

A Calculus for Java’s Reference Objects

Yarom Gabay

Assaf J. Kfoury

{yarom, kfoury} @cs.bu.edu
Computer Science Department
Boston University
Boston, MA 02215

Abstract

Java’s *Reference* objects provide the programmer with limited control over the process of memory management. Although reference objects are often helpful, they introduce nondeterminism into program evaluation and lead to ambiguous program outcome. In this paper we present a calculus to formally reason about Java’s Reference objects. We model multiple levels of reference objects in a single calculus and apply a different garbage collection policy to each one of them. Accordingly, *weak* references are given the semantics of eager collection and *soft* references are given the semantics of lazy collection. In addition, we constrain garbage collection with the scarcity of two resources: time and space. We demonstrate the viability of our calculus by modeling a Java program which addresses a commonly-encountered caching problem. Using our model, we reason about the program’s evaluation and interaction with the garbage collector.

Keywords: Formal Languages, Java Reference Objects, Weak References, Garbage Collection

1 Introduction

One of the reasons for the rising popularity of Java is the fact that it provides automatic memory management. A garbage collector completely frees the programmer from the task of tracking the lifecycle of memory objects. As a result, using a garbage collector considerably reduces the amount of programming errors and thus the development cost. There are programming tasks where it may be beneficial to have some knowledge of the lifecycle of an object or even a certain level of control over memory management. The Reference class hierarchy, described in [1], was introduced into the language for this reason. Objects of the Reference class, known here as non-strong references, are references that do not prevent the garbage collector from collecting the referred objects. They essentially provide the programmer with a limited amount of interaction with the process of garbage collection.

This interaction with the garbage collector, although helpful in some cases, introduces nondeterminism and unpredictability into the language. Garbage collection usually implies nondeterministic program evaluation resulting from the fact that program-evaluation points where garbage collection occurs change from one run to another. This nondeterminism however does not affect program result since the garbage collector removes heap objects that can no longer be used by the program. This is no longer the case with programs that use non-strong references. Non-strong references provide the programmer with the tools to inquire about the existence of heap objects or the collection thereof. As a result, actions taken by the garbage collector are the basis for decision making in the program and its evaluation.

In this paper we present the calculus $\lambda_{\text{multiref}}$ to formally reason about non-strong references. $\lambda_{\text{multiref}}$ provides a framework to model multiple levels of non-strong references such as weak and soft references. We base our efforts

on λ_{weak} , given by Donnelly, Hallett and Kfoury in [3], and λ_{gc} , given by Morrisett, Felleisen and Harper in [7]. By using a garbage collection approach different from the one used in λ_{weak} and λ_{gc} , we model several levels of non-strong references in a single calculus. We apply a different garbage collection policy to each type of non-strong reference. Corresponding to the semantics given to the various non-strong references in Java’s API Specification [1], we apply a policy of “eager” garbage collection to *weak* references and a policy of “lazy” garbage collection to *soft* references. In addition, we condition garbage collection upon the scarcity of two resources: time and space. While the scarcity of time may restrain garbage collection, the scarcity of memory may trigger it.

In addition, we compare our garbage collection approach to the one taken in λ_{weak} and λ_{gc} and show our approach is capable of covering more garbage collection strategies. We give examples of plausible garbage collection scenarios that can only be modeled by our approach.

To demonstrate the usefulness of $\lambda_{\text{multiref}}$, we use it to model a Java example inspired by an online discussion [2]. The Java program in this example addresses a commonly-encountered caching problem. Because the program’s behavior in Java created much confusion, we offer our model to examine the program’s evaluation and interaction with the garbage collector. In this example, given in Figure 1, a list-like data structure and non-strong references are used to implement an automatically managed cache of `Apple` objects which are commonly used at multiple points of the program. The function `requestObject` is used to get an `Apple` object by its identifying index. This function encapsulates the decision of whether to create a new object when requested or return the existing one kept in the cache. If the `Apple` object does not exist in `list` it is created, saved in `list` by calling `list.set` and returned to the caller. Otherwise, the object retrieved from `list` is returned. The entries of `list` are non-strong references pointing to the actual objects in the memory. An `Apple` object that has no references other than the one kept in `list` can be collected by the garbage collector. On the other hand, if a reference resulting from a call to `requestObject` still exists then the object has a strong reference and may not be considered for garbage collection.

```

public Apple requestObject(List<Reference<Apple>> list, int appleIdx) {
    Apple apple = list.get(appleIdx).get();
    if(apple == null){
        apple = new Apple(appleIdx);
        list.set(appleIdx, new WeakReference<Apple>(apple));
        return apple;
    }else{
        return apple;
    }
}

```

Figure 1: Automatically Managed Cache in Java

Using our calculus, we give an explicit representation of the cache list on the heap. We compare the behavior of the modeled list considering different kinds of non-strong references. We show that weak references, which are eagerly collected, are inappropriate for the purpose of the cache list whereas soft references, which are lazily collected at the discretion of the garbage collector, are much more suitable for the task.

The rest of the paper is organized as follows. In Section 2, we formalize the notion of reachability upon which we base our calculus. We present and discuss our calculus, $\lambda_{\text{multiref}}$, in Section 3. In Section 4, we use our calculus to model the above-mentioned Java example. We conclude our work in Section 5.

2 Reachability Based Garbage Collection Model

$\lambda_{\text{multiref}}$ is based on the calculi λ_{weak} and λ_{gc} . The garbage collector model in these calculi, inspired by Felleisen and Hieb’s work in [4], uses a free variables approach to determine the disposability of heap objects. We find this

approach to be unsuitable for modeling multiple levels of non-strong references. We take a different approach based on the reachability of heap objects. Our reachability based approach provides the flexibility needed in order to apply different garbage collection policies to different strengths of reachability. In addition, there are garbage collection decisions that can only be modeled by our approach. In this sense, the reachability based approach provides a more accurate model for garbage collection.

References in Java are available in several levels of strength. References that are created in the conventional way in Java, e.g. `MyObject myRef = new MyObject()`, are considered strong references preventing the garbage collector from removing any object pointed by them. In addition to strong references Java contains Reference objects, or non-strong references, as specified by [1]. In this paper we focus on two main types of non-strong references. A *weak* reference points to an object that may be collected right away. Weak references by themselves are not enough to keep an object on the heap. A *soft* reference points to an object which is essential to the computation but should be disposed when the memory becomes scarce. The garbage collector attempts to retain objects that are marked by soft references as much as possible discarding them only when memory reaches a certain condition unspecified by the language specification. Motivating examples for using weak and soft references are provided by [8].

In a language which has only strong references, heap objects can either be reachable or unreachable from the program. The set of reachable objects is determined by the set of class variables and method variables in the program pointing to heap objects. This set is usually referred to as the *root set* of the program. An object pointed by a variable in the root set of the program is reachable. In addition, an object might be indirectly reachable. That is, an object is reachable if there is another reachable object pointing to it. Such chain of references from the root set of the program to a heap object is called *reachability path*. An object may have more than one reachability path to it as well as have no reachability paths at all. If the object has no reachability paths it is deemed garbage and can be immediately collected by the garbage collector.

Non-strong references introduce additional complexity to the notion of reachability and thus to garbage collection. In the presence of non-strong references, there are several classes of reachability forming a linear order with respect to their strength. As described in [9], this order from strongest to weakest is strongly reachable, softly reachable, weakly reachable, phantomly reachable and unreachable.

Moreover, an object's reachability path may include different types of references. Consider for example the case where the only reachability path of an object contains both strong and weak references. Because its reachability path contains a weak reference the object is considered weakly reachable. A strongly reachable object, on the other hand, is an object that has at least one strong reachability path, i.e., a path which contains only strong references. Generally speaking, the rule determining the strength of an object reachability is the following: *An object is as reachable as the weakest reference on its strongest reachability path.*

Figure 2, inspired by [9], demonstrates the notion of reachability. Objects A, B, C, D and E are heap allocated objects. Object C is a weak reference pointing to object D. Objects A, B and C are strongly reachable since they each have a strong reachability path starting at the root set. Object A has three reachability paths, two of which are strong and one is weak. Object D is weakly reachable since its one reachability path goes through the weak reference C. Object E is unreachable since it has no reachability paths.

2.1 Formalizing Object Reachability

As a preliminary step to building the calculus, we formalize the notion of reachability discussed above. An object's reachability is determined by finding a path from the *root set* of the expression to the object.

Definition 2.1 (Root Set). *The root set of an expression e is $FV(e)$.*

We use $FV(e)$ to mean the set of free variables in the expression e . Definition 2.2 formalizes the notion of reachability path using the syntax of λ_{weak} [3].

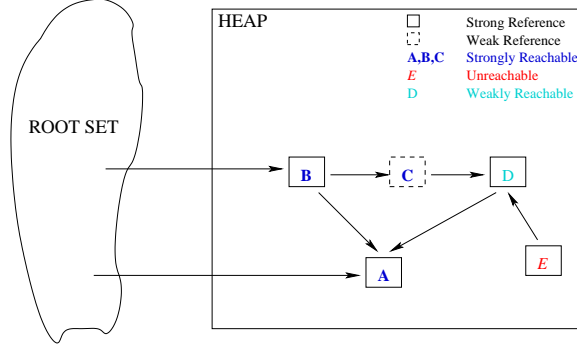


Figure 2: Example of Reachability Paths

Definition 2.2 (Reachability Path). *Let $(\text{letrec } H \text{ in } e)$ be a program and let hv be a heap value in H . A reachability path to hv is a sequence of unique variables, x_1, x_2, \dots, x_n , such that $n \geq 1$ and the following conditions hold:*

1. x_1 is in the root set of e , i.e., $x_1 \in FV(e)$, and
2. $x_{i+1} \in FV(H(x_i))$ for $1 \leq i \leq n - 1$ and
3. $H(x_n) = hv$.

By restricting the reachability path variables to be unique, we exclude cyclic reachability paths. Objects are considered *reachable* in a program if they have at least one reachability path. They are considered *unreachable* if they have no reachability paths. We next define the various reachability strengths.

Definition 2.3 (Strongly reachable). *Let $(\text{letrec } H \text{ in } e)$ be a program and let hv be a heap value in H . hv is strongly reachable in the program if hv has at least one reachability path, x_1, x_2, \dots, x_n , containing strong bindings only. That is, for every $1 \leq i \leq n$ and for every y $H(x_i) \neq \text{soft } y$ and $H(x_i) \neq \text{weak } y$.*

Definition 2.4 (Softly reachable). *Let $(\text{letrec } H \text{ in } e)$ be a program and let hv be a heap value in H . hv is softly reachable in the program if hv has no strong reachability paths and at least one reachability path, x_1, x_2, \dots, x_n , with no weak bindings. That is, for every $1 \leq i \leq n$ and for every y $H(x_i) \neq \text{weak } y$.*

Definition 2.5 (Weakly reachable). *Let $(\text{letrec } H \text{ in } e)$ be a program and let hv be a heap value in H . hv is weakly reachable in the program if hv has no strong reachability paths, no soft reachability paths and at least one reachability path. Note that all reachability paths to a weakly reachable object contain at least one weak binding.*

Although these definitions assume two levels of non-strong references, they can be generalized to any number of levels of non-strong references, provided their strengths form a linear order. For example, *phantomly reachable* could be added as another level of reachability whose strength falls between weakly reachable and unreachable.

3 The $\lambda_{\text{multiref}}$ Calculus

The syntax and operational semantics of $\lambda_{\text{multiref}}$ is given in Figure 3. Based on λ_{gc} , the heap, syntactically explicit in the language, is defined as a set of mutually recursive bindings tying heap values to variables. The evaluation of $\lambda_{\text{multiref}}$ programs generally includes: (1) evaluating the expression into a heap value, (2) creating a new binding pair of the heap value and a fresh variable onto the heap and (3) replacing the heap value in the expression with the new variable.

Programs:

(variables)	$w, x, y, z \in \text{Var}$	
(integers)	$i \in \text{Int}$	$::= \dots -2 -1 0 1 2 \dots$
(expressions)	$e \in \text{Exp}$	$::= x i \langle e_1, e_2 \rangle \pi_i e \pi_2 e \lambda x. e e_1 e_2 $ $\text{weak } e \text{ifdead } e_1 e_2 e_3 \text{soft } e$
(heap values)	$hv \in \text{Hval}$	$::= i \langle x_1, x_2 \rangle \lambda x. e \text{weak } x \text{d} \text{soft } x$
(heaps)	$H \in \text{Var} \xrightarrow{\text{fin}} \text{Hval}$	
(programs)	$P \in \text{Prog}$	$::= \text{letrec } H \text{ in } e$
(answers)	$A \in \text{Ans}$	$::= \text{letrec } H \text{ in } x$

Evaluation Contexts and Instruction Expressions:

(contexts)	$E \in \text{Ctx}$	$::= [] \langle E, e \rangle \langle x, E \rangle \pi_i E E e x E \text{weak } E $ $\text{ifdead } E e_1 e_2 \text{soft } E$
(instruction)	$I \in \text{Instr}$	$::= hv \pi_i x x y \text{ifdead } x e_1 e_2$

Rewrite Rules:

(alloc)	$\text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x]$ where x is a fresh variable
(π_i)	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$ provided $H(x) = \langle x_1, x_2 \rangle$ and $i \in \{1, 2\}$
(app)	$\text{letrec } H \text{ in } E[x y] \xrightarrow{\text{app}} \text{letrec } H \text{ in } E[e\{z := y\}]$ provided $H(x) = \lambda z. e$
(ifdead)	$\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{soft } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \text{d} \end{cases}$
(garb)	$\text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$ provided $\text{letrec } H \text{ in } e \Downarrow_{\text{gc-soft, gc-weak, gc-unr, ref2d}} \text{letrec } H' \text{ in } e$

Auxiliary:

(gc-soft)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc-soft}} \text{letrec } H \text{ in } e$ provided hv is softly reachable in $(\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)$ and $\text{isMemCond}(H \uplus \{x \mapsto hv\})$
(gc-weak)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc-weak}} \text{letrec } H \text{ in } e$ provided hv is weakly reachable in $(\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)$
(gc-unr)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc-unr}} \text{letrec } H \text{ in } e$ provided hv is unreachable in $(\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)$
(ref2d)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{ref2d}} \text{letrec } H \uplus \{x \mapsto \text{d}\} \text{ in } e$ provided hv is either soft y or weak y and $y \notin \text{Dom}(H)$

$$P \Downarrow_r P' \quad \equiv \quad P \xrightarrow{*} P' \text{ and } P' \text{ is irreducible with respect to the rule } r$$

$$P \Downarrow_{r_1, r_2} P' \quad \equiv \quad \text{there is a } P'' \text{ such that } P \Downarrow_{r_1} P'' \text{ and } P'' \Downarrow_{r_2} P'$$

Figure 3: The Syntax and Operational Semantics of $\lambda_{\text{multiref}}$

The calculus includes primitives for introducing and dereferencing weak and soft references. The constructs `weak e` and `soft e` are used for introducing weak and soft references respectively to the heap value resulting from the evaluation of the expression e . When an object is garbage collected all the non-strong references pointing to it are set to the special heap value d . The construct `ifdead e1 e2 e3` is used for conditionally dereferencing weak and soft references. This construct has the semantics of first evaluating e_1 into a non-strong reference and then taking an action which depends on whether the object pointed by the non-strong reference exists on the heap or not. If the object exists evaluation proceeds by applying the expression e_2 to the object. If the object does not exist, i.e. the non-strong reference points to the special value d , e_3 is evaluated. Note that `ifdead` encapsulates the test and the action of dereferencing the non-strong reference in a single construct.

The reduction semantics is given by the evaluation contexts and rewrite rules. In the style of λ_{weak} and λ_{gc} , it corresponds to a left-to-right, call-by-value reduction. The `(alloc)` rule introduces a pair of a heap value and a fresh variable onto the heap and replaces the heap value in the expression by the new variable. The `(garb)` rule is responsible for garbage collecting objects from the heap.

Objects in $\lambda_{\text{multiref}}$ can have one out of four levels of reachability. From strongest to weakest, these reachabilities are strongly reachable, softly reachable, weakly reachable and unreachable. Each level of reachability is handled separately in the auxiliary rules at the bottom of Figure 3. The auxiliary rules are defined in terms of the reachability definitions of Section 2.1. Strongly reachable objects are the only type of objects that are not collected. Softly reachable objects are collected by the `(gc-soft)` rule. The rule `(gc-weak)` handles the collection of weakly reachable objects, whereas `(gc-unr)` collects unreachable objects. After objects have been collected we need to fix the affected weak and soft references to point to the special value d . This is done by the rule `(ref2d)`.

The definition of reachability path and its use in $\lambda_{\text{multiref}}$ is demonstrated in Example 3.1. Note that multiple objects of the same value on the heap are considered different heap values. As an example, the heap $\{x \mapsto 1, y \mapsto 1\}$ contains two different objects of the value 1. The reachability of one object is independent from the reachability of the other.

Example 3.1 (Reachability Path). Let `(letrec H in e)` be a program with $e = x y$ and $H = \{x \mapsto \lambda x.x, y \mapsto \langle u, v \rangle, u \mapsto 1, v \mapsto 3\}$. The sequence y, u qualifies as a strong reachability path to the object 1 since $y \in FV(e)$, $u \in FV(H(y))$ and $H(u) = 1$. Therefore, The object 1 is reachable in the program.

In $\lambda_{\text{multiref}}$ a different collection policy is given to each one of the reachability levels. However, these should be viewed as suggested policies only. The calculus serves as a template where different garbage collection strategies can be evaluated in the context of multiple levels of reachability. The specific policies for each type of reachability are given below.

Strongly reachable objects are not collected at all in $\lambda_{\text{multiref}}$. The garbage collector completely ignores this type of objects. Unreachable objects, on the other hand, are collected immediately. Every time the garbage collector kicks in, it collects all unreachable objects. This is done by the arrow $\Downarrow_{\text{gc-unr}}$ in the `(garb)` rule.

We use the predicate `isMemCond` in the rule `(gc-soft)` to trigger the collection of soft references. `isMemCond(H)` is simply defined to count the number of bindings in H and return true only if this number exceeds a certain threshold. It is important to note that by replacing `isMemCond` with other predicates we can get different behaviors for softly reachable objects. For instance, by defining `isMemCond` to always return false, we give softly reachable objects the collection semantics of strongly reachable objects. On the other hand, by defining `isMemCond` to always return true, we make softly reachable objects eagerly collected.

We collect weakly reachable objects eagerly and softly reachable objects lazily. Lazy collection taken to the logical extreme is no collection at all. We therefore condition lazy collection upon a simple memory constraint such as the number of objects on the heap. This forms the concept of a space-aware lazy collection. Similarly, the logical extreme of eager collection is to collect all objects on every garbage collector cycle. This is how weak references are handled in $\lambda_{\text{multiref}}$. In an analogous manner, eager collection should be constrained by time considerations. To achieve this, we need to make the notion of time precise. We need to assign time units to each of the evaluation

operations including the (garb) rule. With this definition it is possible to construct a time-aware eager collection. We cover this topic further in [5].

In addition to having the flexibility of modeling multiple levels of reference strengths, our reachability-based model of garbage-collection covers more garbage collection cases than the method used in λ_{gc} and λ_{weak} . The program P in Figure 4 demonstrates this additional expressiveness. The bindings $y \mapsto z$ and $z \mapsto 2$ are both garbage in the program since $x \mapsto 1$ is the only binding needed in the evaluation of x . A reasonable garbage collection action in the evaluation of P is to collect the binding $z \mapsto 2$ while leaving $x \mapsto 1$ and $y \mapsto z$ on the heap. In $\lambda_{multiref}$, the object 2 is not strongly reachable in the program since there is no strong reachability path from the root set to it. In fact, there is no reachability path at all starting from the root set, x , and ending at the object 2. This operation can therefore be modeled in $\lambda_{multiref}$.

$P = \text{letrec } \{x \mapsto 1, y \mapsto z, z \mapsto 2\} \text{ in } x$ $\xrightarrow{\text{garb}} \text{letrec } \{x \mapsto 1, y \mapsto z\} \text{ in } x$
--

Figure 4: The Flexibility of The Reachability Approach to Garbage Collection

The same operation however cannot be modeled in λ_{gc} nor in λ_{weak} . Garbage collection in these calculi works by partitioning the heap in the program $\text{letrec } H \text{ in } e$ into two subsets H_1 and H_2 and removing H_2 provided there is no strong reference from the remaining program $\text{letrec } H_1 \text{ in } e$ to a binding in H_2 . This condition does not hold when we partition the heap in P into $H_1 = \{x \mapsto 1, y \mapsto z\}$ and $H_2 = \{z \mapsto 2\}$ as there is a strong reference from H_1 to z . Since this example does not contain weak references, it demonstrates the flexibility of $\lambda_{multiref}$ over the limitation in both λ_{gc} and λ_{weak} . A similar example that makes use of weak references can be easily constructed.

4 Modeling the Java Example

To shed light on the evaluation of the Java example presented in Section 1, we model the example in $\lambda_{multiref}$. We encode both the list, serving as the cache, and the code for `requestObject` into our calculus. Using the model, we examine the evaluation of the program and the interaction between `requestObject` and the garbage collector. We make several assumptions to facilitate the modeling of the example. We assume a special heap value `nil` in the calculus, which is used to mark the empty list. The rest of the assumptions appear later as we construct the example.

The cache list is modeled by using a sequence of pair constructs. A schematic of the list as it appears on the heap is given in Figure 5. The special variable `list` points to the first pair in the sequence. The first component of this pair is the variable y_1 , which points to the first list element, and the second component is the variable `list1`, which points to the rest of the list. y_2 points to the second element in the list and so on to the n th element. A list element is either a soft reference to the actual object or the special value `d` if the object is not stored in the cache list. For stored objects, y_i points to `soft z_i` where z_i points to the actual object. For simplicity, the actual objects stored in the cache are the integers representing the indexes of those objects in the list.

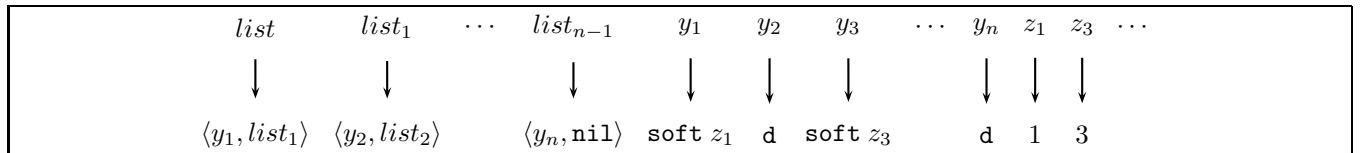


Figure 5: Heap Representation of The Cache List

We assume two list operations, `set` and `get`, for controlling the list. `set` is a function whose input is the list and an integer i . `set` modifies the list on the heap by changing the binding $y_i \mapsto d$ to $y_i \mapsto \text{soft } z_i$, where z_i is the variable which strongly binds the newly allocated object i on the heap. `set` returns z_i . `get` is a function whose

input is the list and an integer i . `get` traverses the list and returns y_i , the i th element in the list. The value bound to y_i is used to determine whether the object is stored in the cache or not. If the required object is on the heap y_i would point to `soft` z_i . Otherwise, it would point to the special value `d`.

During program evaluation, the `(garb)` rule operates occasionally to collect garbage from the heap. In our example, we assume the variable `list` is strongly referred in the program. This means that the variables `listj` for $1 \leq j \leq n - 1$ and y_i for $1 \leq i \leq n$ are all strongly reachable by the program since strong reachability paths from `list` to them exist. The variables z_i for $1 \leq i \leq n$ may not be strongly reachable. If the only path to a particular z_i is through the list then z_i is softly reachable. This corresponds to the case where the program is currently not using the object bound to z_i .

When `get(list, i)` is invoked the requested z_i is returned. As long as this reference is alive the object bound to z_i has a strongly reachable path, which prevents the collection of the object by the `(garb)` rule. When all strong references to the object are discarded the object is softly reachable again and may be garbage collected.

When the garbage collector operates the `(gc-soft)` auxiliary rule tests if the number of objects on the heap exceeds a certain limit. This is done by using `isMemCond`. As long as this threshold is crossed, a heap object may be removed provided it is softly reachable. Consequentially, a minimal number of softly reachable objects are removed to bring the heap size below the threshold again. The softly reachable objects in the cache list are candidates for that collection.

One of the suggestions in the discussion was to use weak references as the values bound to the y_i variables instead of soft references. This approach would result in a different cache behavior. When a z_i variable has a strong reachability path outside the list the object still cannot be collected. However, when the only reachability path is through the list, the object becomes weakly reachable. As a result it will be eagerly collected at the next garbage collector run.

The code for `requestObject(list, i)` is given in Figure 6. `get(list, i)` is used to retrieve y_i , the i th element in the list. By using `ifdead` we check whether the object exists on the heap. If y_i points to the special value `d` it means a request has been made to an object which is not on the heap. As a result, `set(list, zi)` is evaluated, where z_i is the variable binding the input parameter i . When `set` is invoked y_i 's binding is modified to $y_i \mapsto \text{soft } z_i$. The value returned by `requestObject(list, i)` is z_i , which is the value returned by `set`. On the other hand, if the value requested is on the heap then y_i points to `soft` z_i , where z_i points to the required object on the heap. As a result, the `(ifdead)` rule evaluates the expression $(\lambda x.x) z_i$ which returns z_i as a result.

```
requestObject(list, i)
  ifdead get(list, i) set(list, i) λx.x
```

Figure 6: $\lambda_{\text{multiref}}$ Program for `requestObject(list, i)`

5 Related and Future Work

Morrisett, Felleisen and Harper's λ_{gc} [7] provides a formal model for garbage collection and for heap allocated objects. Donnelly, Hallett and Kfoury's λ_{weak} [3] extends λ_{gc} to provide a model for weak references. In our calculus, we chose a different formalism for garbage collection which we showed is able to cover more garbage collection scenarios than the formalism used in λ_{weak} and λ_{gc} . Moreover, inspired by Java, our effort extends the work in λ_{weak} by including multiple levels of reachability along with different garbage collection strategies applied to each level. To our knowledge, this is the only attempt to formalize multiple levels of non-strong references.

Several directions can be taken as a continuation of this work. In this paper we chose a functional calculus, mostly for reasons of simplicity. In order to bring this model closer to non-strong references in Java, we can choose an object oriented calculus for our modeling. As an example, Featherweight Java [6] can be used to model non-strong references. In order to do so, Featherweight Java first has to be extended with a heap modeling and garbage

collection rules as done in λ_{gc} . Once we have that, we can model non-strong references, as presented in this paper, in the context of Featherweight Java.

Our calculus includes only soft and weak references. In our context this simple model suffices. It is however possible to generalize $\lambda_{multiref}$ to have n different levels of non-strong references and to give each one a different collection policy.

Finally, an extension of $\lambda_{multiref}$ could be used to study issues of tradeoffs between computational resources, such as time and space, required by programs that use non-strong references. With a calculus where time and space are formally defined, a Java program, like the example used in this paper, can be analyzed in order to approximate time and space consumption under different garbage collection strategies.

References

- [1] “Java TM 2 platform, standard edition, v 1.4.2 API specification.” [Online]. Available: <http://java.sun.com/j2se/1.4.2/docs/api/>
- [2] Archives of ADVANCED-JAVA@DISCUSS.DEVELOP.COM., Nov. 2003. [Online]. Available: <http://discuss.develop.com/archives/wa.exe?A2=ind0311&L=ADVANCED-JAVA&D=0&I=-3&P=4459>
- [3] K. Donnelly, J. J. Hallett, and A. Kfoury, “Formal semantics of weak references,” in *ISMM '06: Proceedings of the 2006 international symposium on Memory management*. New York, NY, USA: ACM Press, 2006, pp. 126–137.
- [4] M. Felleisen and R. Hieb, “A revised report on the syntactic theories of sequential control and state,” *Theoretical Computer Science*, vol. 103, no. 2, pp. 235–271, 1992. [Online]. Available: citeseer.ist.psu.edu/felleisen92revised.html
- [5] Y. Gabay and A. Kfoury, “Towards formalizing java’s weak references,” Boston University, CS Dept, MA, Tech. Rep. BUCS-TR-2006-031, Dec. 2006.
- [6] A. Igarashi, B. Pierce, and P. Wadler, “Featherweight Java: A minimal core calculus for Java and GJ,” in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99)*, L. Meissner, Ed., vol. 34(10), N. Y., 1999, pp. 132–146.
- [7] G. Morrisett, M. Felleisen, and R. Harper, “Abstract models of memory management,” in *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*. New York, NY, USA: ACM Press, 1995, pp. 66–77.
- [8] E. Nicholas. (2006, May) Understanding weak references. Java.net Article. [Online]. Available: http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w_1.html
- [9] M. Pawlan. (1998, Aug.) Reference objects and garbage collection. Sun Developer Network Article. [Online]. Available: <http://java.sun.com/developer/technicalArticles/ALT/RefObj/>