle \mathbf{error:\ bad\ cast}, \mathcal{S}\rangle$ $[\xrightarrow{ncast}]$

$P \vdash \langle \mathsf{E}\,[\mathsf{null}\underline{\ :\ c\ }.fd]\,, \mathcal{S}\rangle \longrightarrow \langle \mathbf{error:\ deref\ null}, \mathcal{S}\rangle$ $[\xrightarrow{nget}]$

$P \vdash \langle \mathsf{E}\,[\mathsf{null}\underline{\ :\ c\ }.fd = v]\,, \mathcal{S}\rangle \longrightarrow \langle \mathbf{error:\ deref\ null}, \mathcal{S}\rangle$ $[\xrightarrow{nset}]$

$P \vdash \langle \mathsf{E}\,[\mathsf{null}.md(v_1, ..., v_n)]\,, \mathcal{S}\rangle \longrightarrow \langle \mathbf{error:\ deref\ null}, \mathcal{S}\rangle$ $[\xrightarrow{ncall}]$

Table A7: ClassicJava operational semantics