Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1198-1214
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 © J.UCS

A Formal Semantics for Finalizers

Marcus Amorim Leal
(PUC-Rio, Brazil
mleal@inf.puc-rio.br)

Roberto Ierusalimschy
(PUC-Rio, Brazil
roberto@inf.puc-rio.br)

Abstract: Automatic finalization is a common but inherently complex language facility that makes the garbage collection process semantically visible to client programs. With finalizers, memory management becomes more flexible, and garbage collectors can be used to recycle other resources in addition to memory.

Formal language models usually ignore garbage collection, and therefore are unable to properly describe finalization. In this paper we use an operational approach to develop a new abstract model that explicitly represents memory management actions in a garbage-collected programming language based on the λ-calculus. We formally state and prove several important properties related to memory management, and employ the model to describe and explore a semantics for finalizers.

Key Words: memory management, garbage collection, finalization, semantics
Category: F.3.2, D.3.3, D.4.2

1 Introduction

Programming languages with automatic memory management usually support facilities that allow *client programs* to interact with the garbage collector (examples of such facilities include *finalizers*, *weak references* and explicit garbage collector invocation). Naturally, the corresponding semantics may hinge on how the garbage collector executes. But most language specifications are vague about garbage collection, imposing few, if any, constraints on actual implementations. This creates a semantic gap where the precise meaning of certain operations becomes ambiguous or simply undefined.

Only a few formal programming-language models explicitily address garbage collection (e.g., [Hudak, 1986, Demers et al., 1990, Mason and Talcott, 1990, Chirimar et al., 1992, Morrisett et al., 1995, Morrisett and Harper, 1998, Elsman, 2003, Hunter and Krishnamurthi, 2003]). Among those, the one by Morriset, Felleisen and Harper [Morrisett et al., 1995] is probably the best known. It uses a syntactic heap representation and specifies memory actions as a set of rewriting rules expressed with a small-step contextual evaluation semantics. Ordinary references however are not defined, and liveness is formalized by considering only free variables. Within this framework, the authors are able to give a compact description of different trace-based garbage collectors, including mark-and-copy and generational garbage collectors.

In this paper we describe an alternative model that is expressive enough to represent and clearly specify many relevant aspects of memory management, as well as memory management related facilities. On the other hand, the model is sufficiently abstract to let us state and easily prove several important language invariants. All results presented are independent from any singular choice of trace-based garbage collector [1].

Our model is somewhat similar to Morriset's, but we define an abstraction to explicitly represent references (and memory addresses), and describe the underlying language with a finer-grained operational semantics. This allow us to specify some aspects of garbage-collection related semantics that have been disregarded in other models.

Furthermore, as an interesting application, we develop a formal semantics for finalizers, and explore general issues that were only informally discussed by other authors [Schwartz and Melliar-Smith, 1981, Atkins and Nackman, 1988, Hayes, 1992, Dybvig et al., 1993, Boehm, 2003]. In particular, we are able to clearly show how finalizers affect garbage collection and the underlying programming-language semantics.

This paper is organized as follows. In Section 2 we describe a small untyped functional language with side effects that includes explicit memory allocation and references. In Section 3 we extend this language with garbage collection. In Section 4 we discuss finalizers and specify their formal semantics using the model developed in the previous sections. Finally, in the last section we present a summary and discuss future work.

2 The λ_{ref} Language

In this section we formalize the semantics of a small untyped language, which we will call λ_{ref}, using structural operational semantics [Plotkin, 1981]. The syntax of λ_{ref} is similar to a conventional higher-order language based on the λ-calculus, but extended with references and conditional expressions. A reference is represented by a *location* ($l_i \in \mathbf{Loc}$), which can be understood as an address to an allocated memory-cell.

Values ($v \in \mathbf{V}$) are represented by abstractions (functions), locations and the *nil* atom:

$$v ::= \ \lambda x_i.e \mid l_i \mid nil$$

[1] Wilson [Wilson, 1992] and Jones and Lins [Jones and Lins, 1996] provide excellent surveys on garbage collection techniques and algorithms.

Expressions ($e \in \mathbf{Exp}$), as indicated below, are represented by values, variables ($x_i \in \mathbf{Id}$), function applications, conditionals, and the operations on references: *allocation*, which uses the *new* operator, creates a reference to a memory cell; *dereferencing*, which uses the ! operator, retrieves the value stored by a referent memory cell; and *assignment*, which uses the := operator, stores a value in a referent memory cell.

$$e ::= \; v \mid x_i \mid e_1 e_2 \mid e_1 ?\, e_2 : e_3 \mid \mathit{new} \mid !e \mid e_1 := e_2$$

Values are bound to locations by *environments* (H), which are represented as finite maps from **Loc** to **V**. $H_\emptyset$ represents the empty environment, and so $H_\emptyset(l_i)$ is undefined for all l_i (undefined evaluations are represented with the $\perp$ symbol). To denote changes in an environment H we use the notation

$$H[l_i \mapsto v](l_j) = \begin{cases} v & \text{if } l_j = l_i \\ H(l_j) & \text{otherwise} \end{cases}$$

and we write $H[l_i \mapsto \perp]$ to denote a new environment derived from H by removing l_i from its domain.

The basic semantics of λ_{ref} is defined by the set of transition rules described in Figure 1 (this set of rules will be referred henceforth as R_{ref}). Notice that transitions are represented by "$\rightarrow$" (which can be read as *reduces in one step to*) and take place between *programs*. A program ($\mathcal{P}$) is defined as an expression and its associated environment ($\mathcal{P} = \langle e, H \rangle$). Some points deserve further comments:

- Allocations, described by Rule **alloc**, extend H with freshly created locations, which are initially bound to *nil*. This is the only way to introduce a new location in an environment's domain.
- **deref** states that dereferencing a reference evaluates to the value mapped by H.
- Assignments (**assign**) can be performed only on locations that belong to H's domain. An assignment expression evaluates to the value on the right-hand side of the assignment operator.
- Applications, described by Rule **applic**, are equivalent to the traditional λ-calculus β-reduction: all bounded variables in the function expression are replaced by the corresponding argument. This substitution, usually called a *context substitution*, is denoted by $\{x_i/v\}$.
- Rules **cond1** and **cond2** define conditional expressions. Conditions with non-*nil* values cause the expression to evaluate to its second argument, while a *nil* condition causes it to evaluate to its third argument.

$$\langle new, H\rangle \rightarrow \langle l_i, H[l_i \mapsto nil]\rangle \quad \text{where } l_i \notin dom(H) \qquad \textbf{(alloc)}$$

$$\langle !l_i, H\rangle \rightarrow \langle H(l_i), H\rangle \qquad \textbf{(deref)}$$

$$\langle l_i := v, H\rangle \rightarrow \langle v, H[l_i \mapsto v]\rangle \quad \text{if } l_i \in dom(H) \qquad \textbf{(assign)}$$

$$\langle (\lambda x_i.e)v, H\rangle \rightarrow \langle \{x_i/v\}e, H\rangle \qquad \textbf{(applic)}$$

$$\langle v \ ?\ e_1 : e_2, H\rangle \rightarrow \langle e_1, H\rangle \quad \text{if } v \neq nil \qquad \textbf{(cond1)}$$

$$\langle nil \ ?\ e_1 : e_2, H\rangle \rightarrow \langle e_2, H\rangle \qquad \textbf{(cond2)}$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle !e_1, H_1\rangle \rightarrow \langle !e_2, H_2\rangle} \qquad \textbf{(cont1)}$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 := e_3, H_1\rangle \rightarrow \langle e_2 := e_3, H_2\rangle} \qquad \textbf{(cont2)}$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle v := e_1, H_1\rangle \rightarrow \langle v := e_2, H_2\rangle} \qquad \textbf{(cont3)}$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 e_3, H_1\rangle \rightarrow \langle e_2 e_3, H_2\rangle} \qquad \textbf{(cont4)}$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle v e_1, H_1\rangle \rightarrow \langle v e_2, H_2\rangle} \qquad \textbf{(cont5)}$$

$$\frac{\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle}{\langle e_1 \ ?\ e_3 : e_4, H_1\rangle \rightarrow \langle e_2 \ ?\ e_3 : e_4, H_2\rangle} \qquad \textbf{(cont6)}$$

Figure 1: λ_{ref}'s transition rules.

– Rules **cont1** to **cont6** are the basic context evaluation rules, and define a left-to-right applicative-order semantics.

To improve code readability, the syntax of λ_{ref} is extended with two derived forms: *sequencing* and *let-binding*. The sequencing notation, represented as $e_1; e_2$, is a syntactic sugar for $(\lambda x_i.e_2)e_1$, where x_i does not occur in e_2. It has the simple effect of evaluating e_1, throwing away the corresponding result, and then evaluating e_2[2]. A let-binding, represented as $let\ x_i = e_1\ in\ e_2$, is a syntactic sugar for $(\lambda x_i.e_2)e_1$, where x_i usually does occur in e_2. It has the effect of evaluating e_1, replacing every occurrence of x_i in e_2 by e_1's result, and then evaluating e_2.

Before we extend the semantics of λ_{ref} we need to introduce a few more definitions and some related results. $LO(e)$ represents the set of all locations that occur literally in an expression e. The set of occurring locations in an environment H is defined as

$$LO(H) = \bigcup_{e_i \in range(H)} LO(e_i)$$

and the set of occurring locations in a program $\langle e, H \rangle$ is simply $LO(e) \cup LO(H)$.

We say that a program is *closed* if all its occurring locations are defined in its environment, or more formally, $\langle e, H \rangle$ is closed if $LO(e) \cup LO(H) \subseteq dom(H)$. Informally we can think of a closed program as one that has no dangling pointers. An important property of closed programs is stated by the following lemma.

Lemma 1. *Let* $\xrightarrow{R_{ref}}$ *denote any transition in* R_{ref}*. For any two programs* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *such that* $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_2$*, if* $\mathcal{P}_1$ *is closed then* $\mathcal{P}_2$ *is also closed.*

Proof. By cases on the elements of R_{ref}.

Consider $\mathcal{P}_1 = \langle e_1, H_1 \rangle$ and $\mathcal{P}_2 = \langle e_2, H_2 \rangle$. Locations that do not occur in e_1 can appear in e_2 only through **alloc** and **deref** transitions. In the former case, the new location is immediately added to H_2's domain. In the latter, if $\mathcal{P}_1$ is closed then any location referenced by a location that occurs in e_1 belongs to H_1's domain. Since for all transitions in R_{ref} we have that $dom(H_1) \subseteq dom(H_2)$, the referent location also belongs to H_2's domain.

A location that does not occur in H_1 can appear in H_2 only through **assign** transitions. But in this case the assigned location must occur in e_1. Since $\mathcal{P}_1$ is closed, the location belongs to $dom(H_1)$, and therefore, also to $dom(H_2)$. $\square$

A *location substitution* of l_i by l_j in an expression e, written as $\{l_i/l_j\}e$, is defined as a literal substitution of all occurrences of l_i by l_j in e. To avoid name

[2] This, of course, is only useful in languages with side effects, as is the case here.

collisions, this operation is undefined if $l_j \in LO(e)$. Location substitution for environments is defined in a similar way[3]:

$$\{l_i/l_j\}H = \begin{cases} H_\emptyset[l_k \mapsto \{l_i/l_j\}H(l_k)]^{l_k \in dom(H)} & \text{if } l_i \notin dom(H) \\ H_\emptyset[l_j \mapsto \{l_i/l_j\}H(l_i)][l_k \mapsto \{l_i/l_j\}H(l_k)]^{l_k \in dom(H)\setminus\{l_i\}} & \text{otherwise} \end{cases}$$

where the symbol $\setminus$ represents the difference between sets. If $l_j \in LO(H) \cup dom(H)$, $\{l_i/l_j\}H$ is undefined. Finally, location substitution for programs is defined as

$$\{l_i/l_j\}\langle e, H\rangle = \langle\{l_i/l_j\}e, \{l_i/l_j\}H\rangle$$

It is not hard to see that location substitution is preserved by R_{ref}. We state this formally with the following lemma.

Lemma 2. *Let* $\xrightarrow{R_{ref}}$ *denote any transition in* R_{ref}. *For any* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *such that* $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_2$, *and for any context substitution* $\{l_i/l_j\}$ *defined in* $\mathcal{P}_1$, *we have that* $\{l_i/l_j\}\mathcal{P}_1 \xrightarrow{R_{ref}} \{l_i/l_j\}\mathcal{P}'_2$, *where* $\mathcal{P}'_2 = \mathcal{P}_2$ *except perhaps for* α*-substitutions or some context substitution* $\{l_k/l_l\}$.

Proof. By cases on the elements of R_{ref}.

It is trivial to show that $\mathcal{P}'_2 = \mathcal{P}_2$ for any R_{ref} transitions, except **applic** and **alloc**, which respectively introduces an indeterminism in variables and location names.

applic transitions have the form $\langle(\lambda x_i.e_1)v, H\rangle \to \langle\{x_i/v\}e_2, H\rangle$, where $e_1 = e_2$ except eventually for α-substitutions. Applying the context substitution $\{l_i/l_j\}$ to the program $\langle(\lambda x_i.e_1)v, H\rangle$ we have:

$\{l_i/l_j\}\langle(\lambda x_i.e_1)v, H\rangle = \langle(\{l_i/l_j\}\lambda x_i.e_1)\{l_i/l_j\}v, \{l_i/l_j\}H\rangle \to$
$\langle\{x_i/(\{l_i/l_j\}v)\}(\{l_i/l_j\}e_2), \{l_i/l_j\}H\rangle = \{l_i/l_j\}\langle\{x_i/v\}e_2, H\rangle$

alloc transitions have the form $\langle new, H\rangle \to \langle l_k, H[l_k \mapsto nil]\rangle$. Applying the context substitution $\{l_i/l_j\}$ to the program $\langle new, H\rangle$ we have:

$\{l_i/l_j\}\langle new, H\rangle = \langle new, \{l_i/l_j\}H\rangle \to \langle l_l, \{l_i/l_j\}H[l_l \mapsto nil]\rangle$

If $l_l \neq l_k$ then

$\langle l_l, \{l_i/l_j\}H[l_l \mapsto nil]\rangle = \{l_i/l_j\}(\{l_k/l_l\}\langle l_k, H[l_k \mapsto nil]\rangle)$

else, if $l_l = l_k$ then

$\langle l_l, \{l_i/l_j\}H[l_l \mapsto nil]\rangle = \{l_i/l_j\}\langle l_k, H[l_k \mapsto nil]\rangle)$

□

[3] The notation $f(x_i)^{x_i \in \mathbf{X}}$ denotes the expression $f(x_1)f(x_2)...f(x_n)$, for all $x_i \in \mathbf{X}$. The terms corresponding to each x_i can appear in this expression in any order.

To denote that a program $\mathcal{P}_1$ reduces to $\mathcal{P}_2$ following a finite sequence of one or more transition rules from a set R, we use the notation $\mathcal{P}_1 \overset{R}{\Longrightarrow} \mathcal{P}_2$. Likewise, to denote that $\mathcal{P}_2$ is irreducible with respect to R and $\mathcal{P}_1 \overset{R}{\Longrightarrow} \mathcal{P}_2$ we write $\mathcal{P}_1 \Downarrow_R \mathcal{P}_2$.

If $\mathcal{P} \Downarrow_R \langle v, H\rangle$ for some $v \in \mathbf{V}$, we say that v is the result of $\mathcal{P}$, written as $\mathcal{P} \overset{R}{=} v$. A program $\mathcal{P}$ is *decidable* if there is some v such that $\mathcal{P} \overset{R}{=} v$. Otherwise $\mathcal{P}$ is *undecidable*.

In order to simplify our definition of program equivalence, abstracting some of the issues related to memory management that are not relevant to the present work, we introduce the concept of *structural congruence*. Two programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are structurally congruent, written $\mathcal{P}_1 \equiv \mathcal{P}_2$, if there is any finite sequence of α-substitutions and location context substitutions that transforms $\mathcal{P}_1$ into $\mathcal{P}_2$.

Structural congruence between expressions is defined in an analogous way. Two expressions are structurally congruent if they are identical or if one can be transformed into the other by a sequence of α-substitutions and location context substitutions.

Two sets of rules R_1 and R_2 are *equivalent*, represented as $R_1 \simeq R_2$, if for every result of any decidable program under either set of rules, there is a corresponding structurally congruent result for the same program under the other set of rules, and vice-versa.

A rule r is *deterministic* if for any two transitions under r, $\langle e_1, H_1\rangle \rightarrow \langle e_2, H_2\rangle$ and $\langle e_1, H_1\rangle \rightarrow \langle e_3, H_3\rangle$, it is always true that $e_2 \equiv e_3$. Otherwise r is non-deterministic.

Likewise, a set of rules R is deterministic if for any program $\mathcal{P}$ such that $\mathcal{P} \Downarrow_R \langle v, H_1\rangle$ and $\mathcal{P} \Downarrow_R \langle e, H_2\rangle$, it is always true that $e \equiv v$. Otherwise R is non-deterministic.

Notice that a set of rules can be non-deterministic even if all its rules are deterministic. On the other hand, a single non-deterministic rule implies that the correponding set is non-deterministic.

With the above definitions we can postulate a few trivial but important properties of λ_{ref} programs.

Lemma 3. *Let $\xrightarrow{R_{ref}}$ denote any transition in R_{ref}. For any $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1 \equiv \mathcal{P}_2$, if $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_3$, then $\mathcal{P}_2 \xrightarrow{R_{ref}} \mathcal{P}_4$, where $\mathcal{P}_4 \equiv \mathcal{P}_3$.*

Proof. We first consider the special case $\mathcal{P}_1 = \mathcal{P}_2 = \mathcal{P}$. By cases on the elements of R_{ref} it is not difficult to see that for any $\mathcal{P}$ there is at most one rule that defines the possible transition. Since transitions under any rule in R_{ref} are deterministic, it follows that $\mathcal{P}_3 \equiv \mathcal{P}_4$.

The general case follows from the special case by using Lemma 2, and considering that abstractions are equivalent up to α-substitutions. $\square$

Lemma 4. *For any $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\mathcal{P}_1 \equiv \mathcal{P}_2$, if $\mathcal{P}_1 \Downarrow_{R_{ref}} \mathcal{P}_3$, then $\mathcal{P}_1 \Downarrow_{R_{ref}} \mathcal{P}_4$, where $\mathcal{P}_4 \equiv \mathcal{P}_3$.*

Proof. It follows immediately by iterating on the steps of the derivation and applying Lemma 3. □

Corollary 5. *R_{ref} is deterministic.*

3 Garbage Collection

Informally we can define garbage collection as the removal of bindings that do not affect a program's result. In order to formalize this concept we need to devise a way to determine if a specific binding will or will not affect a program's result. One common and conservative solution is to build the graph of references between objects starting from the program's root-set (the *connectivity graph*). Any binding that does not belong to this graph can never be retrieved, and so cannot influence the program's result.

Consider for example the connectivity graph in Figure 2, where the root-set is represented by the single location l_i. The location l_k can be reached following the indicated path. l_m, on the other hand, cannot be reached, and is thus considered garbage.

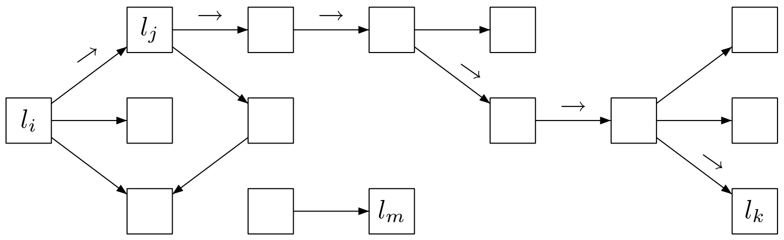

Figure 2: A chain of references in a λ_{ref} program.

We will refer to λ_{ref} extended with garbage collection as λ_{gc}. In λ_{gc} we collect references, or to be more precise, the bindings associated with references. A reference is reachable if it belongs to the root-set (the set of occurring locations in the program expression), or if it occurs in any expression bound to a reference that is reachable from the root-set. The *reachable* function expresses this relation and indicates whether a particular location is reachable in a given program. It is defined as the least fixed-point that satisfies

$$reachable(l_i, e, H) = (l_i \in LO(e)) \vee (\exists l_j \in LO(e) \mid reachable(l_i, H(l_j), H))$$

Although this definition is rather abstract, it conveys clearly the idea of traversing the graph of references (rooted in an expression) in search of reachable locations. An actual implementation of *reachable* can follow any traditional tracing algorithm, and hence, is decidable.

To discover collectable locations we use the *dead* function, defined as

$$dead(l_i, e, H) = (l_i \in dom(H)) \wedge \neg reachable(l_i, e, H)$$

Death is an invariant property under R_{ref}: once a reference becomes unreachable, it can never become reachable again. The following two lemmas states this more formally.

Lemma 6. *For any e_1, H_1 and l_i, if $dead(l_i, e_1, H_1)$ and $\langle e_1, H_1\rangle \overset{R_{ref}}{\longrightarrow} \langle e_2, H_2\rangle$, then $dead(l_i, e_2, H_2)$.*

Proof. By cases on the elements of R_{ref} and using the definition of *dead*.

Suppose that exists a l_i such that $dead(l_i, e_1, H_1)$ and $\neg dead(l_i, e_2, H_2)$. Since there is no transition that removes locations from an environment, $\neg dead(l_i, e_2, H_2)$ implies $reachable(l_i, e_2, H_2)$.

Location l_i can become reachable in $\langle e_2, H_2\rangle$ only if one of the following conditions is satisfied:

(i) l_i appears in e_2.

(ii) There is some l_j that occurs in e_2 and l_i is reachable in $H_2(l_j)$.

The first condition can hold only if l_i appears in e_1, or if e_1 has a subterm with the form $!l_k$ and l_i occurs in $H(l_k)$. But in both cases we would have $\neg dead(l_i, e_1, H_1)$, contradicting our hypothesis. Notice that we don't need to consider **alloc** because we suppose that $l_i \in dom(H_1)$.

If the second condition holds, either l_i is not reachable in $H_1(l_j)$, or l_j is dead in $\langle e_1, H_1\rangle$. l_i can become reachable in $H_2(l_j)$ only by an **assign** transition. In this case l_i, or an l_k such that $reachable(l_i, H_1(l_k), H_1)$, had to occur in e_1, and thus l_i would not be dead in $\langle e_1, H_1\rangle$. Finally, as proved in the previous paragraph, if l_j is dead in H_1, it cannot appear in H_2. □

Lemma 7. *For any e_1, H_1 and l_i, if $dead(l_i, e_1, H_1)$ and $\langle e_1, H_1\rangle \overset{R_{ref}}{\Longrightarrow} \langle e_2, H_2\rangle$, then $dead(l_i, e_2, H_2)$.*

Proof. It follows immediately by iterating on the steps of the derivation and applying Lemma 6. □

We can now define the transition rule that represents the collection of a single binding: references that are not reachable from the root-set are simply removed from H (we will refer to R_{ref} augmented with garbage collection as R_{gc}).

$$\langle e, H\rangle \to \langle e, H[l_i \mapsto \bot]\rangle \qquad \text{if } dead(l_i, e, H) \tag{\textbf{gc1}}$$

Since a collected location can be reintroduced (reused) by an allocation, the introduction of **gc1** invalidates Lemma 7. Nevertheless, reused locations are semantically distinct, and this reuse has no significance in the language defined so far.

A result similar to Lemma 4 can be proven for R_{gc}. To develop this proof we use a result analogous to the Postponement Lemma defined in [Morrisett et al., 1995].

Lemma 8. *Let $\xrightarrow{gc}$ denote any **gc1** transition. If $\mathcal{P}_1 \xrightarrow{gc} \mathcal{P}_2 \xrightarrow{R_{ref}} \mathcal{P}_3$ then there exists a $\mathcal{P}_4$ such that $\mathcal{P}_1 \xrightarrow{R_{ref}} \mathcal{P}_4 \xrightarrow{gc} \mathcal{P}_5$ and $P_3 \equiv P_5$.*

Proof. By cases on the elements of R_{gc} and using Lemma 6.

Except for **assign**, the applicability of any rule in R_{ref} to a given program is determined exclusively by the program expression. Since **gc1** transitions never change a program expression, they do not affect the applicability of those rules.

The applicability of **assign** is also not affected by **gc1** transitions, as these do not add locations to a program's environment, nor remove non-dead locations from it.

Even though the applicability of **alloc** is not affected by **gc1** transitions, its outcome can be: the name of the allocated location may change after a garbage collection. Nevertheless, all resulting programs are structurally congruent.

The proof is easily completed using Lemma 6. □

Lemma 9. *For any $\mathcal{P}$, if $\mathcal{P} \overset{R_{gc}}{=} v_1$ then $\mathcal{P} \overset{R_{ref}}{=} v_2$, where $v_2 \equiv v_1$.*

Proof. Consider a finite sequence of reductions:

$$\mathcal{P} \xrightarrow{R_{gc}} \mathcal{P}_1 \xrightarrow{R_{gc}} \ldots \xrightarrow{R_{gc}} \mathcal{P}_{n-1} \xrightarrow{R_{gc}} \mathcal{P}_n$$

and suppose that $\mathcal{P}_n = \langle v_1, H_n\rangle$. Using Lemma 8 and induction we can rewrite an alternative reduction sequence where all the garbage-collection transitions are performed at the end of the transition sequence:

$$\mathcal{P} \xrightarrow{R_{ref}} \mathcal{P}'_1 \xrightarrow{R_{ref}} \ldots \xrightarrow{R_{ref}} \mathcal{P}'_i \xrightarrow{gc} \ldots \xrightarrow{gc} \mathcal{P}'_{n-1} \xrightarrow{gc} \mathcal{P}'_n$$

where $\mathcal{P}'_n \equiv \langle v_1, H_n\rangle$. Since garbage-collection transitions never change an expression, $\mathcal{P}'_i = \langle v_2, H'_i\rangle$ where $v_2 \equiv v_1$. By cases on the elements of R_{ref} and using Lemma 4 it is easy to see that $\mathcal{P} \overset{R_{ref}}{=} v_3$ where $v_3 \equiv v_1$. □

Lemma 10. $R_{ref} \simeq R_{gc}$.

Proof. Consider any decidable program $\mathcal{P}$ such that $\mathcal{P} \Downarrow_{R_{ref}} \langle v, H \rangle$. Since $R_{gc} \supset R_{ref}$, following only rules in R_{ref} it is always possible to replicate the same sequence of transitions so that $\mathcal{P} \Downarrow_{R_{gc}} \langle v, H \rangle$.

The proof follows by Lemma 9. □

A transition under **gc1** represents the collection of a single reference. A *garbage collection cycle* (denoted by $\overset{gc}{\Longrightarrow}$) over a program $\mathcal{P}$ can be defined as an uninterrupted sequence of transitions under **gc1** such that if $\mathcal{P} \overset{gc}{\Longrightarrow} \langle e, H \rangle$, then all locations in H's domain are reachable.

Lemma 11. *Garbage collection cycles always terminate.*

Proof. Any program has a finite number of dead locations, which decreases by one with every **gc1** transition . □

A simple alternative to enforce that garbage collection happen in cycles is to replace **gc1** by the following rule

$$\begin{aligned} &\langle e, H_1 \rangle \rightarrow \langle e, H_2 \rangle \\ &\qquad\qquad \text{if } \exists l_i \mid dead(l_i, e, H_1) \qquad\qquad (\mathbf{gc2}) \\ &\qquad\qquad \text{where } H_2 = H_\emptyset[l_j \mapsto H_1(l_j)]^{l_j \in \{l_k \mid reachable(l_k, e, H_1)\}} \end{aligned}$$

In this transition the environment is replaced by a copy of it, where only the originally reachable locations are defined, thereby implicitly disposing all unreachable bindings[4]. Notice that although the condition $\exists l_i \mid dead(l_i, e, H_1)$ is not actually necessary to initiate a garbage collection cycle, without it **gc2** could be indefinitely applied to any program.

4 Finalizers

Finalizers are cleanup routines that are automatically invoked in garbage-collected languages before object disposal, allowing the management of application resources in the same way as heap memory. Often viewed as a natural evolution or a counterpart of C++ destructors in garbage-collected languages[5], finalizers have been the focus of much debate and its use is discouraged by many authors (e.g., [Bloch, 2001, Richter, 2002]), except for very specific situations.

In languages that employ tracing garbage collectors, finalizer invocation is generally asynchronous, and hence unpredictable. In addition to that several

[4] *Copy-collectors* follow a similar disposal pattern.

[5] Unfortunately in many languages such as C#, Perl and Python, finalizers are called destructors. We believe that this is misleading and can be a source of confusion among programmers. In this paper we use exclusively the term finalizer to refer to garbage-collected languages' finalization mechanism.

implementations give no guarantees on the order in which finalizers are invoked or even whether a finalizer will be invoked at all[6].

As a result of this lack of guarantees, finalizers, unlike destructors, have limited use in the release of timely critical resources such as file handles[7]. Furthermore, since finalizers can be executed at any time, this can lead to race conditions, and in extreme cases, even deadlocks.

Another negative aspect of finalizers is that garbage collectors require additional routines to deal with finalization-enabled objects. This slows memory allocation, delays memory reclamation, and can burden application performance.

In spite of these problems, most garbage-collected languages support some kind of finalization mechanism. Its use is considered legitimate at least in the following situations:

- As a mechanism to convey information gathered by the garbage collector to the client program. Boehm [Boehm, 2003] describes an application that uses complex DAGs with file descriptors as their leaves. In this case it is very hard to track all references to the file handlers, and explicitly close them after the last reference is dropped. A simple solution is to attach finalizers to the DAG leaves, which will close each file sometime after the corresponding leave becomes unreachable.

- To release memory allocated using a native routine such as `malloc`. Despite the fact that memory is a finite timely critical resource, its release with finalizers imposes a delay that is characteristic of tracing garbage collectors. If memory becomes scarce, the garbage collector will run more often.

- As a fallback mechanism for releasing non-memory finite resources that should have been explicitly released elsewhere. Although there is no guarantee on when or whether the finalizer will be invoked, this promise is still better than never being invoked (as when the programmer forgets to explicitly release the resource).

- To control object disposal. If the cost of creating instances of a class is high, an application can recycle objects by keeping unused instances in an object pool. After the last reference to an object is dropped, the object's finalizer can decide if the object will be disposed or recycled (resurrected) based on

[6] Garbage collectors that use a conservative tracing algorithm may retain objects after they become unreachable, and many language implementations do not invoke finalizers of live objects when applications exit.

[7] Actually, finalizers might be useful in the release of timely critical resources in systems that suport the explict invocation of the garbage collector and a call that runs the finalization methods of any objects pending finalization. In that case client programs should trigger garbage collection and finalization whenever the resources become scarce.

the number of objects already available in the pool. New objects are created only if the pool is empty.

– To break cycles in systems that use reference counting. Christiansen and Torkington [Christiansen and Torkington, 2003] describe an interesting example in Perl that defines finalizers for all classes that implement potentially cyclic data structures (rings, doubled-linked lists, graphs, etc). When the last reference to the data structure is dropped, its finalizer is invoked and explicitly breaks any cycles, thus avoiding memory leaks.

Finalizers are usually represented as functions that are automatically executed after a binding becomes unreachable. To register a finalizer for execution we extend λ_{gc} with the $finalize$ operator: the expression $\lambda x_i.e\ finalize\ l_i$ registers the function $\lambda x_i.e$ as a finalizer for the location l_i.

$$\langle v\ finalize\ l_i, H, F\rangle \rightarrow \langle nil, H, F[l_i \mapsto v]\rangle \quad \text{where } l_i \in dom(H) \qquad (\textbf{fin-reg})$$

To keep track of registered finalizers, a second environment (F) is added to the program context[8]. We will refer to R_{gc} augmented with garbage collection as R_{fin}, and the corresponding language as λ_{fin}.

Finalizer execution should be concurrent or interleaved with the evaluation of program expressions. A simple way to model this dynamics is by employing the sequencing notation, as described in **fin-exec1**. A finalizer associated with a dead location can be invoked at any time during the program evaluation, receiving as a sole parameter the reference being finalized.

$$\langle e_1, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e_1, H, F[l_i \mapsto \bot]\rangle \quad \text{if } (l_i \in dom(F)) \wedge dead(l_i, e, H) \qquad (\textbf{fin-exec1})$$

As a consequence of how finalizers are scheduled for execution, which in many respects is analogous to a multithreaded application scheduling dynamics, languages that support finalizers typically are non-deterministic. This is clearly the case of λ_{fin}, as stated by the following lemma.

Lemma 12. *R_{fin} is non-deterministic.*

Proof. By counterexample. Consider the initial program

$let\ x_i = new\ in$
  $let\ x_j = new\ in$
    $(\lambda x_k.x_i := nil)\ finalize\ x_j;$
    $x_i := \lambda x_k.x_k;$
    $!x_i$

[8] Although not shown, the remaining rules in Figure 1 must be modified to include F. From now on we will also omit new context rules.

which reduces to $\langle(!l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.l_i := nil]\rangle$. Two different transition sequences under R_{fin} that lead to different results are:

(i) $\langle(!l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.l_i := nil]\rangle$
$\overset{\textbf{deref}}{\rightarrow} \langle(\lambda x_k.x_k, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.l_i := nil]\rangle$

(ii) $\langle(!l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \lambda x_k.l_i := nil]\rangle$
$\overset{\textbf{fin-exec1}}{\rightarrow} \langle(\lambda x_k.l_i := nil)l_j; !l_i, H_\emptyset[l_i \mapsto \lambda x_k.x_k], F[l_j \mapsto \bot]\rangle$
$\overset{R_{fin}}{\Longrightarrow} \langle nil, H_\emptyset[l_i \mapsto nil], F[l_j \mapsto \bot]\rangle$

□

Corollary 13. $R_{gc} \not\simeq R_{fin}$

Under **fin-exec1** transitions a finalizer always receive a dead location as its parameter. Moreover, since a finalizer actually represents a function closure, it may refer to other locations, some of which can also be dead. This introduces the possibility of binding resurrection: if during finalizer execution an unreachable location is assigned to a reachable one, the assigned location may remain reachable after the finalizer execution ends (obviously, during finalizer execution all dead locations that occur in the finalizer resurrect). To avoid the problem of dangling references we must postpone the disposal of any reference that can be reached by registered finalizers. Furthermore, as the reference being finalized is passed as the finalizer argument, a finalizer must execute before the respective reference is disposed[9]. To model this semantics we replace **gc1** by

$$\begin{aligned}\langle e, H, F\rangle &\rightarrow \langle e, H[l_i \mapsto \bot], F\rangle \\ &\quad \text{if } dead(l_i, e, H) \wedge (l_i \notin dom(F)) \\ &\quad \wedge (\nexists l_j \in dom(F) \mid reachable(l_i, F(l_j)l_j, H))\end{aligned} \qquad (\mathbf{gc3})$$

With the addition of this transition rule, whenever there is a finalizer associated with a binding, the binding disposal is delayed (this behavior is typical in most actual language implementations). In memory intensive applications this deferral may hamper performance significantly, and even lead to memory exhaustion as the client program allocates objects faster than the garbage collector is able to dispose them. For instance, a Java program that uses a few threads to concurrently instantiate thousands of objects from a very simple finalizable class, without keeping any references to these objects, can very easily run out of memory.

[9] As an alternative to this semantics, we could redefine the concept of reachability to explicitly include (trace) locations in all registered finalizers. This however introduces a problem: objetcs with finalizers that have cyclic references are never collected.

Finalizer invocation order can be important in many situations. Consider for example a buffered file class that is structured as an aggregation of a file and a buffer object. In this case finalization order clearly matters: the buffer should be flushed before the file is closed.

Garbage collectors may try to order finalizer invocation using either topological or chronological (elaboration order) information, but the former is usually more meaningful from an application correctness perspective. An object should be kept in a valid state as long as it is needed by other objects (or their finalizers)[10]. So a finalizer should only be executed if all finalizers that refer to its associated object have already been executed. To model this semantics we replace **fin-exec1** by

$$\begin{aligned}\langle e, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e, H, F[l_i \mapsto \bot]\rangle&\\ \text{if } (l_i \in \mathit{dom}(F)) \wedge \mathit{dead}(l_i, e, H)&\\ \wedge(\nexists l_j \in \mathit{dom}(F) \mid \mathit{reachable}(l_i, F(l_j)l_j, H))&\end{aligned} \qquad (\textbf{fin-exec2})$$

Notice however that there is a major problem with this transition: it fails to invoke finalizers that refer to each other forming cycles. In this particular case it becomes impossible to unequivocally determine a best invocation order, so we break the cycles and force finalization by selecting any arbitrary order. **fin-exec2** is thus replaced by

$$\begin{aligned}\langle e, H, F\rangle \rightarrow \langle F(l_i)l_i;\ e, H, F[l_i \mapsto \bot]\rangle&\\ \text{if } (l_i \in \mathit{dom}(F)) \wedge \mathit{dead}(l_i, e, H)&\\ \wedge(\nexists l_j \in \mathit{dom}(F) \mid \mathit{reachable}(l_i, F(l_j)l_j, H)&\\ \wedge\neg \mathit{reachable}(l_j, F(l_i)l_i, H))&\end{aligned} \qquad (\textbf{fin-exec3})$$

Finally, notice that in the semantics described here, even if a location is resurrected, its finalizer is executed at most once. On the other hand, there is nothing that prevents registering finalizers for resurrected bindings. In some languages (e.g., Java), apparently due to arbitrary reasons, finalizers cannot be reenabled after being executed.

5 Final Remarks

In this paper we used an operational approach to develop a formal model for reasoning about garbage collection and its interaction with client programs. By explicitly representing low-level details, such as heap memory and its addresses, we were able to clearly specify memory management actions, and prove several important memory-related language invariants.

[10] Finalized objects can be considered, from a semantic point-of-view, in an invalid state.

Our main interest in developing this model was to describe a formal semantics for finalizers and weak references, exploring some of its many subtleties. As long as we know, this has not been addressed by other authors.

Automatic finalization, as we have shown, is a complex programming facility that imposes significant restrictions on the garbage collector, and makes the underlying language non-deterministic. Weak references, which were not considered in this paper, are a less known abstraction, with a broad but far from uniform actual language support. In the future we intend to specify and investigate its semantics with the aid of the model developed in this paper.

References

[Atkins and Nackman, 1988] Atkins, M. C. and Nackman, L. R. (1988). The active deallocation of objects in object-oriented systems. *Software – Practice and Experience*, 18(11):1073–1089.

[Bloch, 2001] Bloch, J. (2001). *Effective Java Programming Language Guide*. The Java Series. Addison-Wesley.

[Boehm, 2003] Boehm, H.-J. (2003). Destructors, finalizers, and synchronization. In *Proceedings of the 2003 ACM Symposium on Principles of Programming Languages*, pages 262–272. ACM Press.

[Chirimar et al., 1992] Chirimar, J., Gunter, C. A., and Riecke, J. G. (1992). Proving memory management invariants for a language based on linear logic. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 139–150. ACM Press.

[Christiansen and Torkington, 2003] Christiansen, T. and Torkington, N. (2003). *Perl Cookbook*. O'Reilly, 2 edition.

[Demers et al., 1990] Demers, A., Weiser, M., Hayes, B., Boehm, B., Bobrow, D., and Shenker, S. (1990). Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 1990 ACM Symposium on Principles of Programming Languages*, pages 261–269. ACM Press.

[Dybvig et al., 1993] Dybvig, R. K., Bruggeman, C., and Eby, D. (1993). Guardians in a generation-based garbage collector. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–216. ACM Press.

[Elsman, 2003] Elsman, M. (2003). Garbage collection safety for region-based memory management. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 123–134. ACM Press.

[Hayes, 1992] Hayes, B. (1992). Finalization in the collector interface. In *Proceedings of the 1992 International Workshop on Memory Management*, volume 637 of *LNCS*, pages 277–298. Springer-Verlag.

[Hudak, 1986] Hudak, P. (1986). A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 351–363. ACM Press.

[Hunter and Krishnamurthi, 2003] Hunter, R. and Krishnamurthi, S. (2003). A model of garbage collection for OO languages. In *10th International Workshop on Foundations of Object-Oriented Languages (FOOL10)*.

[Jones and Lins, 1996] Jones, R. and Lins, R. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, NY, USA.

[Mason and Talcott, 1990] Mason, I. and Talcott, C. (1990). Reasoning about programs with effects. In *Proceedings of the 190 International Workshop on Programming Language Implementation and Logic Programming (PLILP '90)*, volume 456 of *LNCS*, pages 189–203. Springer-Verlag.

[Morrisett et al., 1995] Morrisett, G., Felleisen, M., and Harper, R. (1995). Abstract models of memory management. In *Proceedings of the 1995 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 66–77. ACM Press.

[Morrisett and Harper, 1998] Morrisett, G. and Harper, R. (1998). Semantics of memory management for polymorphic languages. In *Higher Order Operational Techniques in Semantics*, pages 175–226. Cambridge University Press.

[Plotkin, 1981] Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.

[Richter, 2002] Richter, J. (2002). *Applied Microsoft .NET Framework Programming.* Microsoft Press.

[Schwartz and Melliar-Smith, 1981] Schwartz, R. L. and Melliar-Smith, P. M. (1981). The finalization operation for abstract types. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 273–282. IEEE Press.

[Wilson, 1992] Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, volume 637 of *LNCS*, pages 1–42. Springer-Verlag.