Teaching Garbage Collection without Implementing Compilers or Interpreters

Gregory H. Cooper
Google, Inc.
ghc@google.com

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Jay McCarthy
Brigham Young University
jay@cs.byu.edu

Robert Bruce Findler
Northwestern University
robby@northwestern.edu

ABSTRACT

Given the widespread use of memory-safe languages, students must understand garbage collection well. Following a constructivist philosophy, an effective approach would be to have them implement garbage collectors. Unfortunately, a full implementation depends on substantial knowledge of compilers and runtime systems, which many courses do not cover or cannot assume.

This paper presents an instructive approach to teaching GC, where students implement it atop a simplified stack and heap. Our approach eliminates enormous curricular dependencies while preserving the essence of GC algorithms. We take pains to enable testability, comprehensibility, and facilitates debugging. Our approach has been successfully classroom-tested for several years at several institutions.

Categories and Subject Descriptors K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms Languages

Keywords Garbage collection

1. INTRODUCTION

Many students appear to have a poor grasp of garbage collection (GC) [8], because it is an automated process that runs without their control. This makes some students intimidated and untrusting of it (which can result in their using unsafe programming languages), while others use it without understanding its impact on performance.

Our antidote is to lay bare the underlying mechanisms by having students *implement* them, in keeping with constructivist educational principles. This approach is consistent with what many programming languages courses already do, following the venerable tradition of studying languages through *definitional interpreters* [11]. A definitional interpreter defines a language by a lightweight implementation, in lieu of abstract mathematics. These interpreters allow students to actually run programs in their languages and explore the consequences of alternate definitions.

There are several programming language texts and courses based on definitional interpreters [1, 6, 9], but most of them do not cover GC within the implementation-oriented tradition. Implementing a GC is taxing because it requires a grasp of low-level concepts such as walking the stack, pointer arithmetic, manipulating raw memory, and so on, and most real-world runtime systems—which focus on performance—are not designed to enable easy modification. The alternative is to provide students with a toy language's implementation, but such a language will necessarily be small and hence hard to write interesting programs in, thereby discouraging testing and experimentation.

Furthermore, even to work with a toy language, students must first learn about interpreters or compilers, which is a significant curricular prerequisite. In contrast, many other university-level courses could and do teach GC, including runtime systems, operating systems, and even middleware. Thus, finding a way to implement GC without a heavy language implementation prerequisite could have broad use.

In this paper, we present a framework for building, debugging, and exploring garbage collectors. It meets several important criteria:

- It *minimizes curricular dependencies*: we enable students to learn and write garbage collection algorithms in a full-fledged language with an IDE. Students do not have to master other complex topics, such as compilers and continuation-passing style, to implement their GC. This makes the approach fit better as a module in a variety of different courses that might want to teach this material.

- It is *flexible*: students can investigate several strategies, including copying collectors, generational collectors, and conservative collectors. We design our framework so that students can trivially change the type of collector their program uses (Figure 1).

- It *encourages exploration*: our framework lets student-written collectors manage memory for a regular programming language. Thus, they are not limited to the (much smaller) set of examples and tests they could write in a toy language. This is especially valuable

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'13, March 6–9, 2013, Denver, Colorado, USA.
Copyright 2013 ACM 978-1-4503-1775-7/13/03 ...$15.00.

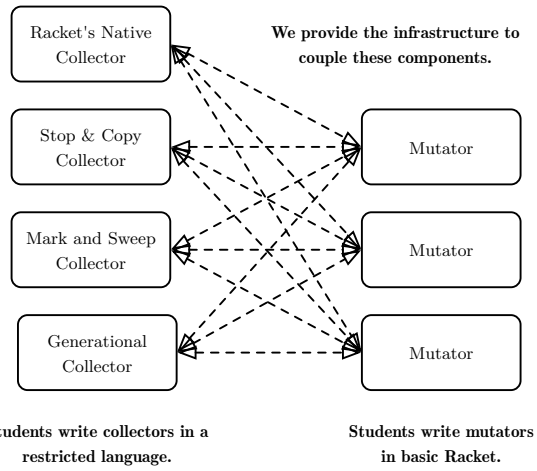

Figure 1: High-Level Assignment Architecture

when students believe they have a working collector and now want to stress-test it on non-trivial inputs.

- It *facilitates testing and debugging*: our framework can use a native garbage collector instead of a student-written collector. This allows the student to validate behavior and to isolate errors. In addition, students have powerful debugging tools as their disposal. We include a *heap visualizer*, and engineer our system so that the IDE's debugger allows students to *single-step between collector and program*.

In end-of-semester surveys, students indicate that whereas they had been apprehensive before, they are now much more likely to trust and rely on garbage collection as a result of their implementation experience. Beyond the pure pedagogic value, this could have a salutary impact on the software students write in their careers.

The solution described in this paper has already been implemented in actual courses at several institutions. Before adopting this solution, some of them used a more conventional approach that depended on compiling to continuation-passing style. Since switching to the new framework, all have noticed a marked improvement in the quality of students' solutions and the extent of their testing and exploration.

Background and Terminology.

We use standard terminology from the garbage collection literature. A language runtime's memory subsystem, or *collector*, has two principal duties. (1) It allocates and de-allocates values on the heap on behalf of programs, or *mutators*. (2) It provides facilities for reading and writing values and inspecting their types at runtime.

When *allocating* a new value, the collector first searches for a free slot of appropriate size on the heap. If found, it stores the value along with metadata, e.g., a *type-tag* for runtime type-inspection. *Garbage collection* usually starts if there are no free slots. GC de-allocates values that are not reachable by following references from local variables on the stack and in registers; these variables are called the *root set*. To follow references through arbitrary values, the GC inspects their type-tags. Accessing and mutating values is straightforward provided type-tags are respected; languages use a combination of static and runtime checks to do so.

The outline above encompasses several kinds of collectors, including mark-and-sweep, stop-and-copy, and generational collectors. Advanced techniques such as concurrent collectors are beyond the scope of our GC assignment.

Linguistic Dependence.

The work in this paper is done for Racket [3], a descendant of Scheme. We use Racket for two reasons. Historically, Scheme and Racket are widely used in definitional interpreter courses (thanks to several influential textbooks [1, 6, 9]). Much more importantly, Racket (but not Scheme) provides us with the linguistic mechanisms needed to implement this solution seamlessly from the student's perspective, through its cutting-edge ability to treat languages as libraries [5, 13]; other languages lack the module systems needed to enable such a solution. Readers who would prefer our solution in a different linguistic context can regard this paper as a technical challenge for the module mechanisms of their preferred language.

2. PROBLEM CONTEXT

Our garbage collection assignment, along with classroom discussions contrasting garbage collection with the cost of manual memory management and the impact of garbage collection on programming style, constitutes about one fifth of a roughly 13-week programming languages course for upper-level undergraduates and beginning graduate (master's and PhD) students. We ask students to implement memory allocation and garbage collection algorithms for a small number of Racket primitives. We assign multiple algorithms to avoid the risk of presenting too narrow a perspective: traditionally, *mark-and-sweep* (which later also provides a starting point for discussing conservative collection) and *semispace swapping* (because it confronts the challenge of moving objects in memory). Students are given about two weeks to complete these implementations.

We often grade the assignment by *code inspection*, in which students present their collectors to the course staff and answer questions about their design decisions. We discuss code inspection for GC in more detail in Section 6.

3. WRITING COLLECTORS

We provide a general framework that allows students to implement several kinds of collectors. The framework exposes essential concepts such as memory layout and roots on the stack. However, we hide lower-level machine details, including registers and the binary representation of data.

A student-written collector is a module that implements the interface in Figure 2a. These functions allocate values, query type-tags, and read values. The depicted functions only manipulate lists and flat values (i.e., numbers and booleans), though it is trivial to add support for other datatypes. For example, we also require students to allocate closures for higher-order functions.

These student-written functions determine the location, layout, and size of values on the heap. Therefore, students can experiment with collectors that add metadata (e.g., lifetimes for generational collectors) or don't (e.g., conservative collectors). Students can also explore simple optimizations, such as reusing allocated constants.

$$\begin{array}{rcl} \textbf{addr} & \in & 0 \leq n < (\textit{heap-size}) \\ \textbf{flat} & \equiv & \textbf{num} \mid \textbf{bool} \mid \textbf{empty} \end{array}$$

$$\begin{array}{r} \textit{gc:cons} :: \textbf{addr} \times \textbf{addr} \rightarrow \textbf{addr} \\ \textit{gc:cons?} :: \textbf{addr} \rightarrow \textbf{bool} \\ \textit{gc:first} :: \textbf{addr} \rightarrow \textbf{addr} \\ \textit{gc:rest} :: \textbf{addr} \rightarrow \textbf{addr} \\ \textit{gc:set-first!} :: \textbf{addr} \times \textbf{addr} \rightarrow \textbf{void} \\ \textit{gc:set-rest!} :: \textbf{addr} \times \textbf{addr} \rightarrow \textbf{void} \\ \hline \textit{gc:flat} :: \textbf{flat} \rightarrow \textbf{addr} \\ \textit{gc:flat?} :: \textbf{addr} \rightarrow \textbf{bool} \\ \textit{gc:deref} :: \textbf{addr} \rightarrow \textbf{flat} \end{array}$$

(a) Garbage Collector Interface (student-implemented)

$$\begin{array}{r} \textit{heap-size} :: \rightarrow \textbf{int} \\ \textit{heap-set!} :: \textbf{addr} \times \textbf{flat} \rightarrow \textbf{void} \\ \textit{heap-ref} :: \textbf{addr} \rightarrow \textbf{flat} \end{array}$$

(b) Heap manipulation (provided)

$$\begin{array}{r} \textit{get-root-set} :: \rightarrow \textbf{listof root} \\ \textit{read-root} :: \textbf{root} \rightarrow \textbf{addr} \\ \textit{set-root!} :: \textbf{root} \times \textbf{addr} \rightarrow \textbf{void} \end{array}$$

(c) Root manipulation (provided)

Figure 2: Generic Allocator Interface

Our framework provides functions to manipulate the heap as an array of cells (Figure 2b). The collector can store (*heap-set!*) and retrieve (*heap-ref*) **flat** values in heap cells—storing structured-data in a single cell would violate the intent of the assignment. The *heap-size* function returns the size of the heap, which is set by each mutator (Section 4).

Our framework provides functions to manipulate the roots on the stack (Figure 2c). A **root** is an updatable reference to the heap: updates are needed by moving collectors, such as stop-and-copy or compacting mark-and-sweep. We do not require students to walk the mutator's stack to find roots themselves; the **get-root-set** function provides this functionality. The next section explains why having students implement root-finding is neither essential nor desirable.

4. TESTING AND CALCULATING ROOTS

To truly exercise a collector we must use it to allocate memory for a program. In traditional courses, students test collectors by writing mutators in small, toy, classroom-only languages. It is difficult to write substantial tests in these toy languages. However, in our framework *students write mutators in Racket itself.* This enables students to write much more rigorous tests: student submissions have included non-trivial programs such as entire interpreters for other languages, and complex graph algorithms. In addition, by automating stack-walking, our approach *eliminates a curricular dependency* on program transformations.

The Traditional Approach.

In a traditional setting, such as a compilers course, writing a collector requires mastery of semantics-preserving program transformations. We sketch these transformations for

```
(define (incr-list lst)
  (if (empty? lst)
    empty
    (cons (+ 1 (first lst)) (incr-list (rest lst)))))
```

Use students' GC primitives

```
(define (incr-list lst)
  (if (and (gc:flat? lst) (empty? (gc:deref lst)))
    (gc:flat empty)
    (gc:cons (gc:alloc-flat (+ (gc:deref (gc:flat 1))
                               (gc:deref (gc:first lst))))
             (incr-list (gc:rest lst)))))
```

Name intermediate results (A-normal form)

```
(define (incr-list lst)
  (if (and (gc:flat? lst) (empty? (gc:deref lst)))
    (gc:flat empty)
    (let* ([t0 (gc:flat 1)]
           [t1 (gc:first lst)]
           [t2 (gc:flat (+ (gc:deref t0) (gc:deref t1)))]
           [t3 (incr-list (gc:rest lst))])
      (gc:cons t2 t3))))
```

Figure 3: Sketch of Program Transformations

a simple mutator that increments the numbers in a list (top of Figure 3).

Given a mutator, such as *incr-list*, students have to first instrument it to call their collector at memory-manipulation points. Although this first transformation is conceptually simple, it is orthogonal to GC algorithms and quickly becomes complex for non-toy languages. In addition, the collector must save the state of the mutator, collect garbage if necessary, perform the memory operation, and then restore the state of the mutator. Implementing these steps in a low-level language requires considerable effort and knowledge.

Students must then transform the mutator to an intermediate form that names all sub-expressions. Such representations (e.g., continuation-passing style, static single assignment, or A-normal form) are advanced topics, tricky to implement, and tedious for even modestly-sized languages.

As a result, a traditional approach incurs several curricular dependencies and is only realistic for a toy, classroom-only mutator language.

Our Approach.

Our framework automates the transformations in Figure 3 for a large subset of the Racket language, eliminating the curricular dependencies described above, and allowing students to test and explore their collectors by writing mutators in ordinary Racket. Our students are familiar with Racket, since it is the language they use to write the collector itself! Mutators do have one non-standard line, which specifies the collector to use and the size of the heap:

(**collector-setup** ⟨*collector-filename*⟩ ⟨*heap-size*⟩)

By varying the GC algorithm and heap size, students can test for more corner-case behaviors.

Finally, our framework allows students to easily run sequences of collector functions on hand-crafted heaps. This allows them to unit-test with complex heaps without crafting complex mutators. To help students further, we provide a program that generates mutators that allocate and traverse random graphs.

5. DEBUGGING

Despite the efforts described above, building a collector remains subtle and error-prone. Bugs often corrupt the heap, and heap corruption may only cause errors and incorrect answers several steps later. Students thus need good debugging support. Building a debugger in this context is a challenge, because bugs manifest in the interaction between collectors and mutators. We thus need to *co-debug the mutator and collector*, set breakpoints in either piece, and single-step between them.

The Traditional Approach.

In a traditional setting, the mutator is first compiled to a low-level, intermediate representation, and then coupled with the collector, which is also written in a low-level language, such as C. A debugger for this low-level language (GDB) can debug both components. However, without extensive tooling, the debugger can only show the mutator's intermediate form, so students have to fully absorb the very program transformations we were trying to hide.

Our Approach.

Figure 4 shows our framework co-debugging a mutator and collector. The collector has a breakpoint at the start of GC. We are paused a few steps beyond this breakpoint, which we reached because the heap filled up after a few recursive calls in the mutator. Note that the stack shows both mutator and collector stack frames. Most significantly, we can debug both in their original source language.

Our approach reuses Racket's native debugger [10] by carefully transforming the mutator using Racket's macros. The debugger works at the original source level, and does not step into intermediate expressions introduced by macros. This is possible because macros can attach original source locations and variable names to generated code. Simple macros can track names and locations automatically, but our more complex, whole-program transformations require more careful engineering. In particular, we are careful to leave the shape of the control stack unmolested by our macros.

Heap Visualization.

Debugging is not enough. Not surprisingly, we noticed that in practice, students have as much difficulty interpreting their data as they do debugging the flow of control. Specifically, we found that they repeatedly printed the entire heap, often as frequently as before and after every allocation (to answer questions like, *What exactly is at location 14?*). They then had to pore over the output to reconstruct its meaning as a heap of values.

While such activity builds character, it may not fit the constraints of some teaching schedules. Fortunately, we can provide a much better interface for this domain. Keeping the heap data structure under the control of our code—a decision made to guard the size of and kinds of values stored in the heap—has a (perhaps unexpected) advantage: we can find the heap without difficulty (whereas otherwise it would be just another datum in the collector code) and employ the student's heap-inspection routines to display it.

Figure 5 shows the heap at the point shown in Figure 4. From this, it is relatively easy to reconstruct what datum is at location 27.

Naturally, the visualizer and debugger are especially useful in conjunction. For instance, it is often convenient to avoid stepping through code that sets up initial data structures in the mutator. The student can thus set a breakpoint in the mutator past this stage. Upon control arriving there, the student can then enable breakpoints in the collector, or use single-stepping, to proceed methodically through the program's execution—using the visualizer to avoid having to manually inspect the heap at each point along the way.

6. PEDAGOGY

Implementing collectors forces students to confront intricacies of the problem that would be difficult to convey through an abstract verbal treatment alone. To evaluate student understanding we often augment the implementation assignment with a *code inspection*, in which students explain their programs to the course staff, then answer questions about their design decisions and how they might handle potential extensions. Our questions include:

basic concepts How did they represent data on the heap, and why? Did they inline the free list in mark-and-sweep? What are forwarding pointers in stop-and-copy, and what purpose do they serve? Is their collector realistic, or have they over-simplified the problem by exploiting high-level features, or even cheated (e.g., ignoring the provided heap interface and relying instead on Racket's collector)?

representations What representations did they choose for flat values? Did every mention of a flat value result in fresh allocation? Did they reuse existing allocated values (and if so, what are the time-space tradeoffs)? Did they pre-allocate any values? Did they make intelligent use of addresses to encode some values in the address itself and, if so, how did this affect their ability to address all of the heap?

cycles How does the collector handle cyclic data? How does it avoid getting stuck in an infinite loop if it traverses data with cycles? Do we need to check for sharing even in programs that do not create cyclic data, and what impact does this have on a copying collector?

subtleties What happens if the collector needs to run in the middle of a recursive procedure? Could the collector run out of memory in the middle of allocating an object? Is the collector idempotent?

testing Did they have test cases to exercise salient features of the language? Did they try nested data structures, such as lists of lists? The assignment does not provide a canonical mutator, so students are forced to exercise their imagination.

extensions What would it take to add support for coalescing adjacent free blocks in the mark-and-sweep collector? How difficult would it be to support other kinds of objects?

invariants What invariants hold for the mark-and-sweep collector? What about semispace swapping?

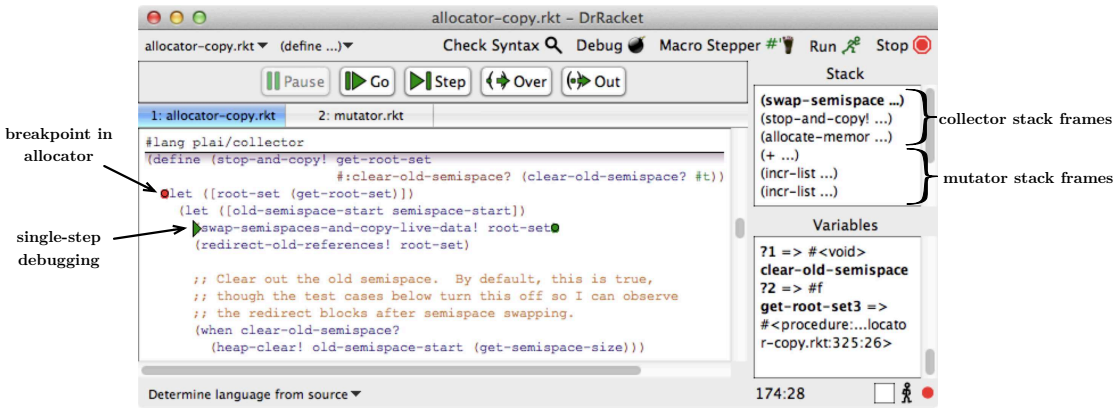

Figure 4: Co-debugging a collector and a mutator

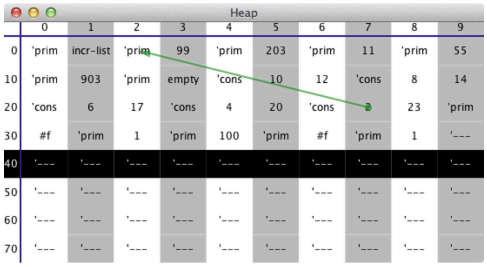

Figure 5: Heap visualization: a filled semi-space

By having a concrete implementation in front of us (that the student has hopefully thought long and hard about), we can ask specific, detailed questions that would otherwise be difficult to frame. Code inspection thus gives us a very strong sense of the student's understanding of the concepts.

Student Feedback.

Many faculty have reported success using this framework. For example, one author, Jay McCarthy, surveyed 258 students over 4 years. Before doing the assignment, 40 students agreed with the statement, "Manual memory management is usually preferable to GC," but only 10 agreed after the assignment. Before the assignment, 80% believed that manual memory management is always faster than GC, but only 40% agree after; the others answer that GC is as fast or faster, given enough memory. The students were also asked what part of the class they learned from and enjoyed the most. Of the 258, only 15 mentioned anything other than GC.

7. IMPLEMENTATION

Our GC framework could be built for any programming language, but Racket provides a combination of several technologies that ease its implementation [13]. Whereas syntactic extensions are part of the Scheme tradition, a Racket language can have entirely different syntax and semantics. Our GC framework is an extreme example of the latter: the semantics of the mutator language changes as students fix bugs in their collectors. The implementation is a separate research contribution beyond the scope of this paper. But, we note some key techniques below.

Collector The collector language provides a heap abstraction to students. The language itself ensures that students do not store structured data directly on the heap using contracts [4]. In addition, the collector language itself starts the graphical heap visualizer.

Mutator The mutator language implements the transformations that students have to learn in traditional GC courses (Section 4). Throughout these transformations, we ensure that generated code is carefully tagged with the source-location of original code. This ensures that runtime errors have correct locations and that the transformations are transparent to the debugger [10].

The mutator language uses students' collectors instead of Racket to manipulate memory (e.g., it rewrites *cons* to *gc:cons*). If done naively, e.g., with macros, we would have to instrument all syntactic forms (that we remember). Instead, we use Racket's ability to precisely control macro expansion: we first transform mutators to primitive syntax and then instrument code. This ensures that students can write mutators with convenient syntax.

Mutators must expose their root set to collectors, which we do with an internal *roots* variable at each program point. To calculate *roots*, we first name all intermediate expressions and then calculate free variables at each program point. By doing so, as Figure 6 illustrates, the set of free variables and the root set coincide *for a single stack frame*. We also augment all functions with an additional *caller-roots* parameter. Thus at each point, the root set is the union of *caller-roots* and the local *roots*.[1]

8. RELATED WORK

We surveyed faculty who teach implementation-based languages courses, as well as experts in the field of garbage collection, and none of them appear to use an infrastructure similar to ours. The only assignments we found involving implementation of garbage collection were in courses on compilers and run-time systems.

At the University of Victoria, Nigel Horspool's course on virtual machines treats GC in depth [7]. Students study the

[1]N.B., we use *continuation marks* to obviate parts of this transformation and preserve tail-calls [2].

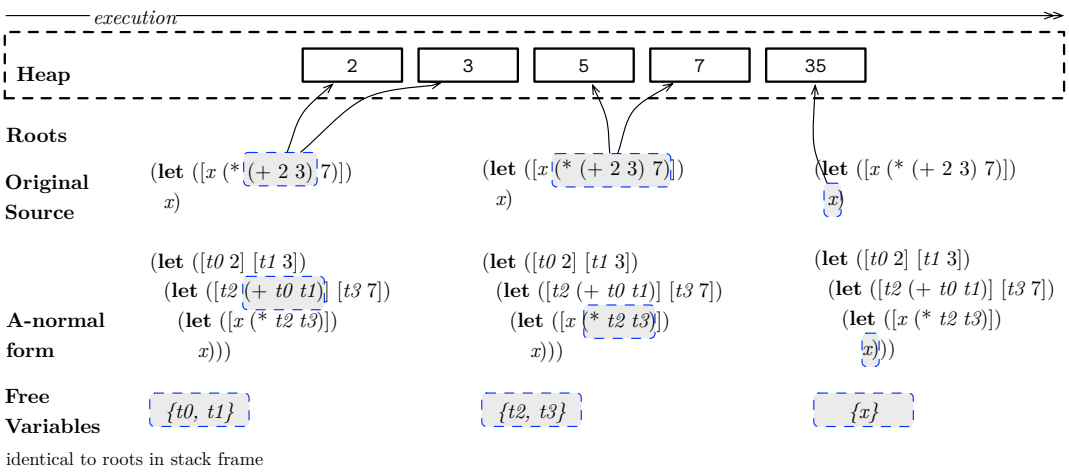

Figure 6: Naming intermediate results allows us to track roots

implementation of a conservative mark-and-sweep collector inside a JVM, then answer questions pertaining to design decisions. Students do not implement collection algorithms, but they instrument the collector to obtain statistics.

At Cornell, Greg Morrisett's course on data structures and functional programming [12] included an assignment to implement parts of a compiler and runtime system for a subset of ML, including generational GC. Programs are first compiled to lambda terms, which are evaluated by an abstract machine. The course framework specifies the heap layout and handles allocation; students only implement collection. At Princeton, David Walker assigned a similar project that had students write a semispace collector for a simple, functional language interpreter [14]. All the interfaces between the elements of the system (memory layout, stack, heap, collector) are spelled out explicitly in the module system. Abelson and Sussman detail the implementation of another semispace collector for a register-based Scheme evaluator [1].

The main difference between all these assignments and ours is their use of explicit compilers, interpreters, and abstract machines, which expose a lower-level model of the runtime system, encouraging a more systems-oriented view of garbage collection. Our approach involves fewer curricular dependencies and allows the students to focus on the collection algorithms. Our unobtrusive transformation strategy allows students to write mutators in a full-fledged language and exploit graphical debugging and testing features in the DrRacket IDE. These variations reflect a difference in course content and philosophy; our approach seems advantageous for instructors interested in covering garbage collection without assuming much low-level background or context.

Acknowledgments

We thank our students for enduring GC, Matthew Flatt for making Racket awesome, and the NSF for partially supporting this work.

9. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] J. Clements and M. Felleisen. A tail-recursive semantics for stack inspection. In *ESOP*, 2003.

[3] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *JFP*, 12(2):159–182, 2002.

[4] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[5] M. Flatt. Composable and compilable macros. In *ICFP*, 2002.

[6] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, second edition, 2001.

[7] N. Horspool. Topics in virtual machine implementation. `http://www.cs.uvic.ca/~nigelh/Courses/csc586a-2008/`.

[8] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[9] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. 2006. `http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/`.

[10] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *ASE Journal*, 14(1):59–86, 2007.

[11] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *CACM*, 3(3):184–185, 1960.

[12] J. G. Morrisett. Mini-ML Compiler. `http://www.cs.cornell.edu/Courses/cs312/2002fa/hw/ps5/ps5.html`.

[13] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, 2011.

[14] D. Walker. Garbage collection programming assignment. `http://www.cs.princeton.edu/courses/archive/spr05/cos320`.