Smarter Garbage Collection with Simplifiers

Melissa E. O'Neill
Harvey Mudd College, California
oneill@acm.org

F. Warren Burton
Simon Fraser University, Canada
burton@cs.sfu.ca

Abstract

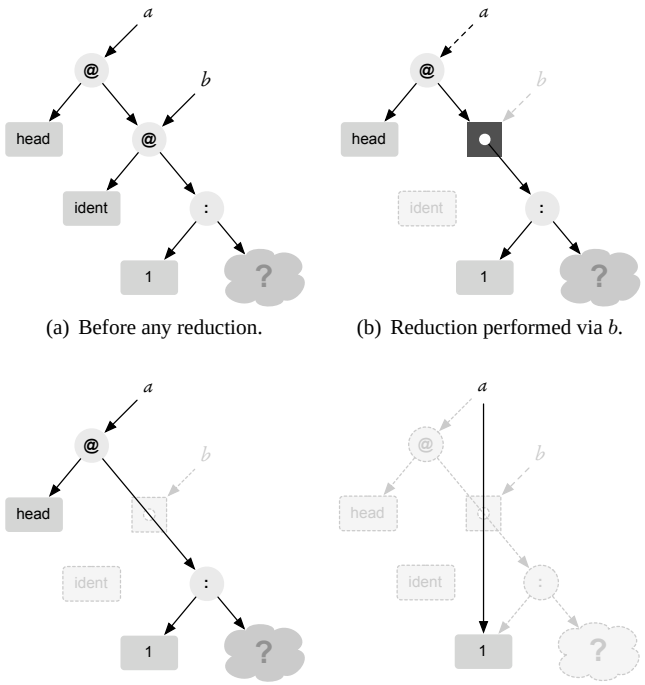
We introduce a method for providing lightweight daemons, called simplifiers, that attach themselves to program data. If a data item has a simplifier, the simplifier may be run automatically from time to time, seeking an opportunity to "simplify" the object in some way that improves the program's time or space performance.

It is not uncommon for programs to improve their data structures as they traverse them, but these improvements must wait until such a traversal occurs. Simplifiers provide an alternative mechanism for making improvements that is not tied to the vagaries of normal control flow.

Tracing garbage collectors can both support the simplifier abstraction and benefit from it. Because tracing collectors traverse program data structures, they can trigger simplifiers as part of the tracing process. (In fact, it is possible to view simplifiers as analogous to finalizers; whereas an object can have a finalizer that is run automatically when the object found to be dead, a simplifier can be run when the object is found to be live.)

Simplifiers can aid efficient collection by simplifying objects before they are traced, thereby eliminating some data that would otherwise have been traced and saved by the collector. We present performance data to show that appropriately chosen simplifiers can lead to tangible space and speed benefits in practice.

Different variations of simplifiers are possible, depending on the triggering mechanism and the synchronization policy. Some kinds of simplifier are already in use in mainstream systems in the form of ad-hoc garbage-collector extensions. For one kind of simplifier we include a complete and portable Java implementation that is less than thirty lines long.



(a) Before any reduction. (b) Reduction performed via b.

(c) After indirection elimination. (d) After further simplification.

Figure 1. Simplification in Graph Reduction.

1. Introduction

Sometimes it is sensible for a garbage collector to perform *simplification* work on a data structure it is traversing.

Consider, for example, the graph corresponding to a lazy functional program shown in Figure 1(a). In this figure, @ corresponds to function application; **:** is the list constructor; head is the function that returns the head of a list; and ident is the identify function (the question mark in the cloud represents the tail of the list, which is not actually needed by our computation, and could be arbitrarily large or even a lazy nonterminating computation). If b is evaluated (as shown in Figure 1(b)), it is customary to overwrite the application node with an *indirection node* (the square in our diagram) pointing to the result of the function application so that other computations need not reevaluate this result. The indirection says, essentially, "skip me and continue on to the node I point to", and can thus be elided if there is a suitable opportunity (resulting in the graph shown in Figure 1(c)). If only a is required, further simplification of the graph can eliminate a space leak—because nothing actually requires the tail of the list and the structure of the list is sufficiently known to identify its head and tail, we could eagerly evaluate head and make a point directly to the head of the list (as shown in Figure 1(d)).

Because both of these simplifications can save space, and because a tracing garbage collector (whether a copying collector or a mark-sweep collector) traverses the data structures of the program, a garbage collector for a lazy functional language will often contain some ad-hoc code to handle some issues of this kind [26, 35]. But language implementers working with an off-the-shelf garbage collector (such as the Boehm collector [7]) are faced with a dilemma, because these off-the-shelf collectors provide no obvious facility to perform these kinds of operations. In fact, there is not even a widely used term for this kind of code. It appears that an implementer must

either modify the internals of their off-the-shelf collector to handle the specific kinds of simplification they require, or implement their own garbage collector from scratch.

This paper

- Introduces *simplifiers* as a solution to the above problem and shows how simplifiers can be used to advantage in a variety of related problems.
- Shows that simplifiers can be very easy to implement in a language that already supports finalizers. An complete implementation of simplifiers and an application that uses simplifiers is given in Java.
- Discusses issues relating to the implementation and use of simplifiers, including direct support in a garbage collector for simplifiers, and various synchronization issues.
- Provides performance results that demonstrate that a well-chosen simplifier can have significant impact on the space performance of real-world programs.

2. Simplifiers

We define a ***simplifier*** to be a lightweight daemon that usually runs for a short period of time and does a small amount of work when activated.

A simplifier should satisfy the following two requirements:[1]

1. A simplifier will do something to improve the efficiency of a program. The correctness of a program will not depend on whether or not, or how often, a simplifier is activated.
2. Except for synchronization issues (discussed below), it will not matter when a simplifier is activated, as far as the correctness of a program is concerned.

This umbrella definition of simplifiers allows several possible realizations that differ according to

1. What triggers the activation of a simplifier; and
2. How simplifiers synchronize with each other and with the program as a whole.

In this paper we will be primarily concerned with simplifiers that are triggered by the garbage collector and are used to better manage memory.

Even if a garbage collector or programming language includes support for a particular form of simplification, there are also choices to be made regarding the level of code that may define simplifiers. Some possible variations include

1. Not exposing simplifiers to ordinary user-level code, but providing them as a part of a low-level interface to the garbage collector. In this context the simplifier mechanism can improve abstraction and separation of garbage collector and programming language.
2. Allowing user-level code to define simplifiers as an unsafe feature.
3. Allowing user-level code to define simplifiers, either with sufficient restrictions to avoid violations of the above requirements, or wrapped in such a way as to ensure that arbitrary user code cannot actually violate them.

2.1 Breeds of Simplifer

There are a number of different *breeds* of simplifiers, which differ in how they synchronize with each other and with the program as a whole.[2] We will always assume that simplifiers may be activated in any order, and that no fairness is guaranteed (i.e. one simplifier may be activated much more often than another in a completely arbitrary manner). Some breeds of simplifiers can be active concurrently with other simplifiers, whereas others cannot; some can be active concurrently with the main program, whereas others can be active only when the main program is halted; and still others must be synchronized with the main program in some way. There is also the question of whether a simplifier can run concurrently with the garbage collector. No single synchronization policy seems best in all cases, but there should be no problem with a single program using several different breeds of simplifiers for different purposes. Similarly, those designing and and implementing garbage collectors and programming languages may choose the breed of simplifier that is most expedient and easiest to implement.

In some situations, simplifiers may have additional constraints, such as running for at most a bounded amount of time on each activation or not allocating any memory.

2.2 Implementation Strategies

Different breeds of simplifier demand different implementation strategies. In this paper, we are concerned with simplifiers that are triggered by garbage collection, but even in that domain, there are a number of possible implementation choices. For example, as we shall see in the next section, simplifiers can be implemented in many languages without actually modifying the garbage collector at all. (We discuss implementation concerns more fully in Section 5.)

3. Example Implementation and Application

In this section we will describe a very simple—but complete and useable—implementation of simplifiers in Java and show how simplifiers can be used in an application. This section should be considered as a worked example, not a final answer.

3.1 Simplifier Invocation Infrastructure

The key to this small implementation of simplifiers is a class, *Watcher*, defined in Figure 2. A *Watcher* is an object that is created on the heap such that it will later be found and collected by the garbage collector in the next garbage-collection cycle; thus, after creating a watcher, the program drops its reference to it.[3] Each *Watcher* has a weak reference to another object called the watchee. The purpose of a *Watcher* is to see that the simplify method of the watchee is run when the garbage collector is run.

When the garbage collector is run, each *Watcher* should be finalized and destroyed. The finalize method of a *Watcher* calls the simplify method of its watchee, then creates a new *Watcher* to continue watching the watchee until the process is repeated on the next garbage-collection cycle.

A *Watcher* has a weak reference to its watchee so that the watchee can be reclaimed by the garbage collector if appropriate.

[1] We will call a simplifier that does not satisfy these requirements a *complicator*, and avoid complicators in this paper. We note that in general a compiler cannot distinguish between a simplifier and a complicator, and in some cases the distinction may be only in the eyes of the programmer. For example, a simplifier that can alter the result that a program produces, perhaps in a heuristic search, may be okay if the program is viewed as a nondeterministic program that may return any result that satisfies certain requirements.

[2] We use the term *breed*, beause many similar terms, such as *type*, *kind*, *sort*, etc. have other specialized meanings.

[3] Depending on the sophistication of the language and its garbage collector, some care may be required to ensure that the watcher is not collected the moment it is created but instead persists until the next garbage-collection cycle. We have found our implementation sufficient for Sun's implementation of the Java language.

```
import Java.lang.ref.*;
public class Watcher {
    WeakReference<Simplifiable> wref;

    private Watcher(Simplifiable watchee) {
        wref = new WeakReference<Simplifiable>(watchee);
    }

    static void watch(Simplifiable watchee) {
        new Watcher(watchee);
    }

    protected void finalize() throws Throwable {
        Simplifiable watched = wref.get();
        if (watched != null) {
            watched.simplify();
            new Watcher(watched);
        };
    }
}
```

Figure 2. Watchers for Triggering Simplifiers.

```
abstract class Simplifiable {
    abstract void simplify();

    public Simplifiable() {
        Watcher.watch(this);
    }
}
```

Figure 3. A Small but Complete Implementation of Simplifiers using Watchers.

Thus finalize needs to check for the special case where the watchee has preceded the *Watcher* in death.

For a class to use a simplifier, two things are necessary. First, the class must have a simplify method. Second, the class must be a subclass of the *Simplifiable* class, as defined in Figure 3.[4] Thanks to inheritance, the constructor for all objects derived from *Simplifiable* will cause a *Watcher* to be launched for the *Simplifiable* object at the time the object is created.

3.2 Simplifying Disjoint-Set Union

In Section 1, we used an example of simplification drawn from the implementation of functional languages, but now let us examine a small, self-contained example from the data-structures community—Tarjan's well-known disjoint-set–union data structure [32] (further examples are given in Section 4). In the disjoint-set–union problem, we can

- Create a new node that describes a singleton set whose element is not a member of any existing sets;
- Destructively union two (previously disjoint) sets—the nodes that previously referred to the two disjoint set now both refer to the same (unioned) set; and,
- Compare two nodes to see if they refer to the same set.

In Tarjan's data structure, an example of which is shown in Figure 4, a node may be either a root node (shaded black in the diagram) or an indirection node that points to another node—two nodes are considered to refer to the same set if they lead to the same root.

[4] We can also achieve similar ends with Java's interfaces.

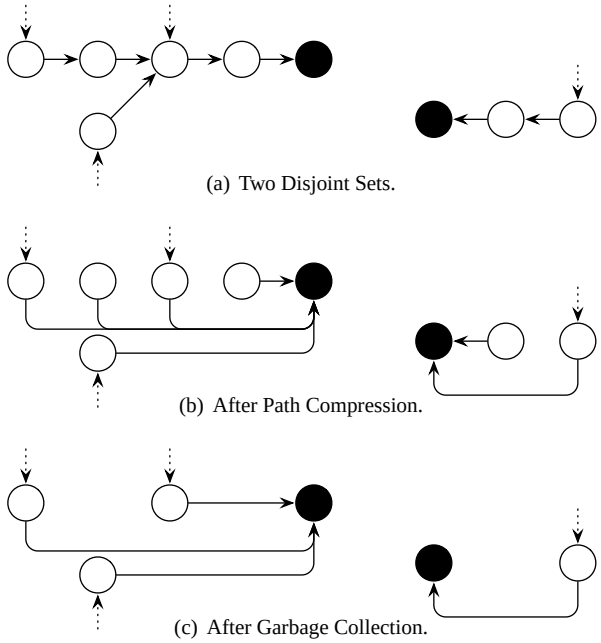


Figure 4. The Disjoint-Set Data Structure.

When comparing two nodes to determine whether they are in the same set, *path compression* is used to short-circuit paths to the root, redirecting links so that they point directly to the root node. Union is performed by making the root node in one set point to a node in the second set. Figure 4(a) shows an example of Tarjan's disjoint-set data structure. Figure 4(b) shows the data structure after path compression.[5] (The dotted arrows in the diagram indicate nodes that are externally referenced.)

Path compression speeds future comparisons, and allows unreferenced nodes to be garbage collected (Figure 4(c))—prior to path compression, a garbage collector would not be able to reclaim any of the nodes in our example. But the entire process of path compression only occurs if a path is traversed. For as long as no comparisons or unions are performed (or none that directly or indirectly involve a particular node), the path-compression optimization is not performed (on that node) and "logical garbage" may be retained. Worse, the amount of retained logical garbage could be arbitrarily large for a suitably pathological sequence of unions where we retain references to only a few of the resulting sets.[6]

Traversing the entire data structure and performing path compression on *all* paths (rather than just those that are encountered in ad-hoc partial traversals), would both ensure that no unnecessary garbage was retained and have the added benefit of reducing subsequent set-membership queries to constant real time, rather than almost-constant amortized time, at least until further union operations were performed.

[5] Ours is a simplified discussion of Tarjan's algorithm, because it serves only to provide a motivating example. Thus, we ignore the details of union by rank.

[6] If we use tricks involving weak pointers and finalizers (not entirely dissimilar to the ones used in Section 5.2), it *is* possible to cause nodes that are no longer externally referenced to be short-circuited, but doing so requires some careful bookkeeping, at least in part because the in-degree of nodes may be greater than one.

```
public class DisjointSetNode extends Simplifiable {
    DisjointSetNode indirect = null;

    public boolean inSameSet(DisjointSetNode that) {
        return this.pathCompress() == that.pathCompress();
    }

    public synchronized void union(DisjointSetNode that) {
        this.pathCompress().indirect = that;
    }

    public synchronized DisjointSetNode pathCompress() {
        if (indirect == null)
            return this;
        else
            return indirect = indirect.pathCompress();
    }

    public void simplify() {
        this.pathCompress();
    }
}
```

Figure 5. A Trivial Implementation of Disjoint Sets, including a Simplifier.

3.2.1 Implementation Essentials

Figure 5 shows a simple Java implementation of a disjoint-set class using path compression (but without union-by-rank).

As we can see, the *DisjointSetNode* type is associated with a simplifier that performs path compression. Even though a given call to simplify may require $O(n)$ real time, for a disjoint-set–union data structure of size n, if simplify is run on all n nodes of the structure, it will also require at most n work. Thus the simplification work parallels the cost of the tracing work performed during garbage collection.[7]

3.3 Caveats

In this example implementation of simplifiers, we have ignored many issues, such as synchronization, possible consequences of generational garbage collectors, and many issues that might arise in languages other than Java, or in applications in which simplifiers are used primarily for purposes other than garbage collection. These issues will be considered to various degrees later in the paper. This example implementation provides a concrete context for examining some of these issues.

It is also probably fair to say that our example code is "yet another abuse of finalizers." We agree with this criticism—while it may be *possible* to leverage finalization as a means to trigger simplification, it is not the best implementation strategy. A garbage collector that supports simplification more directly is preferable.

4. Applications

We have already seen three applications of simplifiers (indirection removal, avoiding space leaks in lazy functional programs, and performing path compression in Tarjan's disjoint-set–union data structure), but we have done so in the context of motivation and background. In this section we bring applications of simplifers to the foreground, reexamine these examples and the issues that surround them, and describe some additional examples.

[7] As our use of the disjoint-set–union data structure is only to provide a motivating example, we leave a proof of this bound as an exercise for the reader.

4.1 Indirection Removal

We mentioned indirection removal in Section 1, and stated that it is common to include indirection removal in the garbage collector for functional languages [26].

In our example, we removed a single indirection, but in practice long chains of indirection nodes can be created. Thus indirection removal can recover an arbitrarily large amount of memory.

In current collectors, indirection-removal code is typically comingled with other garbage-collector code as an ad-hoc garbage-collection extension. Recasting indirection removal as a simplifier allows us to refactor this code and lets the garbage-collector code focus on its primary task, garbage collection.

4.2 Disjoint-Set Union

Tarjan's disjoint-set–union algorithm [32] (discussed in Section 3) is, essentially, another example of indirection removal. As before, long chains of indirection nodes can be created, and in a world without simplifiers, those indirection nodes will only be removed if the path is traversed by the program, which is not guaranteed to happen. Thus, without simplifiers, a disjoint-set–union data structure could consume an arbitrary amount of memory that would not be considered garbage by a traditional tracing collector.

Our earlier discussion of Tarjan's algorithm explicitly avoided the complexities of "union-by-rank", yet it is the rank information (a rough approximation of tree height) that gives this data structure its almost-linear amortized-time bound. Ordinarily, such data structures must sometimes settle for a "rough approximation" because the true value is infeasible to calculate efficiently given the readily available information. Interestingly, although the most obvious simplifier for disjoint-set union is one that performs path compression, an alternative simplifier improves the accuracy of ranks.

It is instructive to consider indirection removal and disjoint-set union side by side. Both data structures have commonalities, and their simplifiers are very similar, but one is typically implemented via an ad-hoc garbage-collector extension, whereas the other has, to our knowledge, never been implemented inside a garbage collector. In the framework of simplifiers, it is possible to treat both examples similarly.

4.3 Amortized Algorithms

Our example of path compression in the disjoint-set–union data structure is just one example in the general area of amortized algorithms and data structures. In amortized algorithms, "expensive" operations are frequently delayed until they can be paid for with the stored potential of the data structure. Simplifiers can provide an alternative way to perform for these expensive operations, provided they can be executed incrementally.

4.4 Lazy Evaluation

In a garbage-collected lazy functional language, an unevaluated thunk (i.e., a computation that has been delayed until it is required) may retain an arbitrary amount of garbage. Consider, for example, the code snippet

```
x = snd pair
    where pair = (huge_value, tiny_value)
```

Until x is evaluated, it will be a thunk that refers to the entire pair, even though only the second part of the pair is required. It might appear that a suitably optimizing compiler could produce more sensible code for this example, but we don't need to stray very far to find cases that are not as obvious, such as

```
l = (fst pair) : (snd pair)
    where pair = expensive_function arg
```

(where : is the list cons operator). Even after the head of the list l is consumed, the entire pair will once again be retained to be given to snd.

The usual solution to this problem is to include special-purpose code in the garbage collector to detect this specific kind of space leak and correct it by replacing snd pair with a reference to the actual second component of the pair after the pair has been evaluated to weak-head normal form.

This example is therefore an instance of simplification. Thus we can say that current garbage collectors for lazy functional languages already perform simplification on a limited basis. But whereas these garbage collectors have been adjusted to have a specific understanding of the semantics of thunks, in our framework the garbage collector itself can treat heap object more generally and uniformly—thunks such as these just happen to have an associated simplifier.

4.4.1 Optimistic Evaluation

It is possible generalize the above mechanism to avoid even more wasted space. Consider, for example, the code

```
sum xs = sumacc 0 xs
     where sumacc n [] = n
           sumacc n (x : xs) = sumacc (n+x) xs
ns = [1,2,3,4,5]
nsum = sum ns
```

If we assume that inlining leaves us with nsum defined as sumacc 0 ns, with conventional lazy evaluation we would proceed no further. Even if we add strictness analysis, the conservatism involved in static analysis might leave us uncertain as to whether nsum is ever evaluated, and thus fail to evaluate nsum eagerly. But because sumacc is tail recursive and can execute in constant space, we can associate its thunk with a simplifier to perform one iteration of the function if its arguments have been evaluated sufficiently for the iteration to proceed.

After five runs of the simplifier, we would have determined that nsum = 15 and potentially freed the list ns. Even after one run of the simplifier, we may have potentially allowed ourselves to discard one element of ns, which is a potentially large savings for analogous situations where the list elements are larger.

4.5 Managing Cached Data

Programs can often improve their time performance at the cost of increased memory use by caching data in case it will be used again, saving the costs of recalculating or refetching it if it is indeed needed. But such a caching approach raises the question of how and when such cached data might be released if it later seems to be unnecessary.

The Java language [21] provides *soft references* as one way to manage this kind of data, but this method is not without problems. The Java approach leaves it up to the garbage collector to use its own heuristic for disposing of softly referenced data (a VM might use factors such as how recently the object was created or accessed, or it could instead only rely on measuring how much memory pressure the VM is under). In some cases, memory pressure might be entirely the wrong basis to dispose of cached data (e.g., if disposing of a memoized result will cause more calculation and even more memory pressure).

An alternative is to use a simplifier to control the continued existence of this cached data. Such an approach gives the programmer the control to make a more informed choice about what criteria to use to determine whether retaining this data continues to be worthwhile.

5. Implementing Simplifiers

We shall now examine two possible implementation strategies for simplifiers. Both strategies revolve around the garbage collector. The first approach extends the garbage collector to understand simplifiers, whereas the second leverages existing functionality present in many current garbage collectors to provide a "user-space" implementation.

In this section, we are mostly concerned with the implementation mechanics of simplifiers, namely how they can be run with little overhead, rather than questions that might arise from providing simplifiers as a feature for user programs.

5.1 Inside-Collector Implementation

To implement simplifiers as an extension to normal garbage collection, we extend the tracing phase of garbage collection to perform simplification on objects prior to tracing their children. In short, for each object examined by the garbage collector, there are two new steps that the garbage collector performs:

1. Determining whether the heap object has a simplifier that should be executed.
2. Executing the simplifier.

The first step requires us to first determine whether the object has an associated simplifier. This determination can be made based on the type of the object, or through an added field in the object's heap header describing the object as having a simplifier.

5.1.1 Basic Simplification

One strategy is to support only those breeds of simplifier that can safely run in the restricted environment of active garbage collection. This restriction limits simplifier use to implementers of language systems (or other programmers willing to write code that runs in a relatively unsafe environment where significant care must be taken).

The exact restrictions will vary depending on the collector and its environment. In some systems, collection only occurs at "safe moments", not halfway through the construction or update of an object, eliminating many synchronization concerns. In other systems, access to program data may need to be handled more cautiously.

5.1.2 Two-Phase Simplification

To support more general simplification, we may extend the basic simplification model above to provide a less restricted environment for simplification. To do so, we divide the simplifier itself into two phases, a *cheap phase* and a *safe phase*, either of which may do essentially nothing. Only the cheap phase is executed by the garbage collector itself; if a safe phase is required, it is run in a separate thread.

Cheap Phase

The cheap phase is executed by the garbage collector itself inline with tracing; that is, before proceeding with tracing the object's children. The cheap phase may

- Indicate that the safe phase should be run;
- Simplify the object by changing its fields;
- Indicate that pointers to this object should be redirected to point to some other object;

and it must

- Avoid allocating new storage and execute in constant space;
- Execute in (amortized) constant (and relatively short) time;
- Avoid acquiring any other resources; and,

- Avoid introducing any race conditions.

The conditions on the cheap phase are exactly those required to ensure that it can be executed by the garbage collector without any significant impact on the collection process. If we were to allow arbitrary code to execute inline with garbage collection, a variety of problems, such as deadlock or memory exhaustion, could occur [6].

Safe Phase

The restrictions that apply to code executing in the cheap phase preclude a number of useful simplifications, but this drawback can be eliminated by providing a safe phase that allows the execution of arbitrary simplification code. The safe phase is not executed by the garbage collector directly; instead, any safe-phase actions are performed in a separate thread. By keeping only weak references to objects awaiting safe-phase execution, we can ensure that pending simplification does not prevent an object from being collected. We may also prevent the simplification thread from falling behind if repeated garbage-collection cycles occur by not enqueuing a safe phase multiple times.

Partitioning Simplification into Phases

It might seem onerous to be required to determine whether a given piece of simplification code can execute in the relatively unsafe environment of an active garbage-collection cycle or whether it requires the safety of a separate thread, but we can always adopt the conservative strategy of running any code that cannot be shown to be cheap in the safe phase. The cheap phase only exists as an optimization, present because some existing simplifiers (such as our example from Section 1) can be executed inline with collection.

When garbage-collector–provided simplifiers are used solely by language implementers behind the scenes, the two-phase structure may be used to aid efficiency; but we might choose to run all user-supplied simplification in the safe phase if simplifiers are provided to user code.

It is also worth noting that static analysis, such as cheapness analysis [23, 16, 17], could be applied to arbitrary simplifier code to assist in determining what work can be performed in the cheap phase of simplification.

5.1.3 Differences from Finalizers

There are broad parallels between finalizers [6] and simplifiers, but also some key differences. In supporting finalizers, it is common to have to maintain an auxiliary data structure that lists those objects that have finalizers so that, even though they are not reachable by any other means, they may be reached for the purposes of finalization. In contrast, only reachable objects are simplified, and such a data structure is not required.

Interestingly, our division of simplifiers into cheap and safe phases could also be applied to finalizers and thereby occasionally eliminate the cost of running finalization code in a separate thread.

5.1.4 Generational Collection

In a generational collector, recently created objects are traced more frequently than long-standing objects, on the basis that these objects are more likely to have become garbage. Thus, in a naïve implementation in which simplification is tied directly to tracing, simplifiers would be applied much more frequently to young objects than old ones.

In fact, very young objects may benefit little from simplification. If they are short lived, the work of simplification may be wasted; moreover, if the objects were just created by the program, it seems reasonable to assume that less time has passed to provide an opportunity for simplification to arise (assuming, of course, that the program isn't deliberately creating obviously simplifiable data from the outset).

Thus, in a generational collector, it may make better sense to only perform simplification on objects in later generations. In other words, ignore young objects, but simplify adolescent and older objects. Such a rule is safe because we make no guarantees about when or if simplifiers are run.

5.1.5 Implementation Experience

Simplification can easily be provided in a custom-written garbage collector—we merely need to extend the tracing algorithm as described at the very start of this section. We have written one such collector for the programming language system used in Section 7.

It is also straightforward to extend some existing collectors. For example, the Boehm collector [7] can support basic simplification (or the cheap phase of two-phase simplification) because it allows objects to be associated with custom tracing functions. Objects can have an associated simplifier by using a custom tracer that first calls the simplifier.

Because the custom-tracer facility is already built into the Boehm collector, we can provide a simplifier mechanism that only has overheads for programs that actually use simplifiers. For objects that have simplifiers, calling out to the custom tracer does have a small overhead compared to normal tracing, but these overheads are minimal compared to the overheads of the finalizer-based trickery we showed in Section 3.

It is also possible to easily extend some Java collectors to support basic simplification. For example, when using a simple mark-sweep collector under the MMTk framework [3] of the Jikes RVM [2], it is sufficient to add lines of the form

```
if (object.toObject() instanceof Simplifiable)
    ((Simplifiable)object.toObject()).simplify();
```

to the mark phase. In this case simplifiers become easily accessible to any Java program, but all such simplifiers must take care not to allocate memory.

5.2 Outside-Collector Implementation

As we have already seen in Section 3, it is possible to achieve almost the same operational behavior as the inside-the-collector implementation approach we have described without actually modifying the garbage collector at all. All that we require is a means to detect when garbage collection is occurring so that we may perform simplification at the same time.

Some garbage collectors provide a hook that allows arbitrary code to be executed every garbage-collection cycle. In other cases, as we saw with the *Watcher* class described in Section 3.1, we can use the mechanism of finalization to provide such a facility. Of course, such an approach may be able to make few guarantees about when simplifiers will be executed, since garbage collectors are often vague about when they will perform finalization. But, in practice, we have found that this approach works adequately with several current collectors, including both the Boehm collector and the collector in Sun's Java 1.5 virtual machine.

Even with a mechanism to determine when garbage collection is occurring, there are a number of design choices we can make when implementing simplifiers outside the collector.

5.2.1 Concurrent Simplification

In this model, each simplifier runs concurrently in its own thread, and thus simplifiers run in parallel not only with the main program but also with each other. In cases where parallel execution could be problematic, appropriate synchronization primitives must be used by the simplification code.

This model is the one used by the code we presented in Section 3, in large part because each simplifier was triggered by the finalizer of a distinct *Watcher* object, and Java is free to run each finalizer in its own thread.

5.2.2 Serial Simplification

In this model, all simplifiers are run from a single thread, and thus the simplifiers run in parallel with the main program but not with each other. The only race conditions that simplifiers need be concerned with are interactions with the main program, and not with one another. Once again, synchronization primitives may be used by the simplification code to avoid problems.

This model can be implemented by maintaining a list of objects that require simplification, and traversing that list at every garbage-collection cycle. The list needs to be maintained in such a way as to not prevent dead objects from being collected, but this requirement is easy to achieve.

A variation on this scheme would be to use different lists for different types of simplifiable objects, and thereby allow greater concurrency.

6. Other Synchronization Options

In this section we examine the synchronization concerns that exist for breeds of simplifier that allow normal program execution and simplifiers to simultaneously access the same objects, thereby allowing the possibility of race conditions. (In some environments, the available breeds of simplifier may naturally sidestep such concerns. For example, if simplifiers are run from a garbage collector that is only invoked when program execution is in a "safe state" where changes to program data can be made safely, simplification will not be prone to these kinds of races.)

If arbitrary code is allowed to execute when a simplifier is activated, we run the usual risks that occur with concurrent code execution, such as race conditions or deadlock. But these issues are not new; in fact, they are already reasonably well understood in the context of finalizers [6]. The usual solution applied to finalizers is to run them in a separate thread and provide conventional multithreaded mutual-exclusion mechanisms to give programmers the necessary means to ensure that the execution of a finalizer does not unduly affect other code. The same approach may be taken for simplifiers, and we have already suggested as much elsewhere in this paper. But simplifiers are not exactly like finalizers, so there are some additional options with regard to synchronization that we shall explore here.

Because simplifiers make no promises as to when or if they will run, a simple mechanism for ensuring safe simplifier behavior is to prevent simplification from occurring whenever the action of the simplifier could introduce a harmful race condition, either in the underlying program or in the simplifier itself. Below we present two alternative ways for preventing a given simplifier from running at inopportune times.

Both approaches are intended to allow a wider range of simplification work to be performed by cheap simplifiers (see Section 5.1.2) in a classic stop-the-world garbage collector, because in that situation we are prohibited from using mutual exclusion based on locks. Neither approach can be applied to concurrent collectors, incremental collectors, or simplifiers run in their own thread because disabling simplification for the object does not prevent an already activated simplifier from continuing to run. This restriction is not as severe as it might seem, because in many of these cases the simplifier could instead use conventional mutual-exclusion techniques.

6.1 Locally Disabling Simplification

We can avoid running the cheap simplifier phase by turning off simplification for that object (e.g., by modifying the object header to remove its association with a simplifier) whenever it would be dangerous to allow the object to be simplified.

Explicitly disabling simplification costs no more than explicit synchronization operations, such as the synchronized methods of Java, because the operations are essentially the same. In both cases, the object's state needs to be updated.

6.2 Locally Avoiding Simplification

In the preceding technique, it was the program's responsibility to lock out simplifiers when necessary. The obvious variation on this approach is to make avoiding race conditions the simplifier's responsibility. In this case, it is the responsibility of the simplifier to check the object's state itself and only execute the simplification action if it is safe to do so.

7. Performance

At this point, a skeptical reader might well ask, "Well, this is all very well in theory, but what about in practice?". Are simplifiers useful, and, in particular, can they really aid garbage collection? In this section, we show that they can indeed help collection. Obviously, simplifiers may not be appropriate for every problem, but even a single example can show that they are *very* useful for *some* problems.

As an example, we will use the runtime for a minimalist toy functional programming language based on combinator reduction [34]. In this system, user programs and even user data structures are formed from the application of very simple built-in functions, represented as a graph. Program evaluation is achieved through graph reduction.

In Sections 4.1 and 4.4, we described several opportunities for applying simplification to functional-programming languages. For this simple combinator-based system, we shall distill those ideas into two simple opportunities for simplification:

- The indirection nodes that arise as part of graph reduction: Whenever possible, they should be to short-circuited.[8]
- Function applications that had "unknowns" as arguments when they were created but now have simple knowns can safely be eagerly evaluated: For example, (*<lazy-thunk>* "Big string") cannot be safely evaluated, but if later evaluation reveals that the lazy thunk evaluates to an instance of the K combinator, such as (K "Ha!") we can safely simplify what we now know to be (K "Ha!" "Big string") to "Ha!".

7.1 A Simple Example under Simplification

We examine the performance of executing the code fragment below to find the value for result.

```
countupto n = counter 0
      where counter i = [], if i = n
                counter i = i : counter (i+1), otherwise
result = sum (ns ++ [head (tail (tail ns))])
      where ns = countupto 1000
```

In this computation, ns is a lazy list for the first thousand non-negative integers. We then form a new lazy list by appending ns to its own third element (the ++ operator is list append, defined recursively in the obvious way). This list is then summed (using the function from 4.4.1).

[8] Irrespective of the presence of simplifiers, the runtime system does short-circuit indirection nodes whenever it encounters them in traversing the

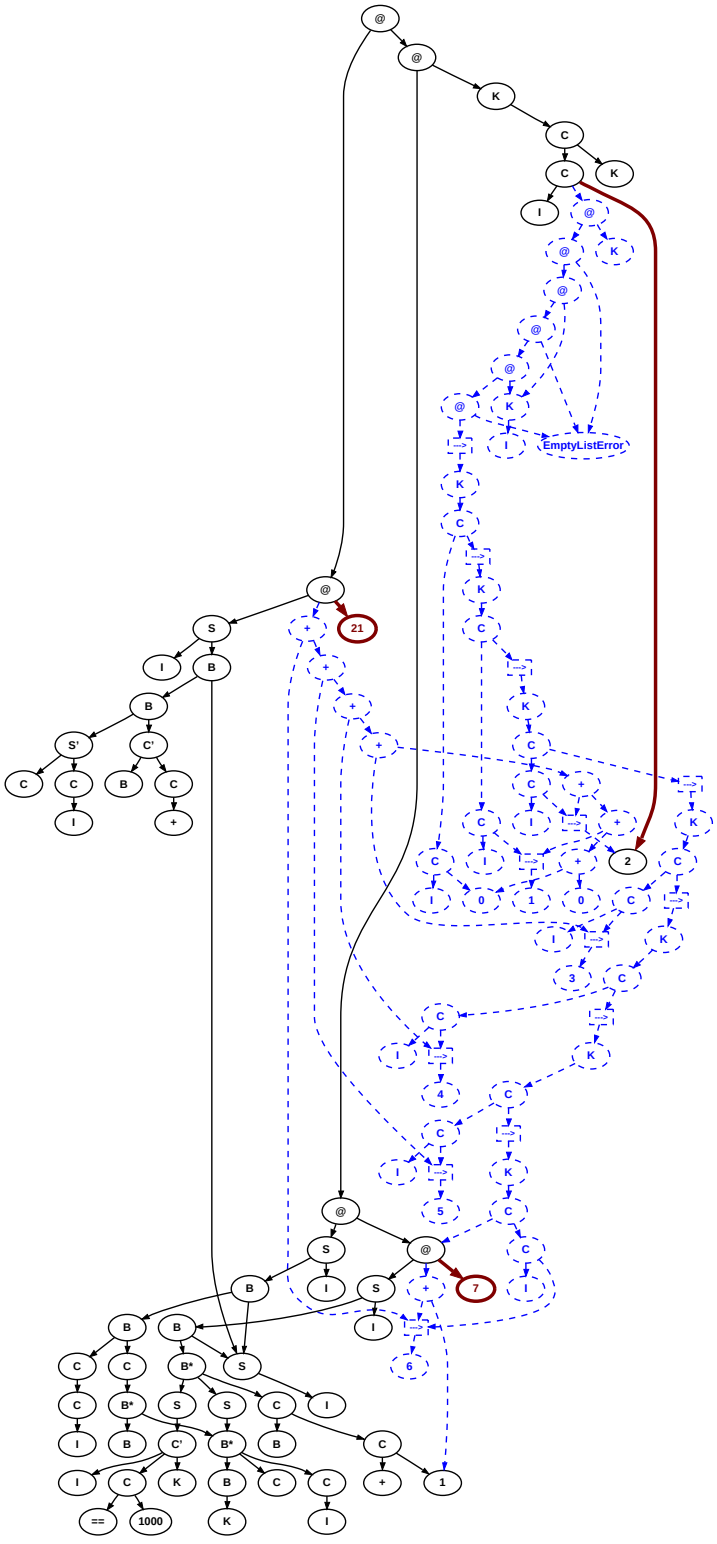
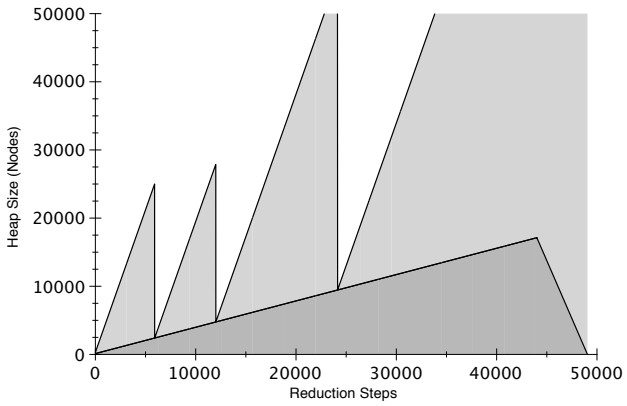
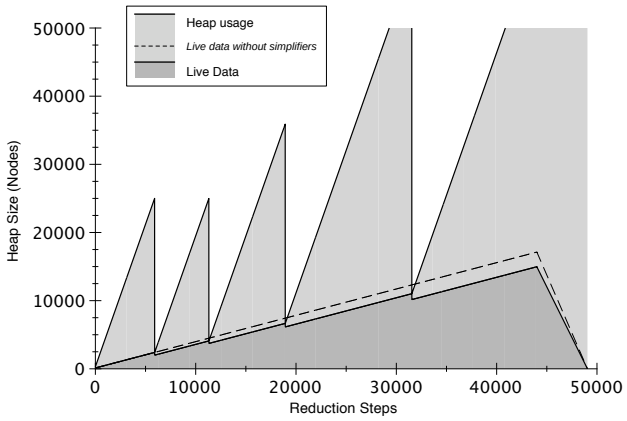


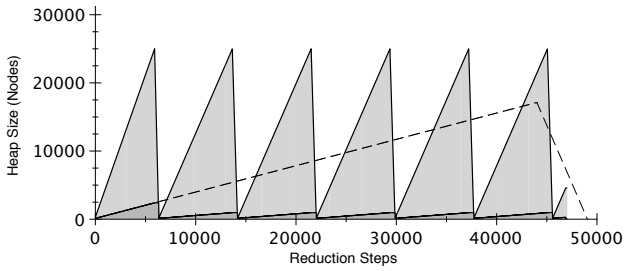
Figure 6. What Happens During Simplification.



(a) Without Simplifiers.



(b) With Indirection Simplifiers.



(c) With All Simplifiers.

Figure 7. Performance of sum.

Ordinarily, the evaluation order of lazy languages and the usual rules for tracing collection conspire to cause this program to use $O(n)$ space. We would normally expect the entirety of the ns list to persist until the end of the computation because head (tail (tail ns)) refers to the start of the list. The program is somewhat contrived, but serves as a short, easy-to-understand example of the kinds of problems that can arise in much larger programs. Its brevity also allows us to examine its execution in more detail than is possible for larger programs.

Figure 6 shows what happens if the simplifiers we described earlier can act on the heap. In this figure, the dotted lines (blue in color renderings) show data that is "simplified away" and the thick lines (red in color renderings) show additions to the graph. At this stage in execution, we have just produced the seventh element of ns and are in the process of computing the eighth (which has the value 7). Two important simplifications have occurred here. First, the sum has been computed rather than waiting to be computed later, and second, head (tail (tail ns)) has been evaluated, freeing the front of the list (we can also see some indirection nodes in the graph (shown as ---> inside a small rectangular node), but these exist entirely in the eliminated section of the graph). After simplification, computation can proceed with the list acting as a stream—there is no need to preserve its contents. Without simplification, all one-thousand elements of the list will be retained; in other words, without simplification, the "unnecessary" subgraph shown in the diagram will grow to well over one-hundred times its present size.

Figure 7 shows graphs of the memory use of the program. In these graphs, the upper line shows total heap allocation and the lower line shows the amount of data that is reachable from the root set. The sharp discontinuities in the upper line are garbage-collections (the garbage collector begins with a heap of capacity 25000 fixed-size nodes and expands the heap as necessary to keep it at about ten times the amount of live data). We have drawn all three graphs to the same scale to facilitate easy visual comparison of the memory usage.

Figure 7(a) shows the performance of the program without any simplification. As we expect, the amount of "live" data has linear growth throughout the execution of the program. Figure 7(b) shows how indirection elimination can make small dents in the space usage of the program but does not change its fundamental $O(n)$ space behavior. Figure 7(c) shows the dramatic difference a simple simplifier that "understands" the semantics of program data can make—heap usage is dramatically reduced, resulting in $O(1)$ space behavior.

Slightly harder to see in the graphs, but nevertheless noticeable, is that the code without simplifiers requires 4.28% more reduction steps in the normal control flow of the program compared to the version with all simplifiers running (49,051 compared to 47,037).

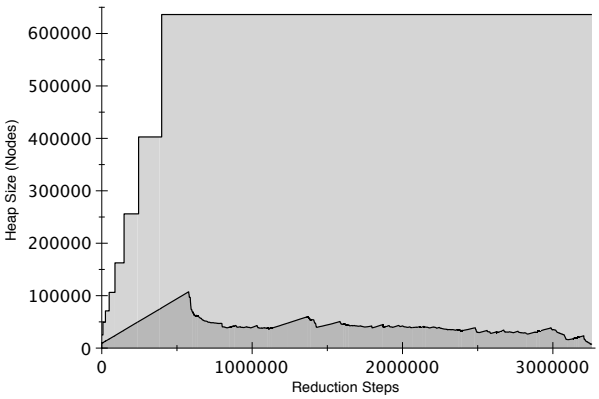
Table 1 shows execution times for this program (and the two programs discussed in Section 7.2) with simplifiers enabled and with the entire simplification framework removed.[9] From these numbers we can see that simplification can significantly speed up execution times. This improvement seems reasonable given the graphs—a smaller heap means much less tracing work for the collector. Note, however, that these particular timings are for our own simple tracing collector—for more some highly tuned collectors, departing from a highly optimized tracing loop to call a simplifier might lead to higher overheads and thus lower time-performance

graph, but there is no guarantee that normal control flow will lead to any particular indirection being encountered in a timely way.

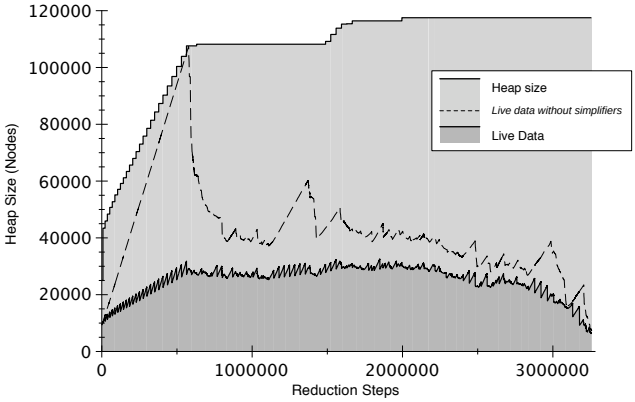
[9] Timings were taken on a Dual 2.0GHz PowerPC G5 running Mac OS X 10.4.7 with 5.5 GB of RAM and energy-saving CPU throttling disabled. The time value is the median from seven runs (although the variance between runs is negligible).

Program	Without Simplifiers	With Simplifiers	Speedup Ratio
sum	1.03	0.21	4.90
quicksort / filter	163.88	39.67	4.13
quicksort / partition	12.46	10.61	1.17

Table 1. Performance Timings (in seconds).



(a) Without Simplifiers.



(b) With Simplifiers.

Figure 8. Performance of quicksort.

benefits. (But other efficient collectors, such as Thomas's tailored collector [33] have no such tracing loop.)

7.2 Quicksort

Because our previous example could be seen as being constructed specifically to showcase the power of simplification, we also show a favorite short functional program that often surprises the unwary with its space performance—quicksort, defined as:

```
quicksort [] = []
quicksort (h : t) = (quicksort lhs) ++ [h] ++ (quicksort rhs)
    where lhs = filter (< h) t
          rhs = filter (>= h) t
```

where filter is the well-known list function, defined recursively in the obvious way.

Figure 8 shows the performance of quicksort executed with and without simplifiers. In the test, we use quicksort to sort a list

of one thousand random integers and then print the sorted list.[10] We can again see the difference that simplification during garbage collection can make, at times reducing heap usage by a factor of four.

The graph follows similar conventions to the previous graphs, but, because of the greater number of reductions, we avoid the noisiness of showing each garbage collection—instead, we show the heap's high-water mark. Also, because there is more fine detail in these graphs, each graph uses its own scale for the y-axis.

As we mentioned in the previous subsection, Table 1 lists the execution times for this quicksort algorithm. We can see once again that running simplifiers significantly improves the time performance of the algorithm, presumably due to the smaller heap size.[11]

7.2.1 Better Quicksort

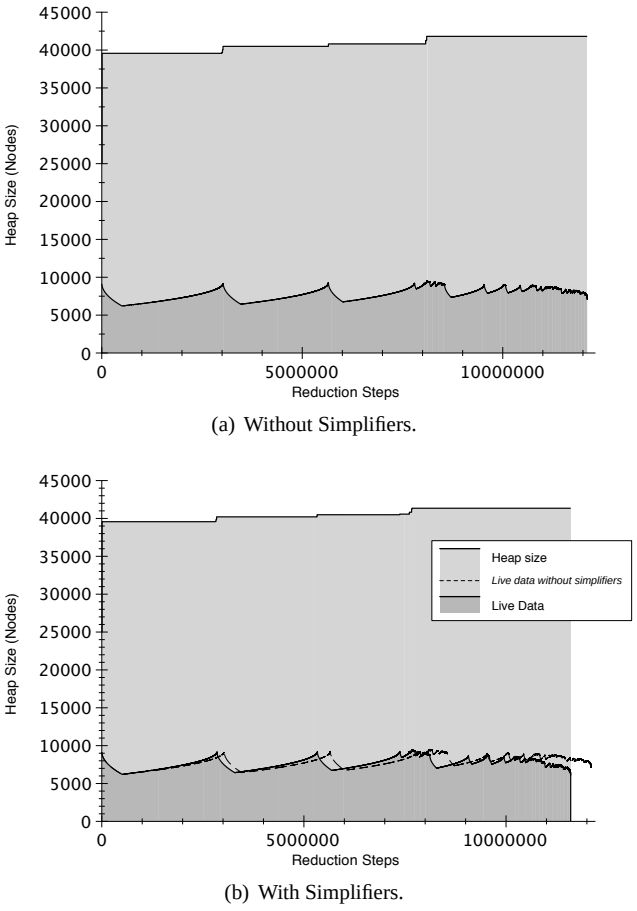
A seasoned functional programmer might recommend that quicksort be written slightly differently, as follows:

```
quicksort [] = []
quicksort (h : t) = (quicksort lhs) ++ [h] ++ (quicksort rhs)
      where (lhs,rhs) = partition (< h) t
```

where partition is the well-known list function, defined recursively in the obvious way.

Figure 9 shows the space behavior of this alternative implementation. We can see that it is generally much more parsimonious in its heap usage and benefits less from simplification. Interestingly, however, we can also see that this version requires many more reduction steps than the previous quicksort implementation (largely because partition spends much of its time packing and unpacking the tuples it uses to return its results). Interestingly, the one clear benefit of simplification in this version is that the program completes with fewer reduction steps.

Table 1 shows that despite the greater number of reduction steps, the reduction in heap usage causes this version of quicksort to run significantly faster than the version from the previous section. But even here (perhaps surprisingly given the small differences between the two graphs in Figure 9) the version with simplifiers does run noticeably faster than the version without.



(a) Without Simplifiers.

(b) With Simplifiers.

Figure 9. Performance of a Revised quicksort.

7.3 Performance Conclusions

The thesis of this section is that "real world" circumstances exist where using simplifiers improves the performance properties of programs. Our small but realistic examples show that the use of well-chosen simplifiers can lead to dramatic improvements in both time and space behavior. Of course, simplifiers are not a panacea—in some situations simplifiers may only bring modest or even negligible benefits.

8. Conclusion

In our view, there should be no debate as to whether simplifiers should exist or can perform useful work. Even if the performance results from Section 7 are taken with a grain of salt (as almost all performance results should be), they nevertheless present a fairly compelling case for providing a simplification mechanism.

In fact, simplification is already performed by ad-hoc code in custom garbage collectors written for some functional languages (although this code is usually fairly limited in scope and not extensible). Prior to our paper, however, there has been no unifying concept corresponding to that of simplifiers, only scattered, ad-hoc pieces of the puzzle.

We hope that having made a case for simplifiers and explained some of the implementation issues, they might become a more widely available and widely used garbage-collector feature.

9. Future Work

The simplifier abstraction is a new idea, so there is almost certainly work to be done in determining what kinds of simplifiers work well in practice. In addition, assessing the value of a given simplifier may turn out to be a topic worthy of study itself, as more complex simplifiers may involve a variety of tradeoffs (in particular, space vs speed).

In this paper we have focused on using simplifiers for memory management. In this context, simplifiers are activated (directly or indirectly) by the garbage collector. Other mechanisms for simplifier activation are possible, and probably worthy of study.

10. Related Work

Simplifiers can be seen as being a natural complement to finalizers. Finalizers have appeared in a number of languages and are becoming an expected feature of most new garbage-collected language implementations; appearing, for example, in object-oriented languages such as Java [21] and C# [14], as well as in functional-language implementations, such as the Glasgow Haskell Compiler

[10] We print the list because otherwise laziness would cause no actual execution to occur.

[11] Perhaps one additional conclusion one might reach from these numbers is that if you have a thousand numbers to sort, using quicksort on a toy combinator-reduction machine is not likely to be the fastest method.

[13, 27]. Hayes [18] provides one of the earliest reviews of finalization and presents a number of applications and issues to consider. Boehm [6] extends this latter work and specifically addresses synchronization concerns that arise in finalization. As we mentioned in Section 5.1.3, some of these concerns do not apply to simplifiers.

The techniques used by the *Watcher* class in our Java implementation are echoed by the SIGGC signal proposed for Standard ML [28] and the Finalize.everyGC mechanism present in Mozart/Oz [12].

Using the garbage collector to bypass indirection nodes is discussed by Peyton Jones [26]; using the garbage collector to eliminate obvious space leaks in lazy functional programs is discussed by Wadler [35]. The simplifiers we present here can be seen as a generalization of both of these approaches. Although Wadler's method handles one space leak in lazy functional programs, current research shows that plenty of opportunities to save space remain [31, 30, 29, 5, 1], although we do not claim here that simplifiers can address all of them.

When simplifiers are applied to aid the efficient evaluation of functional programs, they may be seen as being related to optimistic evaluation [15, 22], which is itself related to speculative evaluation [9]. The key difference is when these simplifications occur—in optimistic evaluation, we must decide whether to evaluate or not at the point a call is made, whereas with simplifiers, we may delay the decision until later, perhaps until enough evaluation has taken place elsewhere for it to be worthwhile.

To allow as much simplification work as possible to be performed in-line with garbage collection, we have divided simplifier execution into a cheap phase and an safe phase. Cheapness analysis [23, 16, 17] provides one possible mechanism for determining whether code is cheap enough to execute in this phase.

Similarly, in addition to cheapness analysis, other forms of abstract interpretation [10, 11] may be useful in determining when it is both worthwhile and safe to perform simplification. Strictness analysis [24, 8, 25, 36], update analysis [4, 19], and sharing analysis [20] may all be able to provide useful information.

11. Acknowledgments

The authors would like to thank everyone who has helped improve earlier drafts of this paper. In particular, we would like to thank the anonymous referees for their efforts, Simon Peyton Jones for his helpful comments and suggestions at the time we were first exploring the concept of simplifiers; and Forrest Briggs, David Malec, Jonathan Beall and Claire Connelly their feedback on more recent drafts. We would also like to thank Andrew Wallis for showing us how easy it is to add simplifier support to MMTk.

References

[1] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 269–279, New York, NY, USA, 1998. ACM Press.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, Sept. 1989. ACM Press.

[5] H. J. Boehm. Bounding space usage of conservative garbage collectors. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–100, New York, NY, USA, 2002. ACM Press.

[6] H.-J. Boehm. Destructors, finalizers, and synchronization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–272, New York, NY, USA, 2003. ACM Press.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.

[8] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7(3):249–278, 1986.

[9] F. W. Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, 34(12):1190–1193, 1985.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977. ACM Press.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, Jan. 1979. ACM Press.

[12] D. Duchier, L. Kornstaedt, T. Muller, C. Schulte, and P. Van Roy. System modules. Technical Report available in Mozart documentation, available at `http://www.mozart-oz.org`, 1999.

[13] R. K. Dybvig, C. Bruggeman, and D. Eby. Guardians in a generation-based garbage collector. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 207–216, New York, NY, USA, 1993. ACM Press.

[14] ECMA. *Standard ECMA-334: C# Language Specification*. ECMA International, Geneva, 3rd edition, June 2005.

[15] R. Ennals and S. L. Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, New York, NY, USA, 2003. ACM Press.

[16] K.-F. Faxén. Cheap eagerness: Speculative evaluation in a lazy functional language. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 150–161, New York, NY, USA, 2000. ACM Press.

[17] K.-F. Faxén. Dynamic cheap eagerness. In *IFL '02: Selected Papers from the 13th International Workshop on Implementation of Functional Languages*, pages 105–120, London, UK, 2002. Springer-Verlag.

[18] B. Hayes. Finalization in the collector interface. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 277–298, London, UK, 1992. Springer-Verlag.

[19] P. R. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, U.S.A., Jan. 1985. ACM Press.

[20] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 54–74, Imperial College, London, Sept. 1989. ACM Press.

[21] B. Joy, G. L. Steele, Jr, J. Gosling, and G. Bracha. *The Java Language Specification*. The Java series. Addison-Wesley, 3nd edition, June

2005.

[22] J.-W. Maessen. Eager haskell: Resource-bounded execution yields efficient iteration. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 38–50, New York, NY, USA, 2002. ACM Press.

[23] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, pages 269–281, London, UK, 1980. Springer-Verlag.

[24] A. Mycroft. *Abstract Interpretation and Optimising Transformations of Applicative Programs*. PhD thesis, Computer Science Department, Edinburgh University, Edinburgh, Scotland, 1981.

[25] A. Mycroft and A. Norman. Optimising compilation—lazy functional languages. In *SOFSEM '92: Proceedings of the 19th Software Seminar*, Bratislava, 1992. INFOSTAT.

[26] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1987.

[27] S. L. Peyton Jones, S. Marlow, and C. Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *IFL '99: Selected Papers from the 11th International Workshop on Implementation of Functional Languages*, pages 37–58, London, UK, 2000. Springer-Verlag.

[28] J. Reppy. Asynchronous signals in Standard ML, 1990.

[29] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 104–113, New York, NY, USA, 2001. ACM Press.

[30] R. Shaham, E. K. Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *ISMM '02: Proceedings of the 3rd International Symposium on Memory Management*, pages 64–75, New York, NY, USA, 2002. ACM Press.

[31] R. Shaham, E. K. Kolodner, and S. Sagiv. Automatic removal of array memory leaks in Java. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 50–66, London, UK, 2000. Springer-Verlag.

[32] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[33] S. P. Thomas. Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach. *Information Processing Letters*, 56(1):1–7, Oct. 1995.

[34] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[35] P. Wadler. Fixing some space leaks with a garbage collector. *Software—Practice and Experience*, 17(9):595–608, 1987.

[36] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *FPCA '89: Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 385–407, London, UK, 1987. Springer-Verlag.