

# Towards Formalizing Java's Weak References

Yarom Gabay  
Boston University  
yarom@cs.bu.edu

Assaf J. Kfoury  
Boston University  
kfoury@cs.bu.edu

Technical Report: BUCS-TR-2006-031

## Abstract

Weak references provide the programmer with limited control over the process of memory management. By using them, a programmer can make decisions based on previous actions that are taken by the garbage collector. Although this is often helpful, the outcome of a program using weak references is less predictable due to the nondeterminism they introduce in program evaluation. It is therefore desirable to have a framework of formal tools to reason about weak references and programs that use them.

We present several calculi that formalize various aspects of weak references, inspired by their implementation in Java. We provide a calculus to model multiple levels of non-strong references, where a different garbage collection policy is applied to each level. We consider different collection policies such as eager collection and lazy collection. Similar to the way they are implemented in Java, we give the semantics of eager collection to *weak* references and the semantics of lazy collection to *soft* references. Moreover, we condition garbage collection on the availability of time and space resources. While time constraints are used in order to restrict garbage collection, space constraints are used in order to trigger it.

Finalizers are a problematic feature in Java, especially when they interact with weak references. We provide a calculus to model finalizer evaluation. Since finalizers have little meaning in a language without side-effect, we introduce a limited form of side effect into the calculus. We discuss determinism and the separate notion of uniqueness of (evaluation) outcome. We show that in our calculus, finalizer evaluation does not affect uniqueness of outcome.

## 1 Introduction

One of the reasons for the rising popularity of Java is the fact that it provides automatic memory management. A garbage collector frees the programmer completely from the task of tracking the lifecycle of memory objects. Therefore, using a garbage collector considerably reduces the amount of programming errors and thus the development cost. However, there are cases where it may introduce certain limitations. There are programming tasks where having some knowledge of the lifecycle of an object or even a certain level of control may be beneficial. For this reason, weak references were introduced into the language. A weak reference is a reference to an object that does not prevent the garbage collector from collecting the object. Essentially, it provides the programmer with a limited amount of interaction with the process of garbage collection. As we shall see later, this interaction, although helpful in some cases, introduces nondeterminism and unpredictability into the language. It is therefore desirable to be able to reason about programs using weak references.

Although the motivation for such a reference might be obscure at this point, in Section 3 we give practical examples of cases where such a feature may be useful.

Garbage collection usually implies nondeterministic program evaluation. This stems from the fact that the points where garbage collection occurs during program evaluation are changing from one run to another. However, this nondeterminism does not affect program result. A garbage collector removes heap objects that can no longer be used by the program. Therefore, program evaluation is independent of the actions taken by the garbage collector. Although there are exceptions to this rule, such as programs that make decisions based on the status of the memory, this is true in most cases. Weak references, however, completely break this rule. They provide the programmer with the tools to inquire about the existence of heap objects or the collection thereof. Therefore, actions taken by the garbage collector are the basis for decision making in the program and thus in the evaluation. Given all this, it is highly desirable to have a formal model to reason about weak references and their interaction with the garbage collector. Such a model serves as a mean to analyze evaluation scenarios that are otherwise complicated and hard to predict.

We provide several models for formalizing different aspects of weak references, inspired by the way they are implemented in Java. We base our efforts on  $\lambda_{weak}$ , a framework to reason about weak references, presented by Donnelly, Hallett and Kfoury in [2]. The garbage collection approach taken in  $\lambda_{weak}$  is based on  $\lambda_{gc}$ , defined by Morrisett, Felleisen and Harper in [7]. The advantage of this approach is its simplicity. We show, however, that this approach is incapable of modelling certain garbage collection scenarios. As an alternative, we offer  $\lambda_{weak1}$ , a more accurate, albeit slightly more complicated, garbage-collection model. In addition to being more accurate, the new approach facilitates the modelling of multiple levels of weak references. Our calculus,  $\lambda_{weak2}$ , aims at bringing the model closer to the implementation of weak references in Java by introducing several strengths of weak references and by considering different garbage collection policies for each one of them. The policies we consider are focused on two axes. The first axis is the eagerness in which the garbage collection collects objects. Some objects are given the semantics of eager collection and others the semantics of lazy collection. This approach roughly corresponds to the way *weak* references are eagerly collected in Java, while *soft* references are kept in memory until memory is running low. The second axis has to do with resource-contingent garbage collection. Specifically, we condition the collection of objects on two primary resources: time and space. We present a calculus where time and space tradeoffs can be evaluated by appropriately constraining garbage collection.

As part of our effort to formalize weak references and garbage collection in Java, we present  $\lambda_{weak3}$ , a calculus to model finalizers. Finalizers are a tricky Java feature. They are usually given the meaning of deallocating non-memory related resources, such as closing a file or a database connection. However, their nondeterministic behavior, partially stemming from the nondeterministic behavior of the garbage collector, makes programs that use finalizers for releasing critical resources highly error prone. Finalizers are generally evaluated for the side effects they produce. Therefore, we introduce a simple form of side effect into the calculus. With this limited level of side effect, we show that finalizers do not compromise predictability and uniqueness of evaluation outcome in  $\lambda_{weak3}$ .

In addition to the goals mentioned above, this report also serves pedagogical purposes. Particularly, the behavior of weak references and their interaction with the garbage collector is described. As stated above, we focus primarily on weak references in the context of Java.

The rest of the report is organized as follows. In section 2, we give an introduction to  $\lambda_{weak}$ ,

upon which we base our calculi. Section 3 reviews weak references in Java from a programmer perspective. Section 4 gives a more in-depth perspective to the way weak references interact with the garbage collector. In particular, we explain the notion of object reachability, which provides the basis for garbage collection decisions. In Section 5, we formalize the notion of reachability and present  $\lambda_{weak1}$ , an alternative model to  $\lambda_{weak}$  based on that formalism.  $\lambda_{weak2}$  is given in Section 6. This calculus defines multiple levels of strengths of weak references. In this context, we consider different garbage collection policies for each type of weak reference. In Section 7, we give an introduction to finalizers in Java. In Section 8, we present  $\lambda_{weak3}$ , a calculus to model finalizers. Finally, we conclude our efforts and offer possible avenues for future research in Section 9.

## 2 The $\lambda_{weak}$ Calculus

The calculi in this paper are based on  $\lambda_{weak}$ , a calculus defined by Donnelly, Hallett and Kfoury in [2].  $\lambda_{weak}$  provides a framework to formally reason about weak references. It serves as an abstract language in which the implementations of weak references in various languages can be modelled.  $\lambda_{weak}$  is inspired by  $\lambda_{gc}$ , a model for automatic memory management defined by Morrisett, Felleisen and Harper in [7]. In this section, we discuss the syntax and formal semantics of  $\lambda_{weak}$  as well as some of the detail of  $\lambda_{gc}$ . We also discuss non-determinism in the context of both  $\lambda_{gc}$  and  $\lambda_{weak}$ .

The syntax and operational semantics of  $\lambda_{weak}$  is given in Figure 1. The syntax defines the heap as part of the program. A heap is a set of mutually recursive bindings tying variables to heap values, which corresponds to heap allocated objects. As in  $\lambda_{gc}$ , memory allocation in  $\lambda_{weak}$  is explicit. Consequentially, evaluation usually includes: (1) evaluating the expression into a heap value, (2) replacing the heap value by a fresh variable and (3) creating a new binding, which ties the variable to the heap value, on the heap.

The calculus includes primitives for introducing and dereferencing weak references. The construct `weak  $e$`  is used for introducing a weak reference to the value resulting on the heap from evaluating the expression  $e$ . In contrast, `ifdead  $e_1 e_2 e_3$`  is used for conditionally dereferencing a weak reference. This construct has the following semantics: First,  $e_1$  is evaluated into a weak reference. The next action taken depends on whether the object pointed by the weak reference exists or not. If the object exists the expression  $e_2$  is applied to the object. If the object does not exist, i.e., the weak reference points to the special value `d`,  $e_3$  is evaluated. Note that `ifdead` encapsulates the test and the action of dereferencing the weak reference in a single construct.

The special value `d` represents a weak reference to an object that has been collected by the garbage collector. When an object is garbage collected all weak references pointing to that object are replaced by `d`. Accordingly, `ifdead` compares the weak reference against the value `d` in order to see if the object pointed by the weak reference has been collected. This can be seen in the rewrite rule (`ifdead`).

The reduction semantics is given by the evaluation contexts and rewrite rules. It corresponds to a left-to-right, call-by-value reduction. Garbage collection is done in a rewrite rule in a similar approach to one used in  $\lambda_{gc}$ . In this approach, reachability is computed by considering the set of free variables in the program, as can be seen in (`gc`). More specifically, garbage collection is done by deterministically partitioning the heap into two subheaps,  $H_1$  and  $H_2$ .  $H_2$  is then considered as target for collection. If it is not reachable from the program `letrec  $H_1$  in  $e$`  it is garbage collected. This reachability is computed by considering the intersection of the free variables of the program

<b>Programs:</b>			
(variables)	$w, x, y, z \in \text{Var}$		
(integers)	$i \in \text{Int}$	$::=$	$\dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$
(expressions)	$e \in \text{Exp}$	$::=$	$x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_i e \mid \pi_2 e \mid \lambda x.e \mid e_1 e_2 \mid$ $\text{weak } e \mid \text{ifdead } e_1 e_2 e_3$
(heap values)	$hv \in \text{Hval}$	$::=$	$i \mid \langle x_1, x_2 \rangle \mid \lambda x.e \mid \text{weak } x \mid \text{d}$
(heaps)	$H \in \text{Var} \xrightarrow{\text{fin}} \text{Hval}$		
(programs)	$P \in \text{Prog}$	$::=$	$\text{letrec } H \text{ in } e$
(answers)	$A \in \text{Ans}$	$::=$	$\text{letrec } H \text{ in } x$
<b>Evaluation Contexts and Instruction Expressions:</b>			
(contexts)	$E \in \text{Ctx}$	$::=$	$[] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E \mid \text{weak } E \mid \text{ifdead } E e_1 e_2$
(instruction)	$I \in \text{Instr}$	$::=$	$hv \mid \pi_i x \mid x y \mid \text{ifdead } x e_1 e_2$
<b>Rewrite Rules:</b>			
(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
( $\pi_i$ )	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$		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)	$\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \text{d} \end{cases}$		
(garb)	$\text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$		provided $\text{letrec } H \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H'' \text{ in } e$ and $\text{letrec } H'' \text{ in } e \Downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e$
<b>Auxiliary:</b>			
(gc)	$\text{letrec } H_1 \uplus H_2 \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H_1 \text{ in } e$		provided $\text{Dom}(H_2) \cap \text{FV}(\text{letrec } H_1^s \text{ in } e) = \emptyset$ and $H_2 \neq \emptyset$
(weak-gc)	$\text{letrec } H \uplus \{x \mapsto \text{weak } y\} \text{ in } e \xrightarrow{\text{weak-gc}} \text{letrec } H \uplus \{x \mapsto \text{d}\} \text{ in } e$		provided $y \notin \text{Dom}(H)$
	$H^s$	$=$	$\{x \mapsto H(x) \mid H(x) \notin \text{weak } y \text{ for any } y\}$
	$\text{FV}(H)$	$=$	$\left( \bigcup_{x \in \text{Dom}(H)} \text{FV}(H(x)) \right) - \text{Dom}(H)$
	$\text{FV}(\text{letrec } H \text{ in } e)$	$=$	$(\text{FV}(H) \cup \text{FV}(e)) - \text{Dom}(H)$

Figure 1: The Syntax and Operational Semantics of  $\lambda_{\text{weak}}$

and the domain of  $H_2$ . If these two sets are disjoint then  $H_2$  is not used in the program and thus can be garbage collected.

The above description corresponds to garbage collection in  $\lambda_{gc}$ . Garbage collection in  $\lambda_{weak}$  differs from that in two ways. First, when checking the reachability of  $H_2$  from the program only strong bindings are considered. This is achieved by using  $H_1^s$ , which corresponds to  $H_1$  without weak bindings. Using  $H_1^s$ , reflects the fact that weak references are not strong enough to keep an object on the heap. Second, collecting objects in  $\lambda_{weak}$  needs to be done with extra caution. More specifically, there are cases where an object is removed while still having weak references pointing to it. Therefore, the garbage collector needs to go over all of the weak reference of the removed object and turn them into the special value  $d$ . This is done in the rule (gc-weak). The rule (garb) is responsible for calling the two auxiliary rules, (gc) and (gc-weak), appropriately.

Evaluation in both  $\lambda_{gc}$  and  $\lambda_{weak}$  is nondeterministic. The nondeterminism stems from two main causes, both related to garbage collection. The first cause is the fact that garbage collection can occur at any point during evaluation where there is garbage to collect. It is possible to evaluate a program completely without using garbage collection even once. At the other extreme, garbage collection may occur at every step of evaluation or even occur at multiple consecutive times. Consequentially, a program may have infinitely many different evaluation paths. The second cause for nondeterminism comes from the way (gc) is defined. Every time the garbage collector is called it nondeterministically picks the garbage set  $H_2$ . This set can be chosen to be a single element, all the garbage on the heap or anywhere in between.

In spite of this nondeterminism, evaluation result in  $\lambda_{gc}$  is unique. Given a program in  $\lambda_{gc}$ , if there is one evaluation path that ends in a normal form then every evaluation path ends at the same value. The reason for this is that when objects are garbage collected they can no longer affect computation. Objects that are chosen for collection are unreachable and thus can no longer be used in the rest of the computation. The result of evaluation is therefore independent from the points at which the (gc) rule is called.

In contrast, evaluation result in  $\lambda_{weak}$  is not unique. Since weak references are present, garbage collection may remove objects that would have been used in the program otherwise. Depending on the points at which the (garb) rule is called, different branches of the program are chosen. This is reflected in the rule (ifdead). This rule rewrites the program differently depending on whether the object in question is dead or not. Therefore, two evaluation paths that differ in the points where garbage collection is called may result in two different values.

This behavior might be worrisome. Therefore, the authors define conditions under which a  $\lambda_{weak}$  program has a unique result regardless of the evaluation path chosen. In particular, the evaluation of programs satisfying those conditions is not affected by garbage collection. Programs that have this property are called *well-behaved* in [2]. Generally speaking, the set of well-behaved programs in  $\lambda_{weak}$  is undecidable. However, the authors define a proper subset of well-behaved programs which is efficiently decidable.

### 3 Weak References in Java

Automatic memory management is one of the most useful features in Java. The job of deallocating memory when it is no longer usable in the program is done by the garbage collector. Free of the obligation to clean up unused memory, the programmer can focus on other demanding tasks that constitute program authoring. While this is an advantage in most cases, there are cases where having

an automatic memory management could present an obstacle. There are examples where having some control over memory management or even just the indication of memory deallocation would facilitate program writing. Weak references were added to Java in order to address this problem. They provide the programmer with the tools to be notified by the runtime system about an object removal as well as the tools to have more control over memory deallocation. This section gives an introduction to Java's weak references and the justification thereof. Information and examples in this section are inspired by the online articles [10] and [9], as well as many other online sources.

References in Java are available in several levels of strength. Those created in the conventional Java way, e.g., `MyObject myRef = new MyObject()`, are considered strong references and thus have the semantics of regular references preventing the garbage collector from removing any object pointed by them. Aside from strong references, Java contains non-strong references. Note that until now we used the term weak reference to mean a general form of a non-strong reference. In the context of Java, we need to be more careful since weak references are only one out of three types of non-strong references.

Creating and managing non-strong references is done by using the Java package `java.lang.Ref`. The class hierarchy in `java.lang.Ref` is given in Figure 2. The abstract class `Reference` is

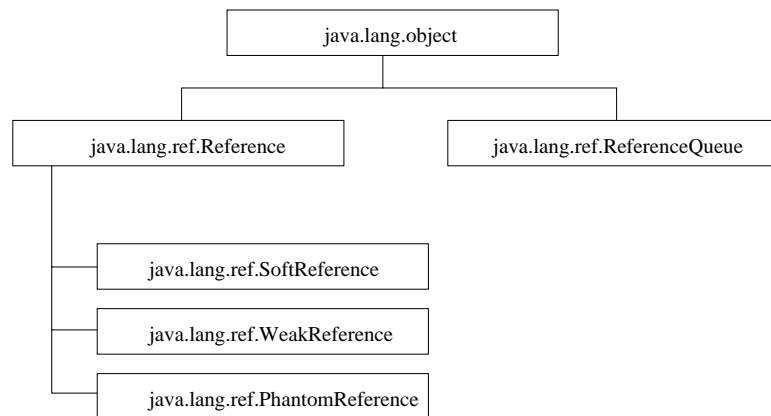


Figure 2: Java Class Hierarchy for Non-Strong References

the supertype of three types of non-strong references, which are `WeakReference`, `SoftReference` and `PhantomReference`. In order to create a non-strong reference in Java, the programmer creates one of the `Reference` subclasses and pass an object to it. This object will then have a non-strong reference to it. Note that creating a non-strong reference to an object introduces an additional level of indirection which does not exist in the case of strong references. In practice, a non-strong reference points to a strong reference that points to the object itself.

The four types of references in Java, string references included, express different levels of strength in which they cling to an object and prevent it from being collected. In fact, the strengths of each one of the reference types form a linear order, where strong references are naturally the strongest, soft references are second, third come weak references and lastly phantom references are the weakest form of references. The garbage collector handles objects differently depending on what kind of references point to them. The general rules guiding the garbage collector are presented below.

**Strong References.** An object pointed by a strong reference will not get collected. Strong references mark objects that are necessary to the rest of the computation.

**Soft References.** An object pointed by a soft reference (and no strong references) can potentially be collected. However, the garbage collector will try to hold on to the object as long as possible getting rid of it only when memory runs low. That is, soft references mark objects that are nice to have, but are not necessary to the rest of computation and thus should be removed when holding them is too costly.

**Weak References.** These are even weaker than soft references. An object pointed by a weak reference (and no strong nor soft references) will usually get collected the next time there is a garbage collector cycle. In essence, weak references by themselves are never enough to keep an object in memory. When are they useful then? They are useful when there is a need for an object whose existence depends on the existence of a second object. The first object should be in memory if the second object is in memory. However, as soon as the second object is collected we have no need for the first one anymore. A more concrete example of this is given below.

**Phantom References.** These are the weakest form of references in Java. Phantom References are not really references in the conventional sense. In fact, they do not provide access to the objects to which they refer. Phantom references are used together with reference queues (see Figure 2) in order to keep track of object's lifecycle. Phantom references are beyond the scope of this paper.

Note that when an object is pointed by references covering several levels of strengths, the strongest reference will determine how the object is to be handled. For instance, if the object is pointed both by a weak reference and a soft reference it will be collected according to the rules applying to soft references.

This summarizes four different categories of references in Java. Still, the motivation of using different types of references might be unclear at this point. A simple program demonstrating the use of weak references is given in Figure 3. Statement (1) creates an object and bound it to the strong

```
(1) Object obj = new Object();
(2) WeakReference wr = new WeakReference(obj);
(3) obj = null;
(4) Object obj2 = wr.get();
(5) if (obj2 == null) {
    ...
(6) } else {
    ...
}
```

Figure 3: Using Weak References in Java

reference `obj`. The object is then bound to the weak reference `wr` in statement (2). Statement (3) set `obj` to `null`, causing the object to lose its strong reference. At this point, garbage collection might or might not occur, affecting the rest of the computation. If garbage collection has occurred at this point, the object would not exist in memory anymore, since it is only pointed by a weak

reference. Thus, getting the object in statement (4) would return a `null` value, which will make the if statement in (5) evaluate to true. On the other hand, if garbage collection has not occurred between statement (3) and (4), statement (4) would return the real object, which is still in memory, causing the if statement in (5) to evaluate to false. It is clear from the example that in the presence of non-strong references different evaluation paths can be chosen depending on the times in which garbage collection occurs in the course of computation.

If weak references introduce so much nondeterminism to the computation, why use them at all? There are cases where using weak references is very useful. Consider, for example, writing a graphical application. The program makes use of a library of graphical widgets that, for the sake of this example, cannot be extended to include any user data. This might be very restrictive as the programmer might need to manage some application specific data along with every widget. One possible and very natural approach to work around this restriction is to maintain a hash table mapping every widget to a state object, which keeps the application specific data. Figure 4 gives a code snippet to support this example. In statement (1) the widget's specific data, `myWidgetState`,

```
(1)    ...
        hashMap.put(widget, myWidgetState);
        ...
(2)    hashMap.get(widget);
        ...
```

Figure 4: Motivating Weak References in Java

is attached to the widget in the hash table. Later when there is a need to use the state kept for a certain widget, we use the code in (2) to retrieve it from the hash table. Since there is a state data attached to every widget, it makes sense to create the hash table entry when the widget is created and remove it when the widget is no longer used. Moreover, failing to remove the entry when the widget is disposable will prevent the garbage collector from collecting the widget, even though it is not useful anymore, thus introducing a form of memory leak. However, knowing the program point where the widget is disposable might not be so trivial. In fact, imposing this constraint on the programmer would diminish the benefit of automatic memory management provided by Java. There is a solution to this problem and it involves weak references. The class `WeakHashMap` in Java implements a hash table, where the keys are wrapped by weak references. By making the hash table in our example a `WeakHashMap` we solve the problem discussed above. Now, the hash table by itself is not enough to keep a widget in memory and whenever the widget is disposed by the garbage collector the corresponding entry in the hash table becomes unusable. Java also includes methods to remove the entry automatically when the widget is garbage collected. This, however, is not covered here.

Another useful type of references is soft references. Soft references are similar to weak references with one major difference. The garbage collector will try to keep a softly referred object as long as possible, whereas it will get rid of a weakly referred object on the first chance it has. Softly referred objects are usually kept in memory until the application is running out of memory. In this sense, weak references are eagerly collected while soft references are lazily collected. As an example justifying soft references, consider an application to manage documents that might have images embedded in them. As the user browses through the document, the software loads images from disk to memory and flushes them back to disk when necessary. On the one hand, we would like to keep the images in memory so the user does not experience long delays in viewing the documents. On

the other hand, keeping all images in memory might not be feasible due to memory constraints. The Cache Management design pattern, described in [5], addresses this problem. In Cache Management, a cache object holds a fixed number of images in memory. When a request to load an image is being made, the cache makes the image available in memory, at the expense of possibly discarding another image from memory. If the image was already in memory, i.e., the cache has previously loaded the image, then there is nothing to be done. An algorithm in the cache decides which images should stay in memory, by making an educated guess of the images that are most likely to be used in the future. Note that in the Cache Management design pattern the programmer is in charge of coding the decision of discarding an image.

A slightly different approach to Cache Management is the notion of Memory Sensitive Cache (MSC), described in [8]. MSC is a variable sized cache where the decision to remove objects is done automatically by the garbage collector. That is, every image held in MSC is wrapped by a soft reference. When the application is using an image, i.e., the user is currently viewing the image, a strong reference points to the image, preventing it from being garbage collected. On the other hand, an image in the cache that is not used by the application can be chosen by the garbage collector for disposal. Recall the garbage collector usually removes softly referred objects when the application is low in memory. Consequentially, MSC provides a cache that is both automatically managed and sensitive to the amount of memory the application is consuming at every point of the computation.

Figure 5 sketches a MSC implementation addressing the above problem of image caching.

```
class ImageMSC{
    private HashMap map;

    public ImageMSC() { ... }

    public getImage(name){
        SoftReference sr = (SoftReference) map.get(name);
        if(sr == null || sr.get() == null){
            Image img = loadImageFromDisk(name);
            map.put(name, new SoftReference(img));
            return img;
        }else{
            return sr.get();
        }
    }
}
```

Figure 5: A Java Example to Motivate Soft References

The class `ImageMSC` has a hash table as an underlying data structure. The main functionality provided in this class is the function `getImage`, which takes care of loading images to memory when necessary. In the function, we first check to see if the cache already has the required image in memory. If the image is in memory (the else branch) we return it to the caller. Otherwise (the then branch), we load the image from the disk and put it in the cache, wrapped in a soft reference. We, then, return the image to the caller. This is all the logic required to be written on the programmer's side. Remove unneeded images from the cache is done by the garbage collector whenever memory is low.

## 4 Reachability of Heap Objects and Garbage Collection

The garbage collector operates occasionally in the middle of program evaluation. When it operates, the garbage collector computes a set of heap objects that are no longer needed by the program and removes them from the heap. There are many algorithms to identify and choose a set of disposable objects. This paper does not cover different garbage collection techniques. Instead, we discuss garbage collection in a more abstract level and examine different classes of disposable objects introduced by different types of references.

How should we define garbage? An object can be determined garbage at a certain timepoint if it is never used in program evaluation thereafter. With this definition however, the problem of identifying garbage is undecidable since it can be reduced to the halting problem [2]. Instead, garbage collectors rely on the more tractable notion of object reachability. According to this approach, a heap object is unneeded in a program if there is no way of accessing the object from the program in its current configuration. Note that if an object is deemed garbage according to the latter definition it is deemed garbage according to the former, but not the other way. The information below, as well as the examples in the rest of the section, are inspired by [10].

In a language with only strong references objects can either be reachable from the program or not. 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*. Notice an object can have more than one reachability path to it as well as have no reachability paths at all. In the latter, the object is deemed garbage and can be collected by the garbage collector right away. Figure 6 give an example demonstrating the reachability of six different heap objects. Objects A, B, C, D, E, F and G are all heap allocated objects. Objects A,

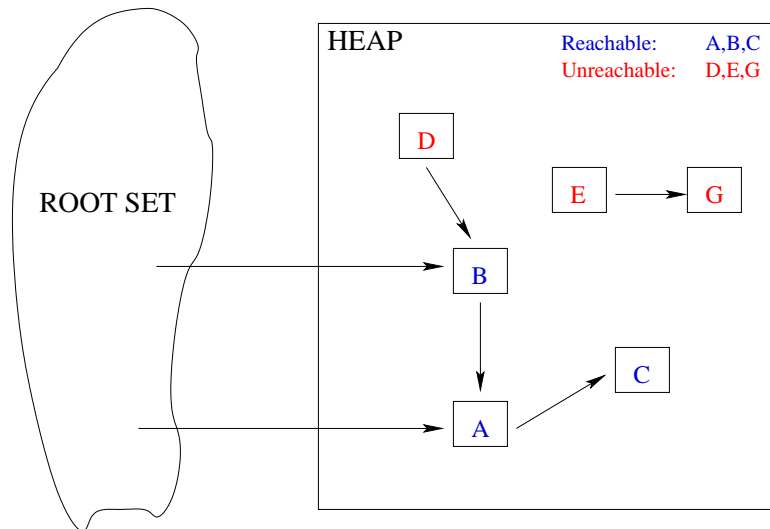


Figure 6: The Reachability of Heap Objects

B and C are reachable since there is at least one reachability path from the root set to each one of them. Objects D, E and G are garbage since they do not have any reachability path. Note that object A has two different reachability paths, one of which going through object B. Similarly, object C has two reachability paths as well.

Non-strong references introduce additional complexity to the notion of reachability and thus add complexity to garbage collection. With non-strong references, there are several classes of reachability forming a linear order with respect to their strength. This order from strongest to weakest is strongly reachable, softly reachable, weakly reachable, phantomly reachable and unreachable. Section 3 gives a simplistic view on the behavior of the garbage collector in the presence of non-strong references. In practice, a reachability path to an object might include different types of references. Consider an object that is referred by a strong reference, but has non-strong references preceding this reference on the reachability path. How should the garbage collector treat this object? Is the object strongly reachable or is it weakly reachable? Assuming this is the only reachability path of the object, the object is not strongly reachable.

A strongly reachable object is an object that has at least one reachability path, where all the references in the path are strong. Stated differently, a strongly reachable object has at least one strong reachability path, where strong reachability paths include strong references only. 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.* The example in Figure 7 clarifies the above-mentioned. Objects A, B, C, D and E are heap allocated objects. There is one

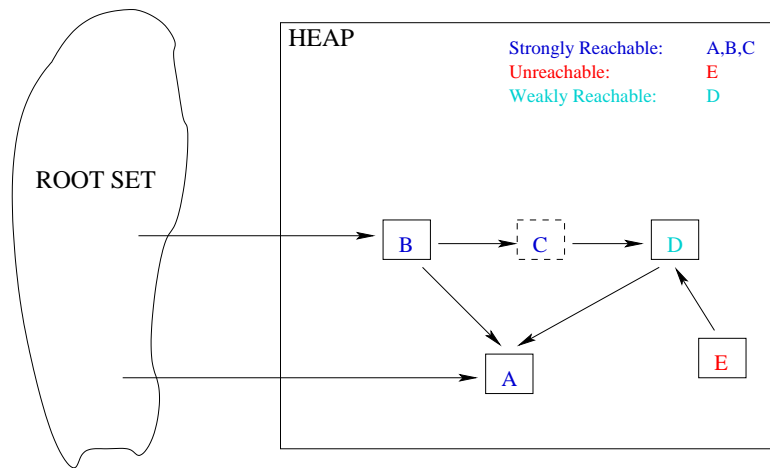


Figure 7: Reachability Paths with Non-Strong References

weak reference in the example, which is object C, and it points to object D. In the example, Objects A, B and C are strongly reachable, object D is weakly reachable and object E is unreachable. Object A has three reachability paths, two of which are strong and one is weak. Object D is only reachable by a path that goes through object C, a weak reference, and thus is weakly reachable. Object E is unreachable since it has no reachability paths, i.e., there is no way of accessing object E from the program's root set.

In conclusion, non-strong references introduce different levels of object reachability forming a linear order with respect to their strengths. An object's reachability is determined by the weakest

reference on its strongest reachability path, where unreachable objects have no reachability paths at all. Garbage collectors apply different policies to different objects depending on their reachability.

## 5 Reachability Based Garbage Collection

Garbage collection in  $\lambda_{weak}$  [2] is based on the free-variable approach. This approach, inspired by garbage collection in  $\lambda_{gc}$  [7], though simple and compact has certain limitations. In particular, there are reasonable garbage collection evaluations that can not be modelled by  $\lambda_{weak}$  and  $\lambda_{gc}$ . This section offers a different modelling of garbage collection based on the notion of reachability introduced in Section 4. This modelling is made formal into a calculus called  $\lambda_{weak1}$ . The new calculus is then compared to  $\lambda_{weak}$  in terms of compactness and completeness. We show that  $\lambda_{weak1}$  is able to model garbage collection evaluations that cannot be modelled in  $\lambda_{weak}$ . Moreover, the reachability approach to garbage collection makes the handling of weak references explicit, facilitating the modelling of multiple levels of weak references later in Section 6.

$\lambda_{weak}$  adopts the free-variable approach to garbage collection. This approach is based on the approach taken in  $\lambda_{gc}$  and dates back to Felleisen 92 [3]. According to the specification of garbage collection in  $\lambda_{gc}$ , the heap is partitioned into two subheaps nondeterministically, where one subheap is declared garbage if there are no references to it in the other subheap nor in the current evaluation expression. Similarly, the garbage collector in  $\lambda_{weak}$  partitions the heap into two subheaps nondeterministically, and declares one garbage if there are no *strong* references to it in the other subheap nor in the current evaluation expression.

The advantage of the free-variable approach is its simplicity and compactness. However, there are reasonable garbage collection evaluations that cannot be modelled with this approach. As an example, consider the program  $P$  in Figure 8. The bindings  $y \mapsto z$  and  $z \mapsto 1$  are both garbage

$$\begin{array}{c}
 P = \text{letrec } \{x \mapsto 1, y \mapsto z, z \mapsto 1\} \text{ in } x \\
 \xrightarrow{\text{garb}} \text{letrec } \{x \mapsto 1, y \mapsto z\} \text{ in } x
 \end{array}$$

Figure 8: The Limitation of Free-Variable Garbage Collection

in the program since  $x \mapsto 1$  is the only binding needed in the rest of the evaluation. Given this program, it may be plausible for the garbage collector to collect the binding  $z \mapsto 1$  while leaving  $y \mapsto z$  on the heap. This may be justified by considerations of time efficiency. However, this garbage collection action cannot be modelled in  $\lambda_{weak}$  nor in  $\lambda_{gc}$ . By partitioning the heap in  $P$  into  $H_1 = \{x \mapsto 1, y \mapsto z\}$  and  $H_2 = \{z \mapsto 1\}$ , the free-variable condition does not hold. That is,  $H_2$  cannot be declared as garbage as there is a strong reference from  $H_1$  to  $z$ . Since this example does not contain weak references, it demonstrates the limitation in both calculi. A similar example that makes use of weak references can easily be composed.

For this reason, we formulate  $\lambda_{weak1}$ , a variation of  $\lambda_{weak}$  which models garbage collection using the notion of reachability. With a reachability based garbage collection,  $\lambda_{weak1}$  is shown to have greater expressive power than  $\lambda_{weak}$  and the ability to model garbage collection evaluations that cannot be modelled in a free-variable based calculus. In order to formulate  $\lambda_{weak1}$  we first need to formalize the notion of reachability introduced in Section 4.

## 5.1 Formalizing Object Reachability

In essence, a reachable object is a heap allocated object, heap value in  $\lambda_{weak}$ , that can be accessed from the program variables. Unreachable objects are considered garbage and therefore are safe to garbage collect. A simple notion of reachability exists when the language contains strong references only. In the presence of non-strong references, several levels of reachability exist each having a different reachability strength. Reachability of an object is determined by finding a path from the *root set* of the expression to the object.

**Definition 5.1 (Root Set).** *The root set of an expression  $e$  is  $FV(e)$ .*

An object is determined reachable if there is a variable in the root set of the expression pointing to it. Moreover, an object can be indirectly reachable. That is, an object is reachable if there is another reachable object pointing to it. Such a chain of references from the root set to a heap object is called *reachability path*. A reachability path is a path of one or more bindings starting from the root set, going through bindings on the heap and ending at the object. Definition 5.2 formalizes the notion of reachability path.

**Definition 5.2 (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.  $H(x_n) = hv$ .

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. Notice the explicit restriction on the path variables to be unique. By doing so, we exclude paths that contains cycles. The following example demonstrates the notion of reachability path.

**Example 5.3 (Reachability Path).** Let  $(\text{letrec } H \text{ in } e)$  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 for a 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.

Note that multiple objects of the same value on the heap are considered different heap values. For instance, there are two different objects of the value 1 in the heap  $\{x \mapsto 1, y \mapsto 1\}$ . The reachability of one object is totally independent from the reachability of the other.

In the presence of non-strong references multiple levels of reachability exist. An object is strongly reachable if it has at least one reachability path consisting of only strong references.

**Definition 5.4 (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  $H(x_i) \neq \text{weak } y$ .*

For the sake of formulating  $\lambda_{weak1}$  the definition above suffice. However, in Section 6 we introduce a calculus containing soft references as well as weak ones. For that calculus, we need to define soft and weak reachabilities.

**Definition 5.5 (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 5.6 (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.*

The above definitions assume two levels of non-strong references. However, this can be generalized to any number of non-strong reference levels as long as their strengths form a linear order. As an example, phantom reachability could be added as another level of reachability between weak reachability and unreachability.

## 5.2 $\lambda_{weak}$ Reformulated Using a Reachability Based Garbage Collection

Using the definition of strongly reachable objects we formulate a variant of  $\lambda_{weak}$  named  $\lambda_{weak1}$ .  $\lambda_{weak1}$  has the same syntax and evaluation contexts as  $\lambda_{weak}$ . However, it is slightly different in the rewrite rules. More specifically,  $\lambda_{weak1}$  redefines the (garb) rule by using Definition 5.4. The definitions of softly reachable and weakly reachable objects are not used in this section and only come into play in the next section, where we introduce soft references into the calculus. The rewrite rules for  $\lambda_{weak1}$  are given in Figure 9.

The only rule in  $\lambda_{weak1}$  that is different from its counterpart in  $\lambda_{weak}$  is the auxiliary rule (gc). This rule is different in two aspects. The first,  $hv$  is determined to be garbage based on the reachability approach offered in this section as opposed to the original free-variable approach. The second difference is in the choice of objects to be collected. In  $\lambda_{weak}$ , a set of bindings  $H_2$  is chosen nondeterministically, where in  $\lambda_{weak1}$  a single binding is chosen each time (gc) is called. We discuss these two aspects below.

$\lambda_{weak1}$  makes use of the reachability based approach in order to determine a heap value is garbage. With this approach,  $\lambda_{weak1}$  is able to cover more garbage collection evaluations than  $\lambda_{weak}$  and  $\lambda_{gc}$ . Consider again the example in Figure 8, which demonstrates the limitation of garbage collection modelling in  $\lambda_{weak}$  and  $\lambda_{gc}$ . Can this evaluation be made in  $\lambda_{weak1}$ ? The bindings  $y \mapsto z$  and  $z \mapsto 1$  are both garbage in the program. In the evaluation, we want to remove  $z \mapsto 1$  while keeping  $y \mapsto z$  on the heap. The object 1 bound to  $z$  is not strongly reachable in the program since there is no strongly reachable path from the root set to it. In fact there is no reachable path at all starting from the root set,  $x$ , and ending in the object. Consequentially, this garbage collection action can be modelled in  $\lambda_{weak1}$ .

This shows that there are garbage collection evaluations that can be made in  $\lambda_{weak1}$  and not in  $\lambda_{weak}$ . How about the converse? Are there evaluations that can be made in  $\lambda_{weak}$  and not in  $\lambda_{weak1}$ ? Proposition 5.7 proves the expressive power of  $\lambda_{weak1}$  is no less than the one of  $\lambda_{weak}$ . That is, every evaluation that can be made in  $\lambda_{weak}$  can also be made in  $\lambda_{weak1}$ .

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
( $\pi_i$ )	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$ 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)	$\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = d \end{cases}$
(garb)	$\text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$ provided $\text{letrec } H \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H'' \text{ in } e$ and $\text{letrec } H'' \text{ in } e \downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e$
Auxiliary:	
(gc)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H \text{ in } e$ provided $hv$ is not strongly reachable in $(\text{letrec } H \text{ in } e)$
(weak-gc)	$\text{letrec } H \uplus \{x \mapsto \text{weak } y\} \text{ in } e \xrightarrow{\text{weak-gc}} \text{letrec } H \uplus \{x \mapsto d\} \text{ in } e$ provided $y \notin \text{Dom}(H)$

Figure 9: The Rewrite rules of  $\lambda_{\text{weak}1}$

**Proposition 5.7 (Expressive Power of  $\lambda_{\text{weak}1}$  with respect to  $\lambda_{\text{weak}}$ ).** *Let  $P$  and  $P'$  be programs in  $\lambda_{\text{weak}}$ . If  $P \rightarrow_{\lambda_{\text{weak}}}^* P'$  then  $P \rightarrow_{\lambda_{\text{weak}1}}^* P'$  (the arrows' subscript annotation corresponds to the calculus under which the evaluation is being carried out). In words, the proposition states that if an evaluation path is taken from  $P$  to  $P'$  in  $\lambda_{\text{weak}}$ , then there is an evaluation path in  $\lambda_{\text{weak}1}$  going from  $P$  to  $P'$ .*

The proof is given in Appendix A.

This proves the reachability based approach to garbage collection enables us to model more garbage collection actions than the free-variable approach. However, does the expressiveness come in the price of compactness and simplicity? In other words, how do the two calculi compare in terms of compactness? It seems  $\lambda_{\text{weak}1}$  uses more definitions than  $\lambda_{\text{weak}}$ . It uses the *FV* definition, which is the only mechanism used in  $\lambda_{\text{weak}}$ , and in addition it uses the definition for strongly reachable objects. It seems  $\lambda_{\text{weak}1}$  is less compact than  $\lambda_{\text{weak}}$ . However, the *FV* definition used in  $\lambda_{\text{weak}1}$  is a restricted form of the one used in  $\lambda_{\text{weak}}$ . In  $\lambda_{\text{weak}1}$ , *FV* is not computed on programs as it is in  $\lambda_{\text{weak}}$ . Instead, *FV* is computed on expressions only. Given all this, it is hard to state which of the calculi is simpler though  $\lambda_{\text{weak}1}$  seems to be slightly less compact.

The auxiliary rule (gc) in  $\lambda_{\text{weak}1}$  is also different in the way it chooses bindings for collection. In  $\lambda_{\text{weak}}$  a subset  $H_2$ , whose size is nondeterministic, is chosen for collection. On the other hand, in  $\lambda_{\text{weak}1}$  a single element,  $\{x \mapsto hv\}$ , is always chosen. Obviously, with a rule that chooses a single element we can express a rule that chooses a set of bindings. This can be achieved by applying the rule as many times as there are elements in the set. Why then not use a single element rule in  $\lambda_{\text{weak}}$ ? The reason is that if this was the case in  $\lambda_{\text{weak}}$  garbage cycles would not be collected. Recall that the free-variable approach checks if the candidate for collection is attached to the rest of the program, where the heap is included. With this in mind, a cycle of garbage will never be able to

be broken and collected since the collection candidate will always have a reference in the rest of the heap pointing to it. As an example, consider the program

$$\text{letrec } \{x \mapsto 1, y \mapsto z, z \mapsto y\} \text{ in } x$$

With a (gc) rule that collects a single element in  $\lambda_{weak}$ , neither  $y \mapsto z$  nor  $z \mapsto y$  can be collected. Since  $y$  points to  $z$  and vice versa, it is impossible to break the cycle in a free-variable garbage collection approach. Due to this reason, the (gc) auxiliary rule removes a sub of bindings,  $H_{\underline{b}}$ , every time it is called. However, the garbage cycle problem does not exist in  $\lambda_{weak1}$ . With a reachability based approach, both  $y \mapsto z$  and  $z \mapsto y$  are not reachable from the root set of the program,  $x$ . The fact that they point each other is independent from the fact that they are unreachable from the root set of the program. Therefore, they can both be collected separately.

Since  $\lambda_{weak1}$  has more expressive power than  $\lambda_{weak}$ , it is natural to try to define a generalization relationship between the two. That is, we would like to say  $\lambda_{weak}$  is a special case of  $\lambda_{weak1}$  or that it can be achieved by somehow instantiating  $\lambda_{weak1}$ . However, since the approach to garbage collection is essentially different such a connection cannot be trivially made. In order to make such a connection, we need to limit garbage collection in  $\lambda_{weak1}$  in a way which will be awkward and nontrivial.

To conclude, this section presents  $\lambda_{weak1}$ , a reachability based calculus based on  $\lambda_{weak}$ . The free-variable approach to garbage collection used in  $\lambda_{weak}$  has certain limitations. With a reachability based approach more cases of garbage collection can be modelled. Moreover, the garbage collection rule can be made simpler by collecting a single element each time the garbage collection is called. A single element collection rule cannot be used in  $\lambda_{weak}$  due to the problem of garbage cycle collection. It is hard to state if the additional expressive power comes with the price of complexity. However,  $\lambda_{weak1}$  seems to be slightly less compact than  $\lambda_{weak}$ . The examples shown in this section can demonstrate the limitations in  $\lambda_{gc}$  as well. A reachability based  $\lambda_{gc}$  variant can be similarly composed to provide additional expressive power to  $\lambda_{gc}$ . As a final note, the reachability based approach to garbage collection handles weak references more explicitly. This approach facilitates composing a calculus with multiple non-strong references as will become apparent in the next section.

## 6 Multiple Non-Strong References Along with System Resource Considerations

Many garbage collection techniques exist. They all involve various considerations in order to achieve the goal of automatic memory management. The primary goal of garbage collectors is to optimize memory by monitoring usage and freeing unused objects. However, a garbage collector has to be aware of time constraints. That is, if a program is not limited by space but is under tight time restrictions, it may not be reasonable to collect as much garbage as possible. These considerations become even more complex in the presence of non-strong references. Non-strong references usually point to memory that is still used by the program. Therefore garbage collectors have to collect memory with extra caution and have to do so in a timely manner.

In this section we define a framework, based on the reachability approach, which facilitates the modelling of time and space decisions made by garbage collectors in the context of multiple levels of

non-strong references. More specifically, we consider different garbage collection strategies along two dimensions.

**Eager vs. lazy garbage collection:** A garbage collector could opt to eagerly collect all the garbage available every time it kicks in. Conversely, it could choose to defer collection as much as possible. This approach is called lazy collection.

**Time vs. space constraints:** That is, a garbage collector could aim at optimizing time or at optimizing space and these will result in different garbage collection behaviors.

Obviously, the two dimensions are not orthogonal and their interaction is discussed below.

Two calculi are presented and discussed in this section. The first is  $\lambda_{weak2}$ , a calculus based on  $\lambda_{weak1}$ . In  $\lambda_{weak2}$ , we introduce soft references into the calculus in addition to weak references. Similar to the way the references are handled in Java (see Section 3), soft references are given the semantics of lazy collection, whereas weak references are given the semantics of eager collection. Of course, lazy collection taken to the extreme is no collection at all. Therefore, the collection of soft references is contingent on the amount of memory available in the system. Analogously, we should have connected the collection of weak references with time constraints. However, due to complexity considerations, we leave the collection of weak references unconditioned.

Instead, we introduce time considerations into  $\lambda_{gc'}$ , a calculus based on  $\lambda_{gc}$ .  $\lambda_{gc'}$  has only strong references. In this context we consider eager vs. lazy collection under both time and space constraints.

In Section 5, we showed the reachability-based approach can model more garbage collection cases than the free-variable approach. Moreover, our approach appears to be useful in two more aspects. First, the garbage collection rule ((gc) in the case of  $\lambda_{weak2}$ ) removes exactly one element each time it is invoked. This provides finer control over garbage collection as we shall see below. The second aspect has to do with the way it computes reachability. By considering the paths from the expression to the objects, the reachability approach can easily support multiple levels of non-strong references. Although  $\lambda_{weak2}$  includes only two types of non-strong references, the more general case can also be constructed.

## 6.1 Multiple Levels of Non-Strong References in $\lambda_{weak2}$

We construct  $\lambda_{weak2}$ , given in Figure 10, by adding soft references to  $\lambda_{weak1}$ . Soft references are added to the syntax, to the evaluation contexts and to the (ifdead) rule similar to the way weak references appear. We focus our attention on the new (garb) rule, where most of the difference lies.

Objects in  $\lambda_{weak2}$  can have one out of four levels of reachability. From strongest to weakest, these 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 10. The auxiliary rules are defined in terms of the reachability definitions of Section 5.1. 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) takes care of collecting 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  $\bar{d}$ . This is done by the rule (ref2d). The

<b>Programs:</b>			
(variables)	$w, x, y, z \in \text{Var}$		
(integers)	$i \in \text{Int}$	$::=$	$\dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$
(expressions)	$e \in \text{Exp}$	$::=$	$x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_i e \mid \pi_2 e \mid \lambda x. e \mid e_1 e_2 \mid$ $\text{weak } e \mid \text{ifdead } e_1 e_2 e_3 \mid \text{soft } e$
(heap values)	$h_v \in \text{Hval}$	$::=$	$i \mid \langle x_1, x_2 \rangle \mid \lambda x. e \mid \text{weak } x \mid \text{d} \mid \text{soft } x$
(heaps)	$H \in \text{Var} \xrightarrow{\text{fin}} \text{Hval}$		
(programs)	$P \in \text{Prog}$	$::=$	$\text{letrec } H \text{ in } e$
(answers)	$A \in \text{Ans}$	$::=$	$\text{letrec } H \text{ in } x$
<b>Evaluation Contexts and Instruction Expressions:</b>			
(contexts)	$E \in \text{Ctx}$	$::=$	$[] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E \mid \text{weak } E \mid$ $\text{ifdead } E e_1 e_2 \mid \text{soft } E$
(instruction)	$I \in \text{Instr}$	$::=$	$h_v \mid \pi_i x \mid x y \mid \text{ifdead } x e_1 e_2$
<b>Rewrite Rules:</b>			
(alloc)	$\text{letrec } H \text{ in } E[h_v] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } E[x]$		where $x$ is a fresh variable
( $\pi_i$ )	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$		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 $\text{letrec } H \text{ in } e \Downarrow_{\text{gc-soft, gc-weak, gc-unr, ref2d}} \text{letrec } H' \text{ in } e$
<b>Auxiliary:</b>			
(gc-soft)	$\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e \xrightarrow{\text{gc-soft}} \text{letrec } H \text{ in } e$		provided $h_v$ is softly reachable in $(\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e)$ and $\text{isMemCond}(H \uplus \{x \mapsto h_v\})$
(gc-weak)	$\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e \xrightarrow{\text{gc-weak}} \text{letrec } H \text{ in } e$		provided $h_v$ is weakly reachable in $(\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e)$
(gc-unr)	$\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e \xrightarrow{\text{gc-unr}} \text{letrec } H \text{ in } e$		provided $h_v$ is unreachable in $(\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e)$
(ref2d)	$\text{letrec } H \uplus \{x \mapsto h_v\} \text{ in } e \xrightarrow{\text{ref2d}} \text{letrec } H \uplus \{x \mapsto \text{d}\} \text{ in } e$		provided $h_v$ is either <b>soft</b> $y$ or <b>weak</b> $y$ and $y \notin \text{Dom}(H)$

Figure 10: The Syntax and Operational Semantics of  $\lambda_{\text{weak2}}$

shorthand notation  $\Downarrow_{r1,r2}$  used in (garb) is defined below.

$$\begin{aligned} \text{letrec } H \text{ in } e \Downarrow_{r1,r2} \text{letrec } H' \text{ in } e' &\equiv \text{for every } H'' \text{ and } e'' \\ &\text{letrec } H \text{ in } e \Downarrow_{r1} \text{letrec } H'' \text{ in } e'' \text{ and} \\ &\text{letrec } H'' \text{ in } e'' \Downarrow_{r2} \text{letrec } H' \text{ in } e' \end{aligned}$$

In  $\lambda_{weak2}$  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 model template, where different garbage collection strategies can be evaluated in the context of multiple levels of reachability. The concrete policies for each type of reachability is given below.

As one would expect, strongly reachable objects are not collected at all in  $\lambda_{weak2}$ . The garbage collector completely ignores this type of objects. On the other hand, unreachable objects are collected immediately. Every time the garbage collector kicks in, it collects all unreachable objects. This is done by the arrow  $\Downarrow_{gc-unr}$  in the (garb) rule. Softly reachable objects and weakly reachable objects are given a collection semantics similar to the one in Java (see Section 3).

Softly reachable objects are collected lazily. Generally speaking, lazy collection taken to the extreme is just no collection at all. Therefore, the collection of softly reachable objects in  $\lambda_{weak2}$  is contingent on the amount of memory available. This is achieved by using the `isMemCond` predicate in (gc-soft). `isMemCond(H)` is simply defined by counting the number of bindings in  $H$  and returning true only if this number exceeds a certain threshold. This makes softly reachable objects ones that are “nice to have” as long as memory permits.

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 get softly reachable objects that are collected eagerly.

Weakly reachable objects are collected eagerly. That is, every time the garbage collector is invoked all weakly reachable objects are collected. Analogous to the way we tied lazy collection with memory constraints, it might be appropriate to connect eager collection with time constraints. With this approach, weakly reachable objects would be collected eagerly as long as time permits. However, we chose not to introduce time constraints into  $\lambda_{weak2}$  due to complexity considerations. Note that, as defined in  $\lambda_{weak2}$ , weakly reachable objects have the collection semantics of unreachable objects.

## 6.2 Time Vs. Space Considerations in $\lambda_{gc'}$

$\lambda_{gc}$  presented in [7] aims at modelling heap allocated objects and garbage collection thereof. It includes only strong references. We present  $\lambda_{gc'}$ , a reachability based variant of  $\lambda_{gc}$ . The only type of objects that is considered for removal in  $\lambda_{gc'}$  is unreachable object. In  $\lambda_{gc'}$ , as well as in  $\lambda_{gc}$ , garbage collection does not affect the result of program evaluation. However, there is non-determinism in the behavior of the garbage collector. In the rest of the section, we focus on garbage collection strategies in the context of  $\lambda_{gc'}$ . In particular, we consider different strategies along two dimensions. The first dimension is eager vs. lazy garbage collection. A garbage collector could opt to eagerly collect all the garbage available every time it kicks in. On the other hand, it could choose to defer collection as much as possible. This approach is called lazy collection. The second

dimension we consider is the tension between time and space constraints. That is, a garbage collector could aim at optimizing time or at optimizing space and these will result in different garbage collection behaviors. The interaction between these two dimensions is discussed below.

The syntax and operational semantics of  $\lambda_{gc'}$  is given in Figure 11. The only difference from  $\lambda_{gc}$  lies in the (garb) rule. The (garb) rule in  $\lambda_{gc'}$  removes a single object provided the object is unreachable as defined in Section 5.1.

<b>Programs:</b>			
(variables)	$w, x, y, z$	$\in$	$\text{Var}$
(integers)	$i$	$\in$	$\text{Int} \quad ::= \dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$
(expressions)	$e$	$\in$	$\text{Exp} \quad ::= x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \lambda x. e \mid e_1 e_2$
(heap values)	$hv$	$\in$	$\text{Hval} \quad ::= i \mid \langle x_1, x_2 \rangle \mid \lambda x. e$
(heaps)	$H$	$\in$	$\text{Var} \xrightarrow{\text{fin}} \text{Hval}$
(programs)	$P$	$\in$	$\text{Prog} \quad ::= \text{letrec } H \text{ in } e$
(answers)	$A$	$\in$	$\text{Ans} \quad ::= \text{letrec } H \text{ in } x$
<b>Evaluation Contexts and Instruction Expressions:</b>			
(contexts)	$E$	$\in$	$\text{Ctx} \quad ::= [] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E$
(instruction)	$I$	$\in$	$\text{Instr} \quad ::= hv \mid \pi_i x \mid x y$
<b>Rewrite Rules:</b>			
(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
( $\pi_i$ )	$\text{letrec } H \text{ in } E[\pi_i x]$	$\xrightarrow{\pi_i}$	$\text{letrec } H \text{ in } E[x_i]$ provided $H(x) = \langle x_1, x_2 \rangle$ and $i \in \{1, 2\}$
(app)	$\text{letrec } H \text{ in } E[x y]$	$\xrightarrow{\text{app}}$	$\text{letrec } H \uplus \{z \mapsto H(y)\} \text{ in } E[e]$ provided $H(x) = \lambda z. e$
(garb)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e$	$\xrightarrow{\text{garb}}$	$\text{letrec } H \text{ in } e$ provided $hv$ is unreachable in $(\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)$

Figure 11: The Syntax and Operational Semantics of  $\lambda_{gc'}$

The first garbage collection strategy we consider is lazy collection. Since garbage collection does not affect the program's result, a complete lazy collection will not collect objects at all. This can be achieved by simply taking the (garb) rule out of the calculus. This approach aims at optimizing time while completely neglecting any space consideration.

The complete opposite of the lazy approach is eager collection. In this approach, the garbage collector collects all unreachable objects whenever it kicks in. To model an eager collection we replace the (garb) rule in  $\lambda_{gc'}$  by the following.

$$\begin{aligned}
 (\text{garb}) \quad & \text{letrec } H \text{ in } e \Downarrow_{\text{garb-aux}} \text{letrec } H' \text{ in } e \\
 (\text{garb-aux}) \quad & \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{garb-aux}} \text{letrec } H \text{ in } e \\
 & \text{provided } hv \text{ is unreachable in } (\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)
 \end{aligned}$$

This way, whenever the (garb) rule is evaluated it collects all unreachable objects eagerly. As opposed to the lazy approach, the eager approach aims at optimizing space while completely neglecting time consideration.

Both of the above methods ignore one of the aspects of time and space. To remedy this, we offer a lazy approach which is also space aware. This approach optimizes time by not collecting objects if possible. However, when memory becomes scarce it starts collecting objects as much as needed. To this end, we add a memory condition similar to the one used in  $\lambda_{weak2}$ . The following (garb) rule models this approach.

$$\begin{array}{l}
(\text{garb}) \quad \text{letrec } H \text{ in } e \Downarrow_{\text{garb-aux}} \text{letrec } H' \text{ in } e \\
(\text{garb-aux}) \quad \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{garb-aux}} \text{letrec } H \text{ in } e \\
\text{provided } hv \text{ is unreachable in } (\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e) \\
\text{and } |H \uplus \{x \mapsto hv\}| > H_{max}
\end{array}$$

With this (garb) rule, whenever the garbage collector kicks in it collects objects from the heap as long as the number of bindings on the heap exceeds a certain threshold,  $H_{max}$ . Note that the objects removed are still chosen nondeterministically.

Analogous to the space-aware lazy approach, we offer a time-aware eager approach. This approach optimizes space by collecting as much objects as possible. However, we restrict the number of objects the garbage collector collects when it operates by putting an upper bound on the time each cycle may take. To achieve this, we first make the notion of time precise. For simplicity, we define a single time unit to be one step of evaluation in any of the rewrite rules except for (garb). Garbage collection takes as many time units as the number of objects collected. That is, collecting a single object from the heap takes a single time unit. Note that collecting a single object involves computing all reachability paths for which a single time unit might seem out of proportion. However, we make the simplifying assumption that this information can be computed once for a group of objects so that it is not very time costly for each one of them.

With the definition of time, we construct the (garb) rule in the following way.

$$\begin{array}{l}
(\text{garb}) \quad \text{letrec } H \text{ in } e \Downarrow_{\text{garb-aux}}^{t_{max}} \text{letrec } H' \text{ in } e \\
(\text{garb-aux}) \quad \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{garb-aux}} \text{letrec } H \text{ in } e \\
\text{provided } hv \text{ is unreachable in } (\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)
\end{array}$$

Where  $\Downarrow_{\text{garb-aux}}^{t_{max}}$  means evaluating  $\xrightarrow{\text{garb-aux}}$  as much as possible and up to  $t_{max}$  times. This way we have a time bound eager garbage collector.

To conclude,  $\lambda_{gc'}$  can be used as a framework to model different garbage collection strategies under time and space consideration. In the context of  $\lambda_{gc'}$ , two computation strategies are presented. In the first, the garbage collector eagerly collects all objects that are considered garbage at the time of collection. The second approach suggests a lazy collection of garbage. To make the model more realistic, time and space considerations are added. This results in two more advanced garbage collection models. The first model corresponds to a time-aware eager garbage collector. This garbage collector optimizes space while respecting a given time restriction. Conversely, the second model corresponds to a space-aware lazy garbage collector. This garbage collector optimizes time while respecting a given space restriction.

## 7 Finalizers in Java

Often times it is considered good programming practice to attach termination actions to the event of object removal. In Java, non-memory related termination work can be done on a per-object basis

in a mechanism called *finalizer*. However, finalizers have a nondeterministic behavior which makes programs that rely on them unwieldy. This section gives an introduction to Java finalizers.

Consider a class that implements a file abstraction. Creating objects of this class corresponds to creating a file in the system. Similarly, object removal corresponds to closing the file. In C++, the Resource Acquisition Is Initialization (RAII) pattern, described by Stroustrup in [11], is a natural approach to implement this class. Following RAII, the object's lifecycle should correspond exactly to the lifecycle of the file handler. That is, the file should be opened in the object constructor and be closed in the object's destructor. This way, the moment the programmer explicitly or implicitly deletes the object the language automatically takes care of closing the file. Hence, programmers using this class will not run into the risk of forgetting to close the file. Such reliance on C++ destructors to manage resources is encouraged in [11] since C++ destructors have a deterministic and expected behavior.

Finalizers are the Java way to attach automatic termination actions to objects. In Java, every object has a `finalize` method. `finalize` is defined in the `Object` class, but can be overridden by subclasses to provide customized finalization behavior. There is however a problem about using Java finalizers as an automatic termination mechanism since the removal of objects in Java is nondeterministic. That is, there is no guarantee finalizers will be called in a timely manner and thus no guarantee the file in our example will be closed when it needs to.

The Java Programming Language Specification [4] does not specify how soon a finalizer will be invoked. The only guarantee is that the finalizer will be invoked before the storage for the object is reclaimed. It is not impossible then for a finalizer to not be invoked at all since there is nothing in the language specification saying an unused object should be collected.

In his title "Effective Java – Programming Language Guide" [1], Bloch recommends avoiding the use of finalizers in Java entirely. He claims they are unpredictable, often dangerous and generally unnecessary. There are two exceptions given by Bloch to the rule. The first is to use finalizers as a fallback mechanism to object termination in case the programmer forgets to do so. Java programmers should write and use explicit functions to release non-memory resources. However finalizers should be written to do release work only as a "safety net" in case the programmer forgets to use the explicit termination function provided. The second exception to the rule, is to use finalizers when using native objects in Java. The garbage collector in Java will not release memory taken by a native object and since it is not a critical task, using a Java finalizer to do that is suitable.

In addition to the above-mentioned problems, finalizers in Java can be abused with detrimental consequences. As described in [12], a Java programmer has the ability to resurrect an object that was declared dead by the garbage collector. When an object is deemed dead by the garbage collector its finalizer method is called just before the memory is reclaimed. In practice, the object's finalizer can contain code to create a new strong reference to the object in some data structure outside the object. After running this finalizer, the garbage collector would not be able to remove the object from memory as there are new valid references to it. In essence, the object has been brought back to life. The code snippet in Figure 12 demonstrates this problem. Class `Immortal` redefines the `finalize` method. In the method, a new reference to `this` is created in `list`, a static list whose lifetime, for the sake of the example, lasts to program termination. When the garbage collector decides to remove an object of class `Immortal` it has to call its finalizer. However, by doing so it introduces a new reference to the object in `list`. After running the finalizer, the garbage collector is unable to remove the object since `list` has a valid reference to it.

To conclude, Java finalizers provide a way to attach termination actions to the event of object

```

public class Immortal {
    public static List list = new ArrayList();

    public void finalize()
    {
        list.add(this);
    }
}

```

Figure 12: Finalizer Abuse in Java

removal. Java finalizers should not be confused with their C++ deterministic counterparts, the destructors. Finalizers are invoked by the garbage collector nondeterministically and thus should not be used to perform timely tasks as file closing. In more extreme cases, finalizer can be abused to resurrect and object deemed dead by the garbage collector just before object removal. Given all that, programmers should use finalizer with extreme caution as programs relying on finalizers tend to be unwieldy and hard to debug.

## 8 Modelling Finalizers

A finalizer allows the programmer to attach automatic termination actions to an object. This way, if an object logically corresponds to a resource the finalizer can perform the task of deallocating the resource. In this section, we present  $\lambda_{weak3}$  which extends  $\lambda_{weak1}$  by modelling finalizers and their interaction with the garbage collector. Finalizers are not evaluated for their results. They are evaluated for the side effects they introduce. Hence, we introduce a simple form of side effect, abort, to make finalizers more interesting and at the same time keep the calculus as simple as possible. We then discuss the behavior of the calculus with finalizers and the abort construct. Finally, we discuss the non-determinism introduced by finalizers. In a calculus that has side effects, two different orders of finalizer evaluation could result in two different evaluation outcomes. We show however that in  $\lambda_{weak3}$ , with abort as the only form of side effect, finalizers do not affect the uniqueness of evaluation outcome.

### 8.1 $\lambda_{weak3}$ – A Calculus with Finalizers

We extend  $\lambda_{weak1}$  by adding finalizers to the calculus. The calculus  $\lambda_{weak3}$ , given in Figure 13, is described in this section. A heap allocated value  $hv \in \text{HVal}$  may include a finalizer, separated by a colon. may have a finalizer attached to them, separated from the object by a colon. This excludes weak  $e$  which is really a reference to an object rather than an object. Moreover, attaching finalization work to the special value  $d$  is meaningless.

Finalizers in  $\text{Fin}$  are composed using a restricted form of the expression syntax in  $\text{Exp}$ , so that nested finalizers are disallowed. Although nested finalizers may appear to necessitate recursive garbage collection, they, in fact, can be handled by a non-recursive garbage collector. However, until we further explore the value gained by modelling nested finalizers we decide to avoid this complexity by restricting them to a single nesting level. In addition, we exclude the use of weak references inside finalizers for simplification reasons.

<b>Programs:</b>			
(variables)	$w, x, y, z \in \text{Var}$		
(integers)	$i \in \text{Int}$	$::=$	$\dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$
(expressions)	$e \in \text{Exp}$	$::=$	$x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \lambda x. e \mid e_1 e_2 \mid$ $\text{weak } e \mid \text{ifdead } e_1 e_2 e_3 \mid i : f \mid \langle e_1, e_2 \rangle : f \mid$ $\lambda x. e : f$
(finalizers)	$f \in \text{Fin}$	$::=$	$x \mid i \mid \langle f_1, f_2 \rangle \mid \pi_1 f \mid \pi_2 f \mid \lambda x. f \mid f_1 f_2$
(heap values)	$hv \in \text{Hval}$	$::=$	$i \mid \langle x_1, x_2 \rangle \mid \lambda x. e \mid \text{weak } x \mid \text{d} \mid$ $i : f \mid \langle x_1, x_2 \rangle : f \mid \lambda x. e : f$
(heaps)	$H \in \text{Var} \xrightarrow{\text{fin}} \text{Hval}$		
(programs)	$P \in \text{Prog}$	$::=$	$\text{letrec } H \text{ in } e$
(answers)	$A \in \text{Ans}$	$::=$	$\text{letrec } H \text{ in } x$
<b>Evaluation Contexts and Instruction Expressions:</b>			
(contexts)	$E \in \text{Ctx}$	$::=$	$[] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E \mid \text{weak } E \mid$ $\text{ifdead } E e_1 e_2 \mid \langle E, e \rangle : f \mid \langle x, E \rangle : f$
(instruction)	$I \in \text{Instr}$	$::=$	$hv \mid \pi_i x \mid x y \mid \text{ifdead } x e_1 e_2$
<b>Rewrite Rules:</b>			
(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		
( $\pi_i$ )	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$ provided $H(x) = \langle x_1, x_2 \rangle$ or $H(x) = \langle x_1, x_2 \rangle : f$ 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 $H(x) = \lambda z. e$ or $H(x) = \lambda z. e : f$		
(ifdead)	$\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \text{d} \end{cases}$		
(garb)	$\text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$ provided $\text{letrec } H \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H'' \text{ in } e$ and $\text{letrec } H'' \text{ in } e \Downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e$		
<b>Auxiliary:</b>			
(gc)	$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H' \text{ in } e$ provided $hv$ is not strongly reachable in $(\text{letrec } H \text{ in } e)$ and $H' = \text{Fin}(H, hv)$		
(weak-gc)	$\text{letrec } H \uplus \{x \mapsto \text{weak } y\} \text{ in } e \xrightarrow{\text{weak-gc}} \text{letrec } H \uplus \{x \mapsto \text{d}\} \text{ in } e$ provided $y \notin \text{Dom}(H)$		
$\text{Fin}(H, hv) = \begin{cases} H' & \text{if } hv = \_ : f \text{ and } \text{letrec } H \text{ in } f \Downarrow_{(\text{garb})} \text{letrec } H' \text{ in } x \\ H & \text{otherwise} \end{cases}$			

Figure 13: The Syntax and Operational Semantics of  $\lambda_{\text{weak3}}$

The last addition to the syntax of programs includes adding finalizer-carrying expression to  $\text{Exp}$ . We add a finalizer version to those expressions that result in an allocated heap value. These are integer  $i : f$ , pair constructor  $\langle e_1, e_2 \rangle : f$  and lambda expression  $\lambda x.e : f$ . Note that programs that do not use finalizers can still be written.

We add the necessary evaluation contexts controlling the evaluation of the new pair constructor. With these evaluation contexts, finalizers are lazily evaluated. We choose a lazy evaluation in order to avoid unnecessary non-determinism. To exemplify this, consider the expression  $3 : 4$ . If finalizers are evaluated eagerly, this expression has two different valid evaluation paths. In the first, 4 is allocated on the heap and replaced by the new variable which binds 4. In the next step, the whole expression is allocated on the heap. In the second evaluation path, the whole expression is allocated immediately on the heap. We can, of course, add some mechanics to the syntax to handle this non-determinism, but we choose to avoid this complexity completely by evaluating finalizers in a lazy fashion. There is another reason to make the evaluation of finalizers lazy. Evaluating finalizers prematurely introduces bindings onto the heap which require additional care in the maintenance of the heap.

All the rewrite rules except for  $(\text{garb})$  are modified as expected. The only difference is that the rules  $(\pi_i)$  and  $(\text{app})$  have to consider two versions of expressions, one with a finalizer and one without.

The key difference in evaluation lies in the way garbage collection is performed. The  $(\text{garb})$  rule is written as before, but in  $\lambda_{\text{weak3}}$  the auxiliary rule  $(\text{gc})$  performs finalization work everytime an object is collected. In  $(\text{gc})$ , before removing the object  $hv$  we invoke  $\text{Fin}(H, hv)$ , an operator to finalize  $hv$  in the context of the current heap. In  $\text{Fin}$ , we first check if there is a finalizer associated with  $hv$ . If there is such finalizer we evaluate it in the context of the given heap. Since  $\text{Fin}$  is operated within the  $(\text{garb})$  rule, we disallow recursive garbage collection by evaluating the finalizer with all the rules except for  $(\text{garb})$ . This is denoted by  $\Downarrow_{(\text{garb})}$ . The result of the evaluation is discarded and the new evaluated heap is the one used in the outcome of the  $(\text{gc})$  rule. Note that this heap might contain bindings created by evaluating the finalizer.

To conclude, finalizers in Java allow the programmer to attach a user-defined action to a heap allocated object. This action is executed by the garbage collector upon removal of the object from the heap. Accordingly, heap values in  $\lambda_{\text{weak3}}$  can include finalizers. Finalizers' syntax and evaluation are restricted in order to prevent recursive garbage collection or evaluation of nested finalizers.

## 8.2 Adding Side Effects to $\lambda_{\text{weak3}}$

As shown above, the result of finalizer evaluation is never used. Finalizers are evaluated for their side effects. The side effects are recorded as a new set of bindings or potentially modified existing bindings in the heap. Therefore, having finalizers in a side-effect free calculus does not have much meaning. For this reason, we add a simple form of side effect to the calculus. The new expression  $\text{abort}$  is added to  $\lambda_{\text{weak3}}$ .

$$(\text{expressions}) \quad e \in \text{Exp} ::= \dots \mid \text{abort}$$

This is the only change we make to the calculus.  $\text{abort}$  is not a heap value and thus cannot be allocated on the heap. Moreover, there is no rule handling  $\text{abort}$ . Therefore, trying to evaluate  $\text{abort}$  runs the program into a stuck position. This way,  $\text{abort}$  is given the expected meaning of immediate termination of program evaluation. It might seem unnatural to introduce a construct that has no

evaluation rules and intentionally runs the program into a stuck position, but in the future we might mend this by using a type system. With a type system, abort might be given an integer type so it can be used wherever an integer is used. The standard progress lemma would not work here as well. However, it can be mended by restating it as follows: a typable expression in  $\lambda_{weak3}$  that is not in normal form is either abort or it can be further evaluated.

With the new addition to the calculus, we construct an example to motivate the use of finalizers and demonstrate their work in  $\lambda_{weak3}$ . Consider the case where the programmer wants to understand the behavior of a particular garbage collection implementation. She decides to write a program that aborts at the first time garbage collection kicks in. This would give her some information on the timeliness of operation of this particular garbage collector. In our calculus, the program  $P$ , in (A) of Figure 14, accomplishes that. The evaluation of this program begins by allocating three heap objects:  $\lambda x.x, 1 : \text{abort}$  and weak  $b$ . At this point, there are two possible scenarios. In the first scenario, given in (B) of Figure 14, the next evaluation rule is (app) which ends the evaluation by producing an answer. In the second scenario, given in (C) of Figure 14, the next evaluation rule is (garb), i.e., garbage collection has kicked in. The heap value  $1 : \text{abort}$ , being the only not-strongly reachable object, is chosen for collection. As defined in (gc), the Fin operator is invoked on this object thereby evaluating abort, the object's finalizer. At this point, the program reaches a stuck position, letting the programmer know garbage collection has occurred.

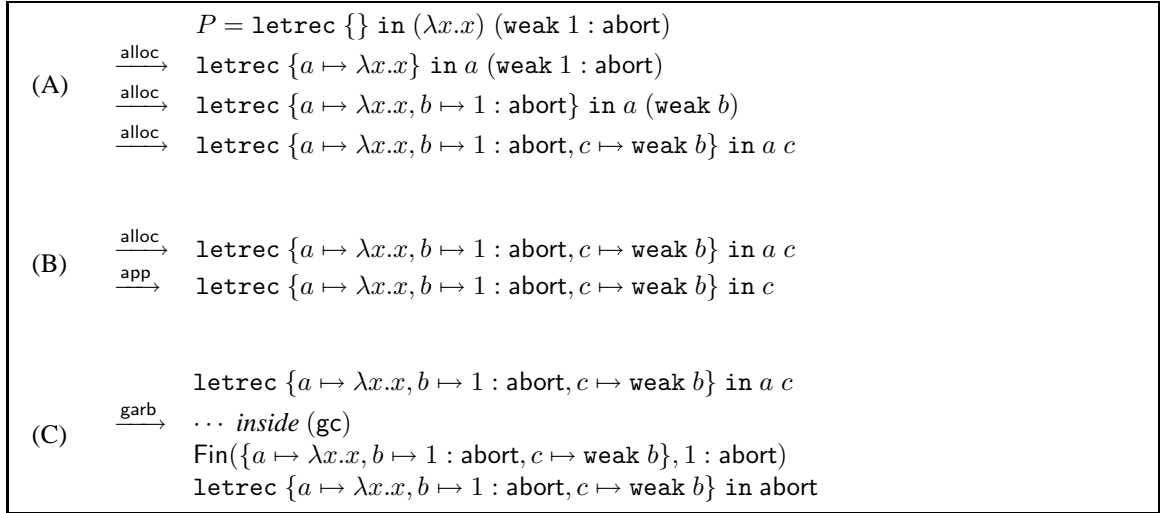


Figure 14: An Example of Using abort in a Finalizer

Generally speaking, finalizers might introduce non-determinism to a calculus that has side effects. Consider assignment for example. If finalizers are allowed to change bindings on the heap, such in the case of assignment, two different orders of finalizer evaluation might result in two different evaluation outcomes. That is, if two objects, each has its own finalizer, are candidates for garbage collection choosing one order of collection over the other might result in a different evaluation answer. However, in our calculus, there is no such problem. In  $\lambda_{weak3}$ , abort is the only form of side effect present. If a program tries to evaluate abort it stops immediately and does not reach the final answer. On the other hand, in programs that do not try to evaluate abort the order of object collection does not affect the final answer. The reason for this follows. The Fin operator might change the heap in the course of finalizer evaluation. However, it does so restrictively. In particular,

heap bindings can never be modified by the operation of `Fin`. The only heap change possible is the addition of new bindings to the heap. Given all this, the bindings that are added to the heap will always be unreachable from the root set of the program and thus can be garbage collected right away. Therefore, finalizers or the order in which they are evaluated can never affect the uniqueness of the final answer. It is important to note that in  $\lambda_{weak3}$ , as well as in  $\lambda_{weak}$ , evaluation uniqueness is not guaranteed due to the presence of weak references. However, outcome uniqueness is not lost as a result of adding finalizers to the calculus. In particular, if we add the finalizers above to  $\lambda_{gc}$ , where uniqueness of outcomes is guaranteed, it will be preserved.

If all the new bindings added by the `Fin` operator are garbage why do we keep them in the outcome of the `(gc)` rule? Although they do not play a role in the current calculus, if we extend the calculus to include more forms of side effects this decision may play a very big role.

## 9 Conclusions

This report provides a framework to formally reason about weak references in the context of Java. Specifically, we present a set of calculi based on  $\lambda_{weak}$  to model specific weak references aspects such as multiple levels of reachability and finalizers. Furthermore, we offer an alternative to the garbage-collection model given by  $\lambda_{weak}$ . We show it is more accurate in the sense it models garbage collection cases that are impossible to model in  $\lambda_{weak}$ .

The calculi presented in this report are based on  $\lambda_{weak}$  and  $\lambda_{gc}$ .  $\lambda_{gc}$  enables garbage collection modelling by making the allocation of objects on the heap explicit and by including garbage collection as a rewrite rule.  $\lambda_{weak}$  extends this effort to include the modelling of weak references. Weak references are explicitly created by the programmer and the garbage collection rule considers weakly reachable objects as well as unreachable objects when collecting objects. In both calculi, the decision of what objects to collect is left unspecified in order to give flexibility of instantiating it into different policies.

In  $\lambda_{weak1}$ , we offer an alternative to the modelling of garbage collection in  $\lambda_{weak}$ . To demonstrate the benefit of the new model, We consider the case where the garbage collector decides to remove garbage objects that are pointed by other objects that are also garbage. We show that although  $\lambda_{weak}$ , and in fact  $\lambda_{gc}$  as well, cannot model this scenario,  $\lambda_{weak1}$  can. On the other hand, we show that  $\lambda_{weak1}$  is capable of modelling any evaluation performed in  $\lambda_{weak}$ . Therefore,  $\lambda_{weak1}$  is a more accurate modelling of garbage collection.

This is not the only benefit of the new model. In  $\lambda_{weak2}$ , we use the new approach in order to model multiple levels of reachability. Furthermore, we apply different policies to each reachability level. That is, we model an eager garbage collection for weakly reachable objects and at the same time a lazy garbage collection for softly reachable objects. Along the same lines, in  $\lambda_{gc'}$ , we provide a framework to evaluate garbage collection under constraints of time and space. On the one hand, we offer an eager garbage collection that is restricted by time constraints and on the other hand, we offer a lazy garbage collection that is triggered by memory constraints such as memory reaching a certain capacity.

We then turn to examine the Java notion of finalizers in  $\lambda_{weak3}$ . Finalizers have nondeterministic behavior and usually make programs relying on them error prone. To give a meaning to finalizer evaluation in our calculus we introduce a simple side effect in the form of abort expression. We show that in the context of this calculus, finalizers do not affect the uniqueness of evaluation outcome.

## 9.1 Related Work

Morrisett, Felleisen and Harper’s  $\lambda_{gc}$ , in [7], provides a formal model for garbage collection and for heap allocated objects. Donnelly, Hallett and Kfoury’s  $\lambda_{weak}$ , in [2], extends  $\lambda_{gc}$  to provide a model for weak references. In our calculi, we chose a different formalism for garbage collection which we showed is able to model more cases than the model in  $\lambda_{weak}$  and  $\lambda_{gc}$ . Moreover, Our effort extends the work in  $\lambda_{weak}$  by including multiple levels of reachability along with different garbage collection strategies applied to each level. Furthermore, we provided a formal model for finalizers that includes a limited form of side-effect. To our knowledge, this is the only attempt to formalize these aspects of weak references.

## 9.2 Future Work

Several directions can be taken as a continuation of this work. In order to bring this model closer to weak references in Java, we can choose an object oriented calculus for our modelling. As an example, Featherweight Java, given in [6], can be used to model weak references. In order to do so, we first need to extend Featherweight Java with a heap modelling and garbage collection rules as done in  $\lambda_{gc}$ . Once we have that, we can model weak references, as presented in this report, in the context of Featherweight Java.

$\lambda_{weak2}$  can be generalized to have  $n$  different levels of reachability. Although we have only soft and weak references in this calculus, it is possible to have multiple types of non-strong references and to give each one a different collection policy. The reachability-based formalism, given in Section 5, is geared toward having that capability should the motivation to do so arise.

Finally, several possible extensions could be applied to  $\lambda_{weak3}$ . It would be beneficial to examine the calculus with side effects that would make finalizer evaluation break the uniqueness of result. Having such a framework would help study the unwieldy behavior of finalizers in Java and perhaps suggests tools to mend it. For example, a type system to tame finalizer behavior could be devised and studied. Another Java problem related to finalizers is object resurrection. Mechanisms to support object resurrection can be added to  $\lambda_{weak3}$ . With this extension, we can study object resurrection and devise the tools to limit it or prohibit it altogether.

## References

- [1] J. Bloch, *Effective Java programming language guide*. Mountain View, CA, USA: Sun Microsystems, Inc., 2001.
- [2] 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.
- [3] 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](http://citeseer.ist.psu.edu/felleisen92revised.html)
- [4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Second Edition*. Boston, Mass.: Addison-Wesley, 2000. [Online]. Available: [citeseer.ist.psu.edu/gosling00java.html](http://citeseer.ist.psu.edu/gosling00java.html)
- [5] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with Uml, Volume 1*. New York, NY, USA: John Wiley & Sons, Inc., 2002.
- [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. [Online]. Available: [citeseer.ist.psu.edu/igarashi99featherweight.html](http://citeseer.ist.psu.edu/igarashi99featherweight.html)
- [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] I. Nenov and P. Dobrikov, “Memory sensitive caching in java,” in *CompSysTech '05: Proceedings of the 6th international conference conference on Computer systems and technologies*. New York, NY, USA: ACM Press, 2005.
- [9] 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](http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.1.html)
- [10] M. Pawlan. (1998, Aug.) Reference objects and garbage collection. Sun Developer Network Article. [Online]. Available: <http://java.sun.com/developer/technicalArticles/ALT/RefObj/>
- [11] B. Stroustrup, *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
- [12] B. Venners. (1998, June) Object finalization and cleanup. JavaWorld Article. [Online]. Available: <http://www.javaworld.com/javaworld/jw-06-1998/jw-06-techniques.html>

## A Proofs

**Proposition 5.7 (Expressive Power of  $\lambda_{weak1}$  with respect to  $\lambda_{weak}$ ).** *Let  $P$  and  $P'$  be programs in  $\lambda_{weak}$ . If  $P \rightarrow_{\lambda_{weak}}^* P'$  then  $P \rightarrow_{\lambda_{weak1}}^* P'$  (the arrows' subscript annotation corresponds to the calculus under which the evaluation is being carried out). In words, the proposition states that if an evaluation path is taken from  $P$  to  $P'$  in  $\lambda_{weak}$ , then there is an evaluation path in  $\lambda_{weak1}$  going from  $P$  to  $P'$ .*

*Proof.* Proof by induction on the evaluation. The claim is trivially true for evaluations of length zero. Assume evaluation length is greater than zero and  $P'$  is the one but last evaluation step. That is,

$$P \rightarrow_{\lambda_{weak}}^* P'' \rightarrow_{\lambda_{weak}} P'$$

Note that in evaluations of length one  $P = P'$ . According to the induction hypothesis

$$P \rightarrow_{\lambda_{weak1}}^* P''$$

Consider all cases for the rule applied in  $P'' \rightarrow_{\lambda_{weak}} P'$

**Not (garb):** All the rules except for (garb) are the same in both calculi. Therefore,

$$P'' \rightarrow_{\lambda_{weak1}} P'$$

**(garb):** According to the assumptions, the following holds in  $\lambda_{weak}$

$$P'' \xrightarrow{\text{garb}} P'$$

Let  $P'' = (\text{letrec } H \text{ in } e)$  and  $P = (\text{letrec } H' \text{ in } e)$ . Let  $H_1$  be a heap such that

$$\begin{aligned} & \text{letrec } H \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H_1 \text{ in } e \quad \text{and} \\ & \text{letrec } H_1 \text{ in } e \Downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e \end{aligned}$$

Let  $H_2 = H - H_1 = \{x_1 \mapsto hv_1, x_2 \mapsto hv_2, \dots, x_n \mapsto hv_n\}$ . According to the assumption,

$$\text{Dom}(H_2) \cap \text{FV}(\text{letrec } H_1^s \text{ in } e) = \emptyset$$

However, this means that for every  $1 \leq i \leq n$  there is no strong reachability path from the root set to  $hv_i$ . That is, for every  $1 \leq i \leq n$   $hv_i$  is not strongly reachable. Consequentially, we can apply the (gc) in  $\lambda_{weak1}$   $n$  times as follows:

$$\begin{aligned} & \text{letrec } H \text{ in } e \\ & \xrightarrow{\text{gc}} \text{letrec } H - \{x_1 \mapsto hv_1\} \text{ in } e \\ & \xrightarrow{\text{gc}} \text{letrec } H - \{x_1 \mapsto hv_1, x_2 \mapsto hv_2\} \text{ in } e \\ & \dots \\ & \xrightarrow{\text{gc}} \text{letrec } H_1 \text{ in } e \end{aligned}$$

With this along with the fact that (weak-gc) is the same rule in both calculi we derive the following in  $\lambda_{weak1}$

$$P'' \xrightarrow{\text{garb}}^* P'$$

Given all this, we conclude

$$P \rightarrow_{\lambda_{weak1}}^* P'$$

□