A Framework for Memory-Management Experimentation

Stephen P. Carl
Department of Mathematics and Computer Science
The University of the South, Sewanee, Tennessee

Abstract

Phobos is a framework for experimenting with memory management systems. This framework provides two types of operation – profiling program allocation behavior and simulating the actions of memory management systems. Profiling is used to generate data about a program's allocation behavior including total memory allocation and memory object lifetimes. Simulation is used to measure the performance of different memory management strategies on particular program runs. In both cases, Phobos takes its input from a *trace file* generated during execution of a targeted application which lists the memory events of interest.

This paper describes the design of the Phobos system. In particular, it shows how the system takes advantage of the code structuring facilities provided by PLT Scheme, highlighting the use of signed units, mixin classes, and other features of this system.

1 Introduction

We are developing Phobos, a framework for studying memory management systems. Most popular functional languages, such as Scheme [15], and object-oriented languages, such as Java [1], use some form of *garbage collection* to implement automatic memory management [13]. While there are a number of garbage collection algorithms, most systems today have some form of *generational* collection available. Some languages best known for scripting capabilities, such as Perl [24] and Python [23], use *reference counting* systems for automatic memory management, while implementations such as Jython [14] benefit from advances in the Java language runtime. Our goals for Phobos include classifying the memory allocation patterns of different types of applications and determining how well or poorly different memory management algorithms interact with these patterns.

The framework has two modes of operation: *profiling*, which can be used for studying the allocation behavior of applications, and *simulation*, which can be used to determine how well different garbage collection algorithms perform when matched with applications exhibiting these behaviors. Information about an application's use of memory comes from *memory trace files*, which contain information about object allocations, object deallocations (for profiling), pointer stores and pointer reads (for simulation), and so on. Trace files are produced by instrumented virtual machines or interpreters which log events of interest as they occur.

This paper describes the design of Phobos and simple examples of how it is used. The framework design is based on the program structuring features provided by PLT MzScheme, a R^5RS-compliant Scheme implementation featuring a number of useful extensions including a fully integrated module system, units for creating separately-compilable components, and a Java-like class system which supports *mixin-based* programming [5]. We describe how use of these language features helped create a system capable of specifying various simulator configurations from information provided by the user.

2 Simulator Design

2.1 Design Goals

Phobos has been designed with the following goals in mind:

- **Experimental Control.** The user controls the system through a script for specifying experimental parameters such as heap size, input format, memory management components used in simulation, and statistics to collect.
- **Prototype Development.** Implementations of new memory managers can be prototyped to quickly explore the design space.
- **Language independence.** While initial experiments are targeted at Java programs, other languages can be accomodated by providing an execution environment instrumented to produce trace files.
- **Extensibility.** Phobos can be extended to handle new types of trace file formats, new memory management algorithms, and new types of statistics to collect.

2.2 Structuring the System

The framework is divided into two sets of modules: the *engine* and the *simulation components*. The engine is made up of the main driver, handlers for reading trace files, and an interface for experiment scripts. Simulation components are defined as MzScheme classes and represent the heap and heap partitions, basic memory

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

management systems, and statistics generators.

The engine and components are combined according to information in an experiment script which is written by the user to control the simulator. Information from the script tells the system how to combine the main driver with the functions which read a specific trace file format. The script further specifies the memory management functionality to be used; the script interface determines which classes from the simulation components are needed and loads them at startup time. The memory manager is made by combining classes encapsulated in *units* that represent allocation and collection systems with the heap classes. A detailed description of how and why units are used in our framework is given in Section 5.

Trace handlers are functions which deal with specific memory trace events given in a trace file. The framework currently supports three formats, one for manual memory management, one for a JVMPI-compliant profiling agent for Java programs [22], and one for general simulation. New formats can be accomodated by developing a new set of handler implementations. As each memory event is read from the trace file, the driver transfers control to the appropriate handler function for checking that the event is in the proper format and creating an object that represents the trace. Such objects in turn cause some change in the heap by sending it a message corresponding to the event type (e.g., object allocation, object deallocation, pointer store, etc.).

When profiling, the heap is used to simply store allocated objects, then remove them when deallocated, while maintaining a set of counters which track basic statistics. When doing simulation, the heap object sends messages to one or more *heap frames* which it manages. A heap frame is conceptually just a range of addresses in (simulated) memory, coupled with a specific set of methods for handling allocation or collection; these impose a logical organization on the heap frame. This organization allows us to model monolithic heaps which use a single allocator and collector, and also heaps that are partitioned into regions which are managed differently, such as in generational garbage collectors.

The simulation components that report statistics log basic information in each memory object allocated. The system produces two files of raw data. The **log file** contains allocation time,[1] size, and deallocation time for each object allocated. The **lifetimes file** contains the *lifetimes* of each object, where lifetime is the difference between deallocation time and allocation time. The information in these files is suitable for processing by external tools such as Matlab for producing graphs, such as object type and lifetime distributions.

3 Profiling Applications

The simplest use of the system is to profile the memory usage of an application. A trace file to be used for profiling records the actions of the memory manager used by the system which executes the application, including GC start and stop events, per-object allocation and deallocation events, and object copy events. In profile mode, Phobos simply replays these events in a *shadow heap* and tracks information about each object and the events which affect them. It also computes statistics about each object when they are deallocated or the trace file ends, and displays the amount of heap space needed by the application being profiled, the total amount of allocation both in number of objects allocated and size of memory used, and the number of garbage collections required including the number of objects and total memory collected at each.

The following simple script runs a single profiling experiment:

```
(experiment
 (connect 'PROFILE "~/traces/robo-trace"))
```

The `connect` form specifies the trace format type and the full pathname to the trace file. When an experiment run begins, the system selects the trace handlers to use based on the trace format type (though this can be overridden in the script) and combines these with the main driver and the simulation components used for profiling. When invoked, the driver first attempts to open the trace file specified, and then calls the trace handler functions to read and respond to memory events stored in the trace file.

Once the trace file processing is completed, the system displays the global heap statistics it has computed:

```
For this trace file run:
Total types recorded: 1022
Total amount of allocation: 98993720 bytes
Total number of objects allocated: 2350147
Total number of objects collected: 2237897

4786686 Events Processed
```

Also, the statistics component produces the object lifetimes and allocation log.

4 Simulating Memory Managers

In simulation, the trace file produced when an application is run contains allocation events, pointer update events, and enough information about the execution to drive the actions of a simulated memory management system. A typical simulation script describes each experiment to be run, including the trace file to be used, the memory management system (or systems) to simulate, and the characteristics of the simulated heap.

For example, the following script runs a single experiment using a trace file `robo-trace` on a typical heap managed by a best-fit allocator and mark/sweep garbage collector:

```
(experiment
   (connect SIM "~/traces/robo-trace")
   (base-heap 64 0 (allocator first-fit)
                   (collector mark-sweep)))
```

The simulation is again driven by the memory trace file in the path specified in the `connect` form. The second part of the experiment script specifies the characteristics of the heap: size is 64 Mbytes, the base of the heap (conceptually) starts at address 0, and it is managed by a first-fit allocator and a mark/sweep garbage collector. This script runs the experiment and reports the statistics just as in the profiler.

The names `first-fit` and `mark-sweep` are predefined. In general, the names are used to choose the module in which the particular class definition implementing these algorithms is found. For example, `first-fit` is defined in a module in the file **first-fit.scm**, which provides a class that implements the first-fit allocation policy. Actually, the algorithms are defined as *mixin classes*, and what is provided is a function which creates a new class representing a

[1] Per the GC literature, allocation time is measured in bytes allocated so far – the first object is allocated at time 0, the next is allocated at time 0 + (size of first object), and so on.

heap frame extended with the mixins. Section 6 describes this in more detail.

Experimental setups can be defined and saved singly or in groups. Saving an experimental setup by associating it with a name allows results to be labeled with experiment names rather than with experiment characteristics. In the example shown in Figure 1 we define a set of experiments, called an experiment *suite*, which will be run one after the other by the simulator. The suite is defined by the form `define-experiment-suite`.

5 Under the Hood

In MzScheme, a *unit* is a separately compilable component which can be linked to other units to create a (as yet unevaluated) program. Units may import external variables which are used in the body of the unit, and may export its own variables to be imported by other units. When the list of variables imported or exported is long, *signed units* are used instead as a convenience. In this case, the programmer provides *signatures* to specify those names to be exported to other units, and import using other unit's signatures. The program formed by linking units is evaluated when *invoked* with the `invoke-unit` (with regular units) or `invoke-unit/sig` forms (with signed units).

In Phobos, evaluating an experiment script selects those units whose code is to be used to handle specific types of traces and construct the simulated heap. Scripts are processed by passing the script file name on the MzScheme command line. The script is then evaluated as Scheme code, using the definitions provided by Phobos. The `experiment` form is a macro which uses the information in its body forms to select the appropriate units (representing the engine and required components) and link them together to create a *compound unit* at runtime. Invoking this compound unit starts the main driver.

The simple profiling experiment script shown in Section 3 elaborates to the `let*` form shown in Figure 2. This code creates and invokes a compound-unit out of a set of signed units; the signatures (defined elsewhere) are given by the symbols which end with a caret (^). The compound-unit is the result of linking the individual units `htprof-handlers@`, `unit-heap@`, `sim-driver@`, and `exp@` together. The first three of these are loaded into the system at runtime by a form (elided in the figure) called `dynamic-require`.

The unit `exp@`, which comes first in the `let*` form, defines the "command line" for the engine, using parameters from the script. The `compound-unit` form then links the units which define the specific trace handlers used by engine together with the simulated heap, the driver unit, and the unit which defines the command line. The driver unit exports the procedure name `sim-driver` used by the new unit `exp@`. The link step returns the compound unit `prg@` that is invoked in the `let*`. Invoking the compound unit has the effect of calling `sim-driver` and running the simulation in profiler mode. The results produced will be labeled with the experimental characteristics, that is, the name of the trace file, the memory manager used, and the heap size and layout.

6 Structure of the Simulated Heap

The simulated heap is made up of one or more heap frame objects. For modeling monolithic heaps one heap frame is sufficient. However, in modern runtime systems heaps tend to be partitioned. Generational garbage collectors partition by age; other recently proposed systems partition by type [17] or by connectivity [9]. To model these systems, each heap frame represents a different *partition* of the heap. Global attributes of the heap are captured by the `heap%` class, which also holds the first heap frame. The structure of the definition (minus method code) is as follows:

```
(define heap%
  (class* object% (heap<%>)
    (init-field
     ;; an object that collects statistics
     stats
     ;; default heap size is 32 Mbyte
     (initial-size (expt 2 25))
     (max-size initial-size)
     (alloc-frame
       (make-object heap-frame% initial-size)))

    (field (bytes-allocated 0))
    (field (total-allocated 0))
    (field (total-objects 0))

    (define/public (allocate trace)
      ;; updates global properties of the heap
      ;; sends allocate message to alloc-frame
     ...)

    (define/public (deallocate object-id)
      ;; updates global properties of the heap
     ...)
     ...)))
```

This definition creates the class `heap%` consisting of a set of fields (three defined by the `init-field` form and four by the `field` forms) along with a set of methods (only a subset of the class methods are shown). The field `alloc-frame` refers to the initial heap frame. Each heap frame refers to the "next" heap frame in the system. For example, to define a semi-space copying collector, two frames are used, each refering to the other. For generational collectors, each generation is a separate heap frame which each refer to the succeeding generation in the system.

Each heap frame is defined by a class `heap-frame%` which includes methods for allocating blocks, collecting unreachable objects, and handling pointer reads and writes as shown:

```
(define (heap-frame% %)
    (class* % (heap-wrapper<%>)
      (inherit store! lookup remove!)
      (rename (super-terminate terminate))
      (init-field
        initial-size
        (next-frame '()))
      (field (frame-bytes-allocated 0))
      (field (start-addr 0))
      (field (end-addr (- initial-size 1)))
      (field (roots '()))

      ;; Method Declarations
      (define/public (allocate-slot obj size)
       ;; allocates the next available block
       ;; large enough to store obj of given size
        ...)

      (define/public (collect-slots)
        ;; dummy collector
```

```
(define-experiment-suite gc-suite
   "run experiments on two tracefiles"
   (experiment
     (connect SIM "~/traces/robo-trace")
     (base-heap 32 0 (allocator first-fit)
                     (collector mark-sweep)))

   (experiment
     (connect SIM "~/traces/kaffe-trace")
     (base-heap 32 0
       (partition (name nursery 16)
                  bump-pointer
                  (copy-promote (partition (name old 16) best-fit mark-sweep))))))

(simulate gc-suite)  ;; kicks off experiments
```

Figure 1.

```
(let* ((exp@ (unit/sig () (import sim-driver)
               (sim-driver "/traces/robo-trace" JVMPI (expt 2 24))))
       (prg@ (compound-unit/sig (import) (link [HANDLE : trace-handlers^ htprof-handlers@]
                                               [SIMHEAP: unit-heap^ unit-heap@]
                                               [DRIVER : sim-driver^ (sim-driver@ HANDLE SIMHEAP)]
                                               [RUN    : () (exp@ (DRIVER sim-driver))])
                                  (export))))
  (invoke-unit/sig prg@))
```

Figure 2. Elaboration of `experiment` form

```
       ...)

     (define/public (read addr)
       ;; pointer read
       ...)

     (define/public (write addr ptr)
       ;; pointer write
       ...)))
```

The default heap frame object implements the `NoGC` storage manager, which creates new objects in the next available chunk of memory and removes objects without making the newly-freed space available for future allocations. More useful allocator and collector mechanisms are provided in the form of *mixin classes* which extend `heap-frame%` by overriding the `allocate-slot` and `collect-slots` methods. The `read` and `write` functions can also be overridden for implementing read or write barriers as needed. When a specific type of heap manager is chosen for simulation, the heap frames are created by choosing appropriate allocator and collector subclasses, creating extensions by mixing these in, and then instantiating the resulting classes.

This organization is accomplished as follows: a *mixin* is created in MzScheme by defining a class whose superclass is specified as a parameter, using the `define` form. For example:

```
(define (make-mixin super-class)
  (class super-class ...extension...))
```

The actual class is created by calling the resulting procedure and passing in the name of the superclass to be extended.

There are two main benefits of using mixin classes in this system. First, allocators and collectors can be combined independently as long as they are compatible (for instance, the system will generate an error when processing a script which pairs a non-moving allocator with a copying collector). Second, when placed in their own units, mixin extensions can be selected and combined to form a single unit representing the simulated heap by importing the actual superclass at link time.[2] This allows us to essentially create different heap frame classes on the fly, combining them to form a multiple-partition heap where each partition is managed using a different strategy.

An allocation event only affects a single heap frame. Each allocation algorithm is defined in a separate unit as a mixin class which contains at least the method `allocate-slot` as shown in this example:

```
(define (bump-pointer super%)
  (class super%
    (init-field
      size
      (pointer 0))
    (define/override (allocate-slot trace)
      ;; defines new allocator
     ...)))
```

The procedure `bump-pointer` takes an argument `super%` which is the superclass of the mixin. The `allocate-slot` method overrides that defined in the superclass. This mixin defines the "bump pointer" allocator (also known as fast allocation) which reserves the next free address in the heap frame for the object being allocated. The `init-field` form defines two fields; `size` is the maximum size of the heap frame, and `pointer` tracks the next available position in the frame.

Collectors are created in the same way. A particular collector component overrides the method `collect-slots` and can include any

[2] More information on the use of units and mixins in MzScheme can be found in Findler and Flatt's ICFP'98 paper [4].

other supporting methods or fields necessary. The collector components will be combined with some superclass (again, not usually known in advance) using the same mixin style as with allocators. In general, the `collect-slots` method is called by the allocator when there is no more space available in the heap frame or some threshold size is reached.

Once defined, units for allocators and collectors become part of a library of components to be used in experiments. The name of the module which defines a specific component is given in the experiment script which selects the proper units and evaluates the procedures for each mixin class. When evaluated, these procedures generate a new class which will extend either `heap-frame%` or some subclass of it. The actual superclass does not have to be known in advance. In this way, classes representing the simulated heap are created on the fly based on the experiment script.

7 Memory Management Components

In this section we cover some of the forms used in Phobos experiment scripts to generate the memory management classes. New components are being added as the system matures. Components are in general added by writing mixin classes built along the same lines as `bump-pointer`. In more advanced cases, new macro forms may be required.

7.1 Allocators

The `allocator` form specifies the unit to be used for allocation. Examples of allocators currently available include the default `bump-pointer`, simple `first-fit` allocation, and the more advanced `seg-freelist` for implementing a segregated freelist allocator. An allocator is specific to a single heap frame.

7.2 Collectors

The `collector` form specifies a unit to be used for garbage collection. Collectors defined using this form are generally used to manage a single partition in the heap. The `mark-sweep` and `mark-compact` collectors are two examples of collectors which can be used with this form.

7.3 Partitioned Heaps

The form `partition` allows the user to define the way the heap is divided into heap frames, usually for copying collectors. This form has the following structure:

```
(partition (name <identifier> <size>)
           <allocator>
           <collector>)
```

The `name` subform is optional and associates an identifier with the partition as well as its size (in Mbytes). If `name` is not used, the size is specified there instead. The `<allocator>` and `<collector>` parameters are either the unit names as described before, or one of `copy-to` or `copy-promote`, each of which specifies copy collection between partitions.

The form of `copy-to` is (`copy-to` *size*), where *size* gives the size of a second partition, or semispace. This form is used to create a two-semispace copy collector. Elaboration of this form generates two heap frames which refer to each other via the `next-frame` field.

The form of `copy-promote` is (`copy-promote` *partition-form*), where *partition-form* is another partition declaration. This is generally used to create a generational memory manager, though provision for promoting based on criteria other than age is planned. The second partition form is the "older" generation which receives copies of objects which survive collections of the original partition. Note that partitions defined in `copy-promote` can themselves declare `copy-promote` as their collector (and so on) to generate more than two generations. Currently the form uses a default remembered set write barrier to catch intergenerational pointers.

8 Future Work

8.1 Instrumenting New Implementations

The Phobos framework was originally conceived as a tool to gauge the allocation characteristics of functional languages designed to compile to the Java Virtual Machine [20]. This is one reason why our current set of trace files are generated by executing Java programs. In the future, we would like to conduct experiments with trace files generated from progams executed directly by implementations of Scheme and other languages. This will require modifying existing execution environments to generate information about the memory events of interest.

The Garbage Collection website [12] includes a small repository of memory traces, which is intended to eventually represent many traces from applications written in different languages. The research community has apparently been slow to contribute to this repository; we would like to contribute to it soon and encourage other researchers to do so.

8.2 Visualization

Currently all visualization of data generated by Phobos is done using Matlab to generate graphs. It would be nice to have a set of tools for presenting interesting views of the allocation behavior of the programs and the performance of the memory managers. We will be evaluating other tools specifically aimed at graphing (such as PLTplot [6]), and profiling (such as EVOLVE [25]).

8.3 Developing New Managers

The memory manager components defined in Phobos are useful for studying the behavior of commonly used systems. The scripting system needs to be more flexible, however, if newly proposed and researched systems are to be implemented using this approach. In particular, the `copy-promote` form needs to be modified or complemented so that alternative write-barriers can be specified as well as different criterion for promoting objects.

Researchers who develop improved memory managers may want to develop prototypes to study their high-level performance on application traces. While writing allocators and collectors in Scheme can be an enjoyable exercise, we would like to develop memory management components in an *embedded language* built to work directly with the experiment scripting facility. We would like to try to develop such a "little language" [16] to aid in the process of building up the framework. The more allocators and collectors added to the library of components, the more experience we will have to better understand the abstractions and interfaces that the language must support.

9 Related Work

Simulators are used to study both the allocation behavior of specific applications and the behavior of memory management techniques. One of the first described systems was MARS, the Memory Allocation Research Simulator, described in Ben Zorn's dissertation [26]. This simulator was attached to a running LISP system and allowed the user to study the impact of using different (simulated) garbage collection algorithms with a set of applications. Zorn also proposed using a language specific to this domain for describing management systems to be simulated, but did not define one himself. To our knowledge this has not yet been attempted.

Simulation is also a key component in the work of Darkovic [19], who studied age-based (generational) collectors in the context of Smalltalk and Java, and Hansen [8], who studied older-first generational collectors in the context of Scheme.

Hölzle and Dieckmann developed a trace-driven simulator to provide data about the memory behavior of Java programs to the garbage collection research community [3]. The system was driven by memory traces generated from applications in the SPECjava98 benchmark suite. The simulator generated data for computing statistical information about object lifetime distributions, size variations, and the amount of heap space required to run each program. Few if any such simulators have been made publically available to the research community.

In the Jikes^{TM}Research Virtual Machine, new memory management mechanisms can be implemented directly (not simulated) by subclassing a set of provided Java GC classes which provide the base garbage collector. This allows the programmer to experiment with and determine the effects of different managers on an application or set of applications directly, without the need for generating trace files. However, the entire virtual machine must be rebuilt (a lengthy process) before testing a new manager [10].

Beltway is a framework built on top of Jikes which generalizes copying garbage collection, such that each of semispace, generational, and older-first collection schemes can be defined in a common framework [2]. The simulated heap is divided into some number of partitions called *belts*. Each belt is made up of a number of *increments*; an increment is the unit of allocation. Varying the size and number of belts allows the user to construct any existing copying collector, or create entirely new ones. Furthermore, the system supports partitioning objects in the heap by size, type, or call-site, so several different object characteristics can be exploited at once.

10 Conclusion

This paper has described the design of a framework for profiling the memory allocation behavior of applications and simulating memory-management systems. The framework uses program-structuring features provided by PLT Scheme to build representations of the simulated heap from components chosen by an experimenter at runtime. The use of units to compartmentalize code, specify import and exports in a disciplined way, and link components at runtime makes it possible to specialize the system based on an experiment script.

This approach has given us the ability to build simulations for a number of popular memory managers. It is not clear, however, that building components in this way will be useful for alternative designs currently being researched. It may turn out that some designs may be more difficult to render given the high level of abstraction represented by signed units and mixin classes. But for systems implemented thus far, the approach has allowed us the flexibility of developing components for different allocation and collection mechanisms and make them available to the simulator.

11 References

[1] Ken Arnold and James Gosling. *The Java^{TM} Programming Language, 2nd Edition.* Addison-Wesley, 1998.

[2] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting Around Garbage Collection Gridlock. Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation. In *SIGPLAN Notices.* Vol. 37, No. 5, pp. 153–164, May 2002.

[3] Sylvia Dieckmann and Urs Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. Technical Report 1998-33, UCSB Computer Science Department. December, 1998.

[4] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. Proceedings of the International Conference on Functional Programming (ICFP '98). In *SIGPLAN Notices*, Vol. 34, No. 1, January 1999.

[5] Matthew Flatt. *Programming Languages for Reusable Software Components*. Ph.D. thesis, Rice University, June 1999.

[6] Alexander Friedman and Jamie Raymond. PLoT Scheme. Fourth Workshop on Scheme and Functional Programming, November 7, 2003, Boston, MA.

[7] Richard P. Gabriel, *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

[8] Lars T. Hansen. *Older-first garbage collection in practice*. Ph.D. thesis, Northeastern University, November 2000.

[9] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *The 2002 International Symposium on Memory Management (ISMM 2002)*, pp. 36–49, Berlin, Germany, June 2002. ACM Press.

[10] Jikes Research Virtual Machine from IBM. `http://www.ibm.com/developerworks/oss/jikesrvm`.

[11] Richard Jones' Garbage Collection Bibliography. `http://www.cs.kent.ac.uk/people/staff/rej/gcbib`.

[12] Richard Jones' Garbage Collection Pages. `http://www.cs.kent.ac.uk/people/staff/rej/mtf/traces`.

[13] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[14] Jython Homepage. `http://www.jython.org/`

[15] R. Kelsey, W. Clinger, and J. Rees (Eds). The Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, September 1998.

[16] Olin Shivers. A universal scripting framework, or Lambda: the ultimate "little language." In *Concurrency and Parallelism, Programming, Networking, and Security,* Lecture Notes in Computer Science #1179, pages 254–265, Editors Joxan Jaffar and Roland H. C. Yap, 1996, Springer.

[17] Y. Shuf, M. Gupta, R. Bordawekar, and J.P. Singh. Exploiting prolific types for memory management and optimizations. Proceedings of the 2002 SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02). In *SIGPLAN Notices*, Vol. 37, No. 1, pp. 295–306, January 2002.

[18] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0*. August 1998. `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`.

[19] Darko Stefanovic. Properties of Age-based Memory Reclamation Algorithms. Ph.D. thesis, University of Massachusetts, February 1999.

[20] Robert Tolksdorf. Languages for the Java VM. `http://www.robert-tolksdorf.de/vmlanguages.html`.

[21] Sun Microsystems Inc. The HotSpot^{TM} Performance Engine. `http://java.sun.com/products/hotspot`.

[22] Sun Microsystems Inc. Java Virtual Machine Profiler Interface (JVMPI). `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html`.

[23] Guido von Rossum. *Python Tutorial*. `http://www.python.org/doc/current/tut`.

[24] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, 2nd edition.* O'Reilly & Associates, Inc., 1996.

[25] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren and Clark Verbrugge. EVolve, an Open Extensible Software Visualization Framework. Sable Technical Report SABLE-TR-2002-12. McGill University, School of Computer Science, 2002.

[26] Benjamin Zorn. *Comparitive Performance Evaluation of Garbage Collection Algorithms*. Published as CSD-89-544 from University of California, Berkeley.