Understanding Memory Allocation of Scheme Programs

Manuel Serrano[1] *and Hans-J. Boehm*[2]

[1] `Manuel.Serrano@unice.fr`
`http://kaolin.unice.fr/~serrano`
Université de Nice Sophia-Antipolis
930, route des Colles, B.P. 145
F-06903 Sophia-Antipolis, CEDEX

[2] `hboehm@hpl.hp.com`
`http://www.hpl.hp.com/personal/Hans_Boehm`
Hewlett-Packard Company
1501 Page Mill Road, MS 1U-17
Palo Alto, CA 94304

ABSTRACT

Memory is the performance bottleneck of modern architectures. Keeping memory consumption as low as possible enables fast and unobtrusive applications. But it is not easy to estimate the memory use of programs implemented in functional languages, due to both the complex translations of some high level constructs, and the use of automatic memory managers.

To help understand memory allocation behavior of Scheme programs, we have designed two complementary tools. The first one reports on frequency of allocation, heap configurations and on memory reclamation. The second tracks down memory leaks. We have applied these tools to our Scheme compiler, the largest Scheme program we have been developing. This has allowed us to drastically reduce the amount of memory consumed during its bootstrap process, without requiring much development time.

Development tools will be neglected unless they are both conveniently accessible and easy to use. In order to avoid this pitfall, we have carefully designed the user interface of these two tools. Their integration into a real programming environment for Scheme is detailed in the paper.

1. INTRODUCTION

Since CPU speeds continue to increase faster than memory speeds, memory is and will increasingly be the factor that limits performance [8]. Excessive memory use has two drawbacks: The program itself makes less effective use of the higher layers in the memory hierarchy, and it may interfere with other processes running on the same machine. Functional languages have a reputation for memory consumption. In order to deliver fast applications implemented in these languages, we need tools that can be used to diagnose problems with unexpected memory use. We have designed two such tools that aid in reduction of memory consumption of Scheme programs.

1.1 Garbage collected languages

Garbage collectors (GCs henceforth) have very desirable properties: by automatically reclaiming useless memory cells, they make programs easier to write, safer, and easier to maintain.

Unfortunately, GCs often hide the complexities of memory management so well that programmers lose track of its cost. It is extremely difficult to determine when a GC will deallocate a data structure. It is a misunderstanding to think that because GCs automatically reclaim useless cells, they keep the memory occupation minimal. It has already been noticed that, in some situations, GCs enlarge the size of the programs working sets [19]. Consequently, very often, programs allocate and consume more memory than needed.

1.2 Scheme specificities

Precise evaluation of memory allocation size is more difficult in higher order programming languages than in conventional languages due to the distance between the high level constructs and the instructions executed on the hardware. Compilers have to generate sequences of operations for which the complexity is not always apparent at the source code level. It may happen that the compilers introduce run time heap allocations where none were obvious in the source.

Such "hidden" allocations are frequent in Scheme programs. For instance, let's take the following definition:

```
(define (show-value value)
   (print "value is: " value))
```

The function `print` accepts optional arguments. The Scheme semantics specify that such optional arguments must be placed in a freshly allocated list that is passed as the actual argument. That is, if `print` is implemented as:

```
(define (print . l)
   (for-each display l))
```

each time `show-value` is called three pairs are allocated. These allocations have no location in the source code. They are not apparent!

Scheme library functions may allocate substantial amounts of memory. One may write a function like:

```
(define (append-rev l1 l2)
   (append (reverse l1) (reverse l2)))
```

If `len1` is the number of pairs of `l1` and `len2` is the number of pairs of `l2` then `append-rev` allocates `2*len1+len2` pairs because both `append` and `reverse` allocate. Changing `append-rev` to:

```
(define (append-rev! l1 l2)
   (append! (reverse! l1) (reverse! l2)))
```

eliminates all allocations from `append-rev` because it reverses the two lists in place, and then appends them using one single pointer assignment. Obviously `append-rev!` and `appendrev` are not equivalent because `append-rev!` *changes* its arguments. But `append-rev` can sometimes be replaced with `append-rev!`.

When studying a program it may be difficult to detect that a function such as `append-rev` is responsible for many memory allocations. This is the role of an allocation profiler. It reports the frequency with which allocation sites are used.

1.3 Our Tools

We have implemented two distinct tools for analyzing memory allocation.

KPROF is an allocation profiler embedded in our regular Scheme time profiler. For each source code function, it reports on the number and kind of allocations. KPROF presents estimates of the exact allocation numbers, using a technique similar to gprof [6]. In addition, it provides information about the operation of the GC. All of this can be accomplished with low overhead, and no per object space overhead.

Allocation profiling reports on heap growth, but it cannot report on memory leaks. KBDB is a heap inspection tool. At first, it acts as a debugger. Programs are run interactively. They can be stopped, the variables, the stack and the heap can be inspected, and execution can be resumed. But in addition, the heap can be displayed, with each pixel representing one cell of the heap.

Individual cells can be inspected by simply pointing at their corresponding pixel in the image. When a cell is inspected, the Scheme type of its value, its allocation site, and its approximate age are displayed. In addition, KBDB displays *root chain links*. That is, KBDB can be used to understand why a specific cell is considered live by the garbage collector. This facility can be used to track down most kinds of memory leaks presented in Section 2.

The bitmap representation can be cheaply generated from the heap. However, unlike KPROF cell inspection requires additional per object memory overhead.

1.4 Contributions

We characterize the types of "memory leaks" we have encountered in garbage collected environments, and discuss in detail our experience with memory leaks in one particular Scheme program. We are not aware of other general discussions of the issue, especially in the context of strict languages.

We present a complete, easily usable, set of tools for examining memory allocation in garbage collected languages. We demonstrate how they can be used to identify and track "memory leaks".

Our KPROF allocation and time profiler is based on standard time profiling techniques. We explore and measure its utility as an allocation profiling tool.

Our KBDB tool allows exploration of heap reference patterns in a style similar to Jinsight [4]. However, it uses a different mechanism for displaying the results, and a different data gathering strategy. The latter simplifies use, allows easier scaling to large applications, and allows problems involving the garbage collector itself to be isolated.

We believe the techniques presented here apply to any language environment with garbage collection and run-time type information (*e.g.* Smalltalk, CLOS or Java).

1.5 Organization

Section 2 discusses memory leaks in garbage collected environments. Then, Section 3 presents the facilities provided by KPROF and how it fits into our integrated environment named BEE. It also discusses how allocations are estimated by KPROF. Section 4 presents KBDB and its integration in the BEE. Section 5 demonstrates how KPROF and KBDB can be used in the context of a real application and shows the run time overhead of instrumented programs. Section 6 compares KPROF and KBDB to existing tools. Section 7 presents some possible extensions to KPROF and KBDB.

2. MEMORY LEAKS OF GARBAGE COLLECTED LANGUAGES

In an environment using C-like explicit memory deallocation, the term "memory leak" usually refers to memory that is no longer accessible via any chain of pointer dereferences, but has not been deallocated. Since it is no longer accessible, it cannot possibly be deallocated in the future.

It is the job of a garbage collector to eliminate such leaks. Thus such leaks cannot occur in garbage collected environments. When we talk about a "memory leak" in a garbage collected language, we are referring to memory that still appears accessible to the garbage collector but is, in some other sense, no longer needed by the program. There are a number of reasons why this may occur:

- It may be referenced through an *algorithmically dead* variable or an easily identifiable slot in a data structure. This is the most common problem. Once identified, it can be usually be easily repaired by resetting the reference. Such leaks are usually bounded, but see [2] for a case in which an extra reference introduces an unbounded leak. In most cases, the reference is eventually overwritten, and the leak is thus temporary. But even temporary leaks can appreciably increase the heap size required by a process.

- It may be referenced through and *algorithmically dead* slot in a data structure, but the slots are interspersed in the data structure and expensive to identify. This is rare, especially in programs written for garbage collection. But we know of one case (a compiler) in which it prevented easy replacement of manual deallocation with garbage collection. This is similar to the preceding case, except that it may require much more substantial algorithmic changes to repair.

- It may be referenced through a pointer that is itself still live (*e.g.*, for use in a Scheme `eq?` comparison), but is never dereferenced. This can happen, for example, if the later stages of a compiler refer to identifiers exclusively with symbol table pointers, so that the actual strings representing the identifier could be discarded. This again appears to be rare.

- It may be genuinely referenced from a data structure that grows much larger than intended. The canonical example of this is a cache that was intended to remain bounded, but in fact grows without bound over time. Although the extra data may be accessed, overall performance of the program would increase if some of it were discarded.

- It may appear to be referenced to the collector, even though it is not truly accessible by following pointer chains from a live variable. This may happen because the collector has imperfect information about liveness of pointer variables, because the collector is conservative and has misidentified a non-pointer as a pointer[2], or because of an unfortunate promotion in a generational collector [10]. In most, though not all cases, such leaks are again temporary and bounded.

Usually, though not always, the hardest task in removing such a "memory leak" is to identify its source. We describe tools that can be used to do so. The tools rely on the garbage collector itself, and hence can be used to trace problems caused by idiosyncrasies of the garbage collection algorithm itself, or by interactions between the garbage collector and client, in addition to those caused purely by the client program.

3. KPROF: AN ALLOCATION PROFILER FOR SCHEME PROGRAMS

The first of our two profiling tools is named KPROF. It reports the number of times allocators are called from each function. In addition, KPROF reports on the evolution of the heap during the execution of the program. Each time a collection is triggered, the heap size, the number of live objects, and the number of allocations since the previous GC are recorded.

3.1 How to use KPROF

KPROF is one of the tools comprising the BEE [18], an integrated development environment for the Scheme programming language [7]. We think that it is of primary importance for development tools such as profilers and debuggers to be highly available and easy to use. Profiling with KPROF requires the usual profiling cycle: compile, record (*i.e.*, run the application with a specific input set), display. The BEE helps with all these tasks by handling compilation and execution. That is, to profile the allocation of one program execution, a user just has to click one dedicated icon. Once compilation and execution are completed, a new window displaying the main profile information is popped up. Figure 1 displays the time profile for the Queens program, a small Bigloo program that computes the number of solutions to the N-queens problem. The Queens program builds all possible configurations of the queens on a chess-board. Each configuration is implemented as a list. Thus the benchmark is allocation intensive.

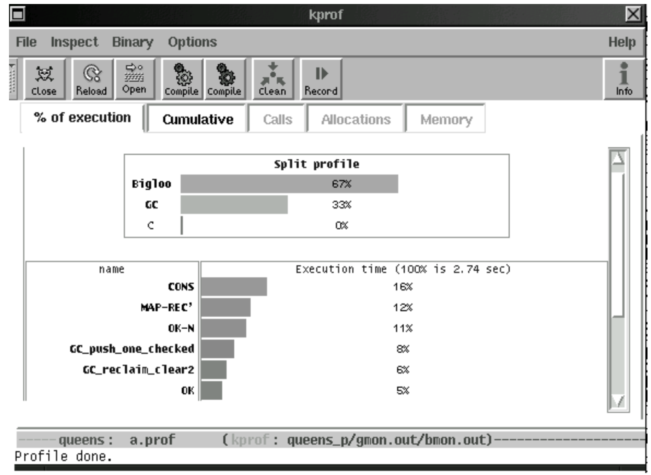


Figure 1: A plain profiling window

3.1.1 Plain profiling

KPROF distinguishes between the execution time spent in Scheme functions (labeled `Bigloo` as the name of our Scheme compiler), the garbage collector, and other C functions. User programs may mix C functions with Scheme functions. In such an environment we have found it best to display information by implementation language. The top lines of the profile window teach us that about 67% of the execution time is spent inside Bigloo functions and 33% inside the garbage collector. The bottom lines display the time spent in each function of the program. In particular, we read that 16% of the total execution time is spent in the Scheme `CONS` function.

KPROF may also display the overall amount of memory allocation during an execution (see Figure 2). We learn that 11.6MB have been allocated to run the Queens program and that 90% of the allocations are lists.

3.1.2 Dynamic call graphs

Since KPROF uses regular profiling facilities, it can browse the dynamic call graph. For instance, it can report which functions are allocating a large fraction of the `CONS`es. Examining the functions that call `CONS` would report that the Scheme function `MAP-REC` is responsible for more than 42% of the calls. KPROF may also compute estimates of the number of allocator calls by a function and its (direct and indirect) callees. We refer to these as *indirect allocations*. For

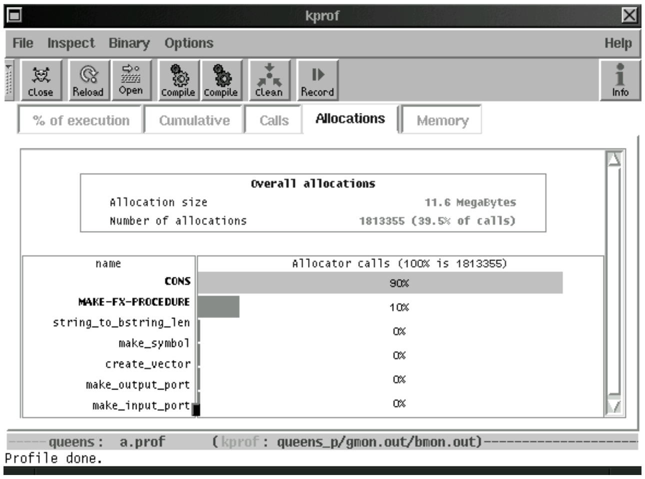


Figure 2: Queens's allocations

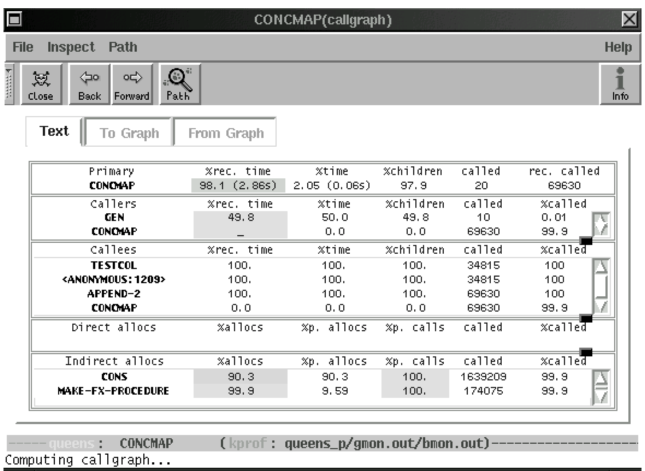


Figure 3: CONCMAP profiling

instance, consider the CONCMAP function of Queens:

```
(define (concmap f l)
   (if (null? l)
       '()
       (append (f (car l)) (concmap f (cdr l)))))
```

This function does not directly call any allocator, but obviously the functions it calls (its callees, notably APPEND) call CONS. Figure 3 presents the profile information computed by KPROF. The `Direct allocs` section is empty, as expected. But we discover that the CONCMAP callees are responsible for 99% of the calls to CONS (`Indirect allocs`, `%called` column). These calls account for 90.3% of the overall allocations invoked from CONCMAP (`Indirect allocs`, `%allocs` column).

KPROF may display the dynamic paths that exist from one function to another, for instance, the paths that go from CONCMAP to CONS as the one presented Figure 4. Each edge label stands for the percentage of the caller calls devoted to that callee. For instance, 20% of the calls operated by

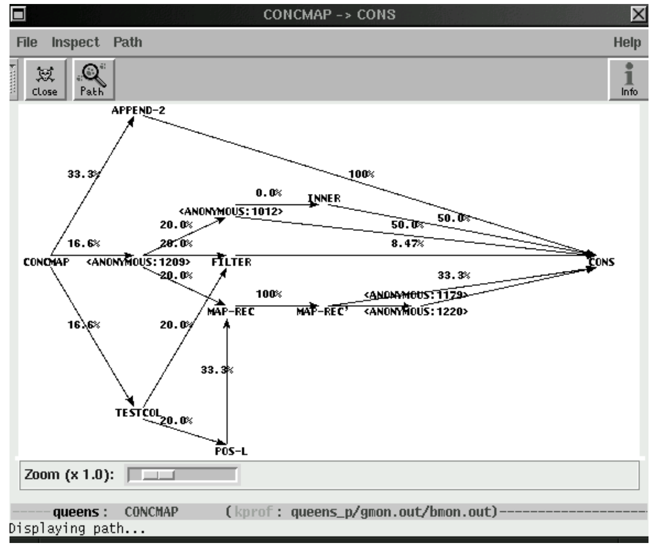


Figure 4: CONCMAP calls to CONS

TESTCOL are calls to FILTER and 8.47% of FILTER calls are calls to CONS.

3.1.3 Memory profiling

KPROF may then display the heap configurations recorded by each garbage collection during program execution, as in Figure 5. Here, execution time is measured in GCs. We learn that eighteen GCs were required to execute Queens. For each GC, we learn the heap size (that varies here from 1.05MB to 1.20MB). KPROF also displays the stack size (the maximum is reached on GC 9 with about 0.25MB). The other low curve represents the number of live objects after each collection. That one goes from about 0.13MB to about 0.30MB with a maximum reached at GC 6 with about 0.30MB of live objects. The fourth curve takes into account the number of objects that have been allocated since the previous GC (about 1.0MB of allocations to GC 12).

During the study of such a profile it may be convenient to focus on some specific parts of the program. For instance, our Queens example includes two different implementations of the same strategy. It could be interesting to compare these different implementations at a glance. This is permitted by our Scheme extension: the `profile` form. Its syntax is:

```
(profile <label> <s-expr>)
```

Its evaluation is equivalent to `(begin` *s-expr*`)` but it will force KPROF to report data for a pseudo-function named *lbl*, with all costs associated with the evaluation of *s-expr* reported as part of the execution of *lbl*. It will thus be possible to determine the number of allocations executed during the evaluation of *s-expr*. In addition, the evaluation period for *lbl* is displayed on the KPROF's memory profile. Figure 5 includes the result of several `profile` forms, labelled SOL1, SOL2, and POS-L, indicated immediately above the memory usage graph.

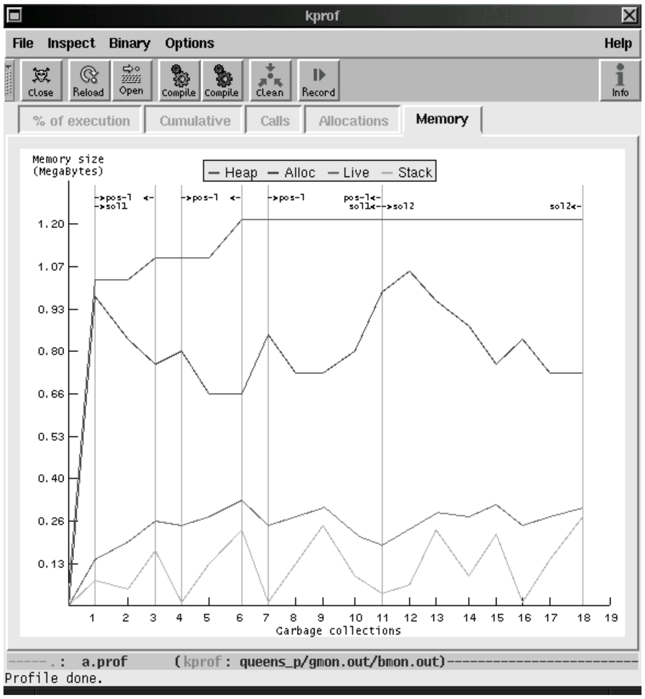


Figure 5: Queens's memory configurations

`Profile` forms are close to the `ghc` Haskell implementation's "set cost center" construction (`scc` in short) [16, 15, 17]. Cost centers are more central to Haskell because they form the basis of profiling for this lazy programming language. To KPROF, `profile` is only a convenience that enhances the presentation of the generated profiles.

3.2 KPROF implementation

Bigloo, our Scheme compiler, translates from Scheme to C. The generated C code conforms to the standard C coding style [3]. In short, Scheme functions are compiled into C functions and Scheme variables are compiled into C variables. Because a direct correspondence exists between the produced C code and the initial Scheme source file, it is possible to reuse standard C tools to profile Scheme source programs. KPROF is designed as a layer surrounding the standard Unix GPROF tool [6, 5]. This technique is described in a forthcoming paper [18]. KPROF decodes GPROF information (in particular, KPROF demangles GPROF symbols) and uses the result to compute allocation profiles.

3.2.1 Inherited features

Since KPROF is a front-end to GPROF it inherits some of its facilities and, alas, some of its inaccuracies. These are described in some details in the GNU-gprof documentation [5]. We shortly summarize them in this section.

- Run-time figures are based on a sampling process. To determine the time spent in each function GPROF samples the hardware program counter at regular intervals (*e.g.*, every 0.01 seconds). The entire time interval is charged to the corresponding function. To produce reliable execution time estimates, the overall execution time of the application must be much larger than the sampling period.

- The number of calls are exact. They are computed by inserting an additional call to an accounting function into the prelude of every profiled function. This function is responsible for recording in an in-memory call graph table both its parent routine and its parent's parent.

- Callees run times presented in the call graph reports are probabilistic. The record made of an execution does not contain any information relative to the dynamic call graph. Here is an excerpt of the GPROF documentation describing the technique:

  "The assumption made is that the average time spent in each call to any function `foo` *is not correlated with who called* `foo`*. If* `foo` *used 5 seconds in all, and 2/5 of the calls to* `foo` *came from* `a`*, then* `foo` *contributes 2 seconds to* `a`*s callees time, by assumption.*

  This assumption is usually true enough, but for some programs it is far from true. Suppose that `foo` *returns very quickly when its argument is zero; suppose that* `a` *always passes zero as an argument, while other callers of* `foo` *pass other arguments. In this program, all the time spent in* `foo` *is in the calls from callers other than* `a`*. But* GPROF *has no way of knowing this; it will blindly and incorrectly charge 2 seconds of time in* `foo` *to the children of* `a`*."*

3.2.2 Allocation estimates

KPROF reports on the number of *direct* and *indirect* calls to allocators. The *indirect* calls are the calls made by the functions themselves and their callees. The *direct* calls are exact values but the *indirect* ones are estimated.

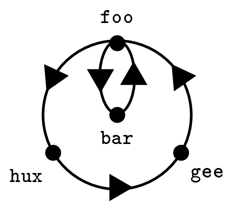
The assumption made to compute *indirect* allocations is similar to the one used to compute callees run times. The indirect allocations of a function $\mathcal{F}$ are the direct allocations of that function plus a percentage of the allocations performed by its callees. For each callee $\mathcal{C}$, this percentage is calculated by dividing the time spent in $\mathcal{C}$ when called by $\mathcal{F}$ by the overall time spent in $\mathcal{C}$. For instance, let us suppose a function `F1` calls an allocator `A1` n times and also calls a function `F2`. `F2` calls the allocator `A1` m times and it calls no other functions. Now let us suppose that 30% of the run time of `F2` is spent when it is invoked by `F1`. KPROF reports that the number of recursive calls to `A1` from `F1` is $n + 0.3 \times m$.

The algorithm has to distinguish between functions that belong to recursive cycles and functions that don't. In the former case, functions are one of their own callees. Here is the algorithm that computes the allocation estimates for a function `f`:

```
alloc( f ) =
   if it exists a cycle going through f
     then cycle-alloc( f )
     else no-cycle-alloc( f )
```

In the presence of recursion, the call graph contains cycles, starting with a given function, proceeding through its (possibly indirect) callees, and leading back to the original function. A function cannot belong to more than one cycle in a

GPROF report. If it could, then a function would be part of two different cycles which can be drawn as follows:



However GPROF has no way to find out that there is no path that goes from `bar` to `hux` or `gee` because `foo` is a callee of `bar` and `gee` a callee of `foo`. Thus Gprof assumes that all such functions can be indirectly called by each other. Thus KPROF's `cycle-alloc` function handles strongly connect components of the call graph as single nodes. This is the same strategy as the one described in the Gprof presentation [6].

If the function `f` does not belong to a cycle, then, the formula illustrated with `F1` and `F2` is applied to all of `f`'s callees:

```
no-cycle-alloc( f ) =
  let C = the cycle going through f
          (a possibly empty set of functions)
      E = f's callees
      R = f's direct allocations
    foreach function g in E - C do
      if g is not an allocator
        then let A = alloc( g )
                 δ = the percentage of time spent
                     executing g when called
from f
               foreach call k in A do
                 add δ * k calls to R
    return R
```

Computing indirect allocations for a function `f` belonging to a cycle $\mathcal{C}$ requires two steps. In the first one, the indirect allocations are computed excluding all the functions of $\mathcal{C}$ by the means of `no-cycle-alloc`. Then, in a second step, all indirect allocations of the functions of $\mathcal{C}$ are added to the indirect allocations of `f`. In the graph example the set of indirect allocations of the function `foo` is the union of the direct allocations of `foo`, `hux`, `bar` and `gee`.

The algorithm is partially validated by the fact that KPROF correctly reports that the entry point of a Scheme program indirectly calls 100% of the allocators (a whisker away because of floating point errors). However, as we will see in Section 5.2 these estimates may be imprecise. Nevertheless, since they are fast to compute, they can be used as a first indication that thorough investigation is needed. In many cases even these estimates may unveil obviously wrong behaviors. We will show in Section 5.1.1 that the estimates have permitted drastic reductions in the memory consumption of the Bigloo compiler itself.

3.3 KPROF and code generation

To "record" an execution, the source code has to be compiled in "profile" mode. That is, the compiler has to introduce some extra instructions for producing profile output file used by KPROF. In this section we describe that code.

3.3.1 *Profiling and optimizations*

Optimizations that change the initial structure of the source code have to be disabled in order to make profile information accurate. Inlining is one of these optimizations, since it replaces function calls with the bodies of the called functions. Since functions are the smallest entities the profiler reports on, it is important that user functions are not inlined when in profile mode. However it has been demonstrated that inlining is an important optimization, especially for functional languages [9]. We are thus facing a dilemma: should the profile compilation mode enable aggressive optimizations such as inlining even if this reduces the accuracy of the profiler reports, or should it disable optimizations? One should notice that the second solution also effectively reduces the accuracy of the profiler, since the measured program is likely to behave differently from the final version.

We don't think there is a "best" solution for this problem, and instead provide the user with two different profiling modes. One enables aggressive optimizations, the second disables all of them. Figure 1 presents the profile with optimizations enabled. Note that Bigloo never inlines `CONS` since the allocation sequence is too long.

`Ghc`'s cost center construct and `profile` forms allow fine grain tuning. The more they are introduced, the more optimization is inhibited. These two forms require source code changes, but they allow exact control of profiling.

3.3.2 *Profiling and local functions*

As in most functional languages, Scheme has local, possibly anonymous, functions. KPROF reports on these as it does on global functions. Because several local functions may have the same name and reside inside the same module, the profiler prefixes their name with the name of the enclosing function.

For anonymous functions, the compiler generates a name from the source file location. For instance, let us suppose an excerpt of a file `F.scm`:

```
150: (define (foo x)
151:    ... (map (lambda (y) (+ x y)) ...) ...)
```

The function of line *151* will be named `F.scm:151:7347`, where `7347` is a stamp that avoids name collision. When the user invokes the editor on that function, the profiler invokes the editor on the correct file.

3.4 KPROF limitations

Allocation profiles are not sufficient to track down memory leaks. For instance, in Figure 1 the "live objects" curve seems to reveal an increase of live objects in the `sol1` stage. When `sol1` completes there are still some live objects and the difference between the number of live objects at the beginning and at the end of `sol1` is positive. KPROF provides no information as to whether this increase is normal or if it is due to a memory leak. It reports on *allocations* whereas memory leaks concern *deallocations*. To address this problem, we have designed and implemented a second tool named KBDB.

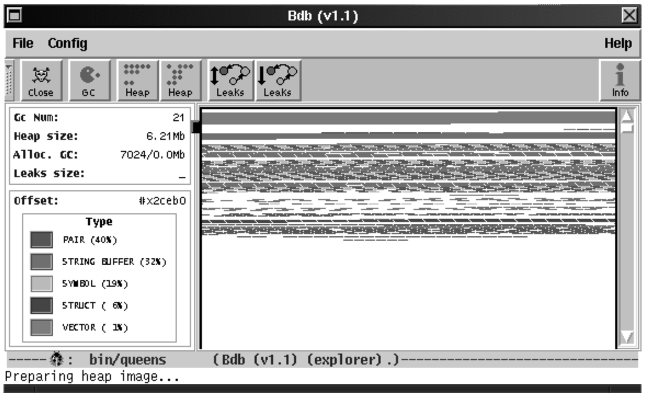


Figure 6: An heap view according to object types

4. KBDB: A HEAP INSPECTOR FOR SCHEME PROGRAMS

KBDB is an interactive heap inspector. It displays the live objects and the chains of pointers that link these objects in the heap. KBDB is embedded in our regular Scheme debugger. Each time an execution is suspended (for instance, when a breakpoint is reached) the heap may be inspected.

When KPROF reveals a *suspect* increase of live objects KBDB can be used to discover if this is due to a memory leak. Obviously this requires thorough knowledge of the source code. To suspect that a computation leaks memory, it must be known that this computation *should not* increase the number of live objects. KBDB can only answer the question "what is the amount of memory still in use after evaluating that particular expression of the source code?".

4.1 Plain heap inspection

To start with, KBDB acts as a regular Scheme debugger. It enables suspension and resumption of execution. When an execution is suspended the variables and the stack of the computation may be inspected. In addition to these traditional features, KBDB can also display a snapshot of the heap as a 2 dimensional picture in which each pixel is associated with a memory location. Unused memory locations are left blank. Objects are distinguished by their color. Currently two color schemes have been implemented. In the first, objects are colored according to their type. For instance, all strings are displayed in blue, the pairs in red, and so on. The second classification uses the age of the objects to determine their color. Figure 6 is a snapshot of KBDB displaying a heap during the execution of the **queens** program, with objects classified by types.

Objects are represented by horizontal stripes ended with a white pixel. The larger an object is, the longer is its associated stripe. Sections of the heap can be magnified to make selection of specific objects accurate. Detailed information about a clicked object is then reported. For instance, clicking on a `PAIR` stripe could display:

```
Holder     : FILTER, frame 4 (local L) ;; The value
holder
type       : PAIR (0)                  ;; The object
type
Producer   : CONCMAP                   ;;        The
producer
Generation: 1                          ;;  The bird
date (in GC)
Size       : 12                        ;;  The byte
size
   [PAIR]    (1)                       ;; The hold-
er chain links
   [PAIR]    (2)
(bdb:MAIN) display (0)
$63 = (8)
(bdb:MAIN) display (2)
$63 = (6 7 8)
```

The `PAIR` we have clicked on has been allocated in the function `FILTER` before the sixth collection and its size is 12 bytes. The producer is the "first" user function that calls the allocator. That is, library defined functions are not reported as producers. For instance, the producer of the pairs allocated by `APPEND` in the `CONCMAP` function of Section 3.1.2 will be reported as allocated by `CONCMAP`, not `APPEND`. The `PAIR` was held by the local variable `L` of the function `FILTER`. An "holder" is either a global variable, a local variable, or simply a stack frame (when the value is not stored in any local variable but simply passed to another function). In the remainder of the paper, we will indistinguishly name an holder, a GC root. The stack frame of the `FILTER` invocation that holds `L` is the fourth one in the stack (`frame 4`). The *holder chain links* represent the pointers chain from the holder (*i.e.*, `FILTER`'s `L` local variable) to the inspected object (here a `PAIR`). The labels of the holder chain links can be used to explore or display the objects of that chain.

4.2 Memory leaks

Provided with a heap inspector, memory leak detection is easy. The framework for finding leaks in a suspect expression $\mathcal{E}$ is the following:

1. Stop the execution before the evaluation of $\mathcal{E}$.
2. The number of already completed collections is GC_0.
3. Trigger a garbage collection.
4. Resume the execution.
5. Stop the execution when evaluation of $\mathcal{E}$ is completed.
6. Trigger a new garbage collection.

The current collection number is now GC_n. Leaking objects are those that have been allocated during the evaluation of $\mathcal{E}$ that are still live, *i.e.* live objects that have been allocated after GC_0 and before GC_n.

To find a memory leak using the current KBDB interface, two breakpoints have to be set. One before the suspected expression and one after. When the execution reaches the first breakpoint, a simple click on the `leaks` icon triggers the previously described steps 2 to 6. A picture consisting only of the live objects is then displayed as in Figure 7. In this snapshot of the heap only the "leaking" objects, *i.e.* newly allocated and still live objects, are displayed. The roots causing the leaks are displayed in a different color than the leaking objects. As reported on the left side of figure 7, the entire leak size is of 8480 bytes in this case. Leaking objects may be selected as before. Clicking on one of these objects could produce:

```
Holder    : COUNT
Type      : PAIR   (0)
Producer  : NSOLN
Age       : 2
Size      : 12
   [PAIR]      (1)
   [CELL]      (2)
   [PROCEDURE] (3)
```

This object can be displayed:

```
(bdb:MAIN) display (0)
(2 3 4 5 6 7 8)
```

Even if this is not obvious in the gray scale display, there is only *one* GC root that is culprit for the entire leak. That root is the end of the root chain of the object we have selected. It can also be selected:

```
(bdb:MAIN) explore (5)
Holder    : COUNT
Type      : PROCEDURE   (0)
Producer  : COUNT
Age       : 2
Size      : 20
```

We can now draw some conclusions from this inspection of the **Queens** program:

- As we suspected, the evaluation of the `SOL1` form leaks memory.
- That leak is composed of small lists (the entire leak size is 8.3KB).
- All lists are accessible from the same root.
- The root is the anonymous closure (a `PROCEDURE` type) that has been allocated in the `COUNT` function.

Actually, the `COUNT` function is a memo function. It allocates lists and stores them in a table that is never reset. In Section 3 we suspected a memory leak. KBDB has demonstrated that this leak really exists.

4.3 KBDB implementation

We modified the Boehm-Demers-Weiser garbage collector to provide back-pointer information, as part of the debug information that could already be associated with individual objects. Each allocated object is provided with additional slots to store the source code location of the allocation and one *back pointer* slot that is filled by the collector during the marking process.

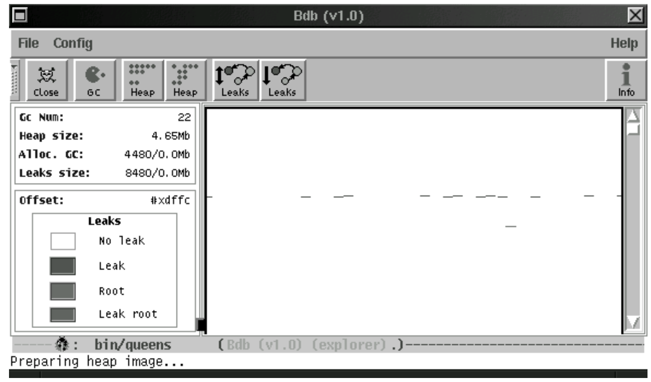


Figure 7: Memory leaks unveiled

The contents of the back pointer slot point to the location of the pointer that caused the object in question to be marked. If the object is reachable by more than one path, the one that happened to be followed by the collector will be reflected in the KBDB output. In the, usually infrequent, cases in which the conservative collector follows a stale stack pointer, or a misidentified pointer, that fact will also be accurately reflected in the chain of back pointers. Thus even such problems become debuggable.

As discussed in the related work section, there are other ways to display backward reference chains. This technique is the second one we have implemented, and by far the simplest. It was suggested by Alan Demers.

Details of the collector backtracing interface can be found in the collector distribution. In this section, we focus on the implementation of the heap picture construction and the KBDB architecture.

4.3.1 *The debugged applications*

Debuggable applications embed special library functions that are in charge of constructing the heap display. When KBDB is to display a heap, it requests that the debugged application produce a file on disk containing the picture. The picture file is constructed without additional memory consumption by means of simple linear scans of the heap. The garbage collector is able to report, for each address of the heap, if it is part of a live object, and to retrieve the size of that object. The algorithm to build a heap picture is:

```
make-picture( ℱ ) =
  let min-addr = The heap min address
      max-addr = The heap max address
      i = 0
      pic = create-picture()
   while i + min-addr < max-addr do
     if i + min-addr is the address of a live object?
     then let size = GC-object-size( i + min-addr )
              stop = size + i
              type = SCM-type( i + min-addr )
          while i < stop do
            set-pixel-color( pic, i, ℱ( type ) )
            i = i + 1
     else set-pixel-color( pic, i, "white" )
          i = i + 1
   return pic
```

The argument $\mathcal{F}$ is a parameter of the picture construction. It is a function that maps Scheme objects to colors. It enables various coloring schemes to be applied to the heap construction (such as the type based or the age based ones).

The key point of this algorithm is that a picture file is directly dumped during a heap traversal. There is no need to allocate a memory area of the size of the inspected heap because of the direct mapping from heap addresses to picture pixels. This is possible only if the garbage collector is able to report information about random addresses in the heap. It is not clear how exact collectors could implement this.

File generation is straightforward. Leak detection and age-based coloring do not require pre-computation. On the other hand, association between types and color requires an additional first linear traversal of the heap in order to allocate colors to the most frequently used types (only those are allocated specific colors, infrequent types are all displayed with one unique color). Then, during a second linear traversal, the picture is built according to the `make-picture` algorithm.

4.3.2 Kbdb *display generation*

When a debugged application has generated a picture file, Kbdb reads that file. Note that a picture file is often much smaller than the inspected heap because one pixel represents a memory word (*i.e.*, 4 or 8 bytes). For a monochrome picture 8 pixels, can be stored in a single byte, making a monochrome picture about 32 times smaller than the inspected heap. A four color picture is 16 times smaller than the heap. Four colors are sufficient for memory leak displays.

We have concentrated on the compactness of the heap representation. We think this is a central issue. If the representation requires too much memory, the system cannot be used to inspect large heaps. We have successfully applied Kbdb to our Scheme compiler, demonstrating that it can be used on heaps larger than 8MB.

5. APPLYING Kprof AND Kbdb

This section presents the results of our first attempt to apply Kprof and Kbdb to a real application: our Scheme compiler. The aim of this section is to show that Kprof and Kbdb are useful in practice. For this experiment we have looked at bootstrapping the compiler, that is, compiling a part of the compiler with an instrumented version of the compiler. We have arbitrarily time bounded that effort by allocating only two days to profiling. This section reports on what we have learned about the compiler during these two days and concludes with some measurements that show the execution overhead of profiling and debugging.

5.1 The Bigloo compiler

Bigloo compiles Scheme into C. It is written in Scheme and compiled by itself. Bigloo consists of 40,000 lines of code. It reads the program to be compiled and builds an abstract syntax tree (henceforth Ast) to represent the program. This tree contains a structure of 23 different node types. There are nodes for constants, variable assignments, conditionals, function calls, etc. The compiler is made up of stages, each of which can be seen as a process that modifies the Ast. The driver is a Scheme function that looks like:

```
(define (compiler src)
   (let ((ast (build-ast src)))
      (macro-expand! ast)          ;; 1st stage
      (function-inline! ast)       ;; 2nd stage
      ...
      (code-generate! ast)))       ;; 20th stage
```

This rigid structure, in which each stage acts as a stand alone program, helps the implementation and maintenance of the compiler. It also helps with profiling the compiler. Because of that structure it is easy to let Kprof reports on allocations stage by stage. It only requires instrumenting the compiler driver with profile forms such as

```
(profile ast (build-ast src)).
```

5.1.1 Reducing compiler memory allocation

Figure 8 presents the heap profile as reported by Kprof when bootstrapping a module of the compiler. The file chosen for that experiment is the one that compiles into the largest C file. That file contains most of the classes representing Bigloo's Ast. In Bigloo's object system, class instance slots are fetched and changed via getter and setter functions. These functions are automatically generated by the compiler. The module we are studying produces a large C file because it contains most of the getters and setters of the Ast.

The overall allocation for compiling that module is 17.3MB amongst which, 56% are `CONS`es. This was a surprise to us. The Ast of the compiler is represented by a class hierarchy and we thought we have successfully avoided `CONS`ing in the code of the compiler. Actually, variable size data structures are implemented using `CONS`es. For instance, the list of formal parameters of a function is represented by a Scheme list, and sequences of expressions are also stored in lists (that is the `sequence` node of the Ast holds several values, one of which is a list of expressions). Kprof shows that these lists are much more numerous than we suspected. It is possible to reduce allocation for such lists. It would for instance, be possible to use vectors instead. This would reduce memory requirements since vectors are more compact than lists (at least when they contain more than two elements).

Kprof points out several other surprising allocations. We focus here on the most significant ones. The last compiler stage (namely `CGEN`) writes the C code on a disk file. This stage only *dumps* the Ast. It is not supposed to allocate memory. However, Kprof demonstrates that `CGEN` actually *does* allocate! Inspecting `CGEN` allocations using the allocator profiler shows that `CGEN` is responsible for more than 13% of the calls to `CONS`. Studying the dynamic paths that go from `CGEN` to `CONS` shows that nearly all the `CONS`es allocated during the evaluation of `CGEN` are called by functions that write on the disk. That is, the `CONS`es are allocated when the compiler uses the standard Scheme output functions (`DISPLAY`, `NEWLINE`, ...). These `CONS`es are allocated because, as reported in Section 1.2, variable arity functions allocate, and standard Scheme output functions accept an

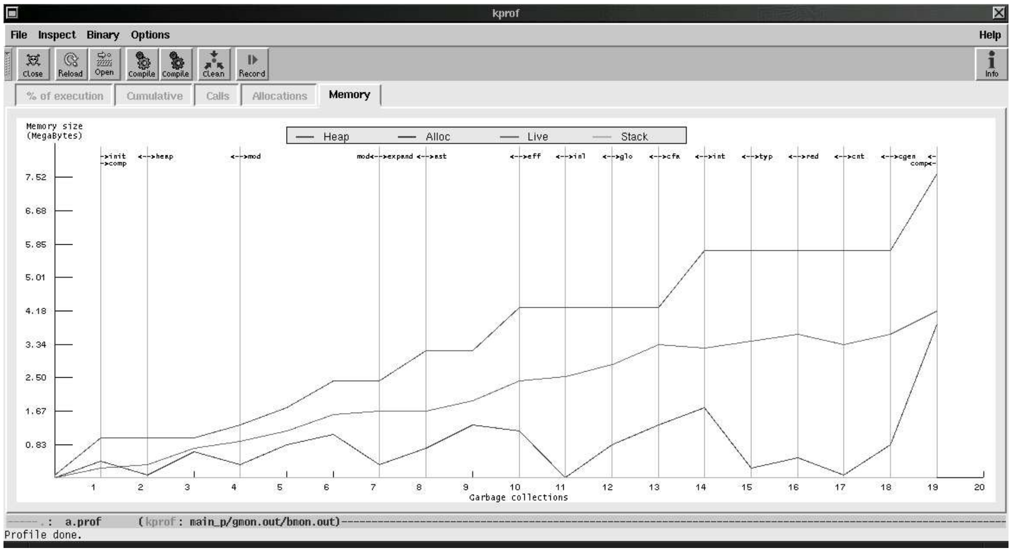


Figure 8: Profiling Bigloo

optional output port. In order to remove these allocations, we have implemented a simple source-to-source transformation. When a call to a regular Scheme output function is detected, it is replaced with a call to a specialized function accepting one or two arguments, depending on the nature of the call. We don't claim that this optimization is "general purpose", we only claim that with a one-hour effort, we have been able to reduce the memory allocation to 14.3MB, which is a reduction of 21%. The proportion of `CONS`es for compilation of the compiler dropped to 47%.

5.1.2 Chasing bootstrap memory leaks

Some stages of the compiler are expected to increase the live memory (such as the `AST` stage that constructs the Ast, or `INL` that implements inlining optimization). On the other hand, some stages implement optimizations or analyses that should reduce the size of the Ast. Amongst these stages is `EFF` (GC 10 to GC 11) which is a pass that computes the side effect property for each function of the Ast. The results of `EFF` are later used to implement regular optimizations such as data optimizations (`RED` stage). Surprisingly Kprof reports that `EFF` slightly increases the number of live objects (2.45MB to 2.55MB). We used Kbdb to inspect memory leaks of the `EFF` stage. Kbdb reported that `EFF` is responsible for 210KB of leaks due to only two GCroots. One is in a function called `MAKE-SIDE-EFFECT!` and the other in a function called `FUN-CALL-GRAPH!`. Inspecting the source code of these functions has revealed the nature of the two leaks: both functions use a table that is not reset when `EFF` is completed.

5.2 Impact of profiling and debugging

The performance difference of programs compiled in profile mode, debug mode or optimization mode should be as small as possible. If the difference is very noticeable, the profiler or debugger would be tedious to use. Even more seriously, if the performance degradation is too substantial, large programs can simply not be profiled or debugged. In this section we present some time and allocation measurements for the different versions. We have used three different programs: a small one (**Queens**, a 100 lines long program we previously discussed), a mid-size one (**Eval** the 500 line long Bigloo Scheme interpreter), and a large one (**Bootstrap**, the 46,000 line long Scheme compiler itself). Figure 9 compares the compilation time, execution time and the heap size of these three programs using different compilation flags, all other things being equal. The compilation time (*), run time (*) and allocation size ($^\sharp$) figures for the compiler **Bootstrap** have been gathered when compiling only *one* module of the source code of the compiler.

Comp is the compilation time, Size the size of the binary file. In order to present the impact of profiled compilation on the executable size, we have decided not to strip (that is not to remove the symbol tables from) the binary files in optimization mode. However, one should be aware that on our working architecture stripping an executable shrinks it by about a factor of three. In addition, all binaries are linked against static Bigloo libraries. (There is no issue here because all modes support shared libraries.) Run is the execution time (the minimum of user plus system time for three consecutive runs). Alloc is the amount of memory allocated during the execution.

The differences in size of the executable are important. Profiled executables are about four times larger. Debuggable executables are up to 10 times bigger (for **Bootstrap**). There is no way to avoid this increase because it does not depend on the Bigloo C generated code but on the assembly code generated by the C compiler. For instance, `app.scm` is one of the compiler source files. The generated C file `app.c` is 56KB long. When compiled with C optimization enabled, the object file `app.o` is 10KB in size When compiled in C debug mode, it enlarges to 51KB!

Debugged and profiled programs run slower than optimized programs. Independent of instrumentation, disabling optimization slows down programs by a ratio of 1.5 to 2. For

	Queens				Eval				Bootstrap			
Comp. modes	Comp.	Size	Run	Alloc	Comp.	Size	Run	Alloc	Comp.	Size	Run	Alloc
Optimized	2.3s	185k	1.3s	15.5MB	3.6s	167k	1.8s	7MB	6.6s*	898k	5.3s*	9.3MB$^\sharp$
Profiled	2.2s	494k	5.5s	15.5MB	3.41s	500k	6.11s	7MB	4.1s*	4643k	8.2s*	9.3MB$^\sharp$
Debugged	3.4s	620k	15s	69.2MB	7.4s	209k	9.0s	43.9MB	5.8s*	15929k	17.4s*	49MB$^\sharp$

Figure 9: Impact of the profiling and debugging compilations. Hardware configuration: K6/200, Linux 2.0.x, 64MB.

small programs, the difference in performance for instrumented programs is important (a factor of 10 for **Queens**). Experience seems to show that the larger the programs are, the smaller are the gaps between optimized and instrumented applications: A factor 5 for **Eval** and a factor 3.2 for **Bootstrap**.

As we have previously stated, KPROF does not increase memory consumption. In contrast KBDB does! The allocation growth factors are: 4.5 times for **Queens**, 6.3 for **Eval** and 5.2 for **Bootstrap**. That is, the memory overhead introduced by KBDB cannot be neglected. For our current implementation of the current run time system, the overhead for each allocated cell is of 6 words: 1 for a back pointer, 1 for producer information, 1 for age, and three other words used internally by the collector, primarily to support C debugging. These 6 additional words explain why the allocation size grows so much. Without debugging information, a Bigloo `CONS` is two words. That is, the run time type information is stored in the pointer to the pair (*i.e.*, there is no header word for pairs). This technique is no longer available in debug mode because all objects must have the very same data layout. As a consequence, in debug mode, a pair is 9 words large. Because of hardware alignment constraints this turned to 10 words, that is, 5 times larger. As reported by KPROF `CONS` allocations are dominant, it is thus not surprising that the overall heap usage increases by about a factor of 5, though we expect that to be reduced somewhat in the future.

5.3 Profiling validation

KPROF is implemented on top of Gprof. That is, KPROF re-targets the Gprof C sampling technique for Scheme. It is usually accepted that the accuracy of the C approximations delivered by Gprof is sufficient. However, it is conceivable that, due to differences in programming style, the Gprof technique applied to Scheme delivers inaccurate results. For instance, if Scheme programs make use of numerous smaller functions, higher smapling rates could be needed than for C. In order to show that it applies equally well to C and Scheme we have conducted another experiment. We have measured the number of calls per second for optimized Scheme and C programs. The number of calls have been gathered using exact Gprof function call counting. We use the three Scheme programs from Section 5.2. It is extremely difficult to compare Scheme and C programs, especially because it is nearly impossible to establish a set of representative benchmark programs. As much as possible we have tried to use C programs that perform the same kind of computation as our Scheme programs. The first one, **Amd** is a test released by AMD, which uses it to estimate the speed of its processors. The second, **Li** is a Lisp interpreter implemented in C, which is part of the Spec 95 benchmark suit. The last one, **Gcc** is the special version of the GNU-C compiler that is also included in the Spec 95 suite.

Prgm	Function call per second
Amd (c)	2,787,920 cs^{-1}
Queens (scm)	3,524,345 cs^{-1}
Spec95 Li (c)	5,280,070 cs^{-1}
Eval (scm)	1,581,905 cs^{-1}
Spec95 Gcc (c)	1,445,991 cs^{-1}
Bootstrap (scm)	1,453,832 cs^{-1}

These time figures show that function call frequency is similar for Scheme and C. In particular, the frequency is astonishingly close for **Bootstrap** and **Gcc**, both of which are compilers. Consequently, the execution time spent in each Scheme function is proportionally close to the time spent in each C function. Thus, there is no a priori reason to believe that Scheme requires different sampling techniques than C.

The second step of our validation was measurement of the accuracy of KPROF's allocation profiler. For this, we built a special version of the Bigloo runtime system that reports exactly on heap allocation performed by each function. In that version, each time an allocator a is called, every active function on the execution stack is marked as calling a. When execution completes, all this information is dumped into a file. This experimental runtime system is unrealistic because it is far too slow. With this version, execution of the **Bootstrap** benchmark requires about 8 hours on our hardware configuration. However, this slow implementation is still sufficient for estimating the accuracy of the indirect allocations reported by KPROF.

For the small and medium sized programs we have tested (including **Queens** and **Eval**), KPROF allocation profiling is very precise. We observed that function allocation estimates computed by the algorithm presented in Section 3.2.2 have an error rate of less than 5 %.

For larger programs, the accuracy of the estimates varies. If we inspect a function f that indirectly calls an allocator a, the quality of the estimate is highly dependent on the *length* of the path from f to a, that is, how many functions calls are needed to reach a from f. For instance, in Figure 4 the longest path from `CONCMAP` to `CONS` is 6, the shortest is 2. If shortest paths from f to a count a lot, that is shortest paths are more frequently traversed (in Figure 4 the shortest path is important because the vertex from `CONCMAP` to `APPEND-2` represents 33.3 % of all the calls made by `CONCMAP`) then the estimate for f is reliable. Otherwise, it is imprecise. In the compiler bootstrap benchmark the paths are relatively small, KPROF reports that the control flow analysis (`cfa` stage on Figure 8) allocates 1% of all closures and 1.2 % of the `CONS` cells, while actually it is responsible for 1.25 % of procedures and 2 % of the `CONS` cells. On the other hand, when paths are longer the estimates are imprecise. For the AST construction (the `ast` label on Figure 8), KPROF estimates the number of `CONS` cells to be 4.5 % and the number of procedures to be 7.1 %, while exact measures report 16 % of `CONS` cells and 12 % of procedures. The errors in KPROF's estimates are inherent to the lack of exact information about the dynamic paths. Using Gprof results, we don't think it is possible to produce more reliable results. However, in addition to the current estimates, KPROF could report on their accuracy. This new information could be computed from the number of paths and their length from a function to an allocator and it could be presented on the same window as the estimates, or it could be based on a call stack sampling technique.

6. RELATED WORK

KPROF is implemented on top of GPROF [6, 5]. Thus, KPROF inherits GPROF's run-time instrumentation and its computation of the dynamic call graph. In the past, alternative techniques to gather profile informations have been proposed [1]; however these techniques seem less accurate than GPROF's ones.

`Mprof` [20] is an allocation profiler. It relies on techniques which are similar to those of KPROF but with a different implementation. Instead of re-using GPROF results, `mprof` implements its own monitoring and textual displayer. Instrumented applications record all the call chains that lead to allocation sites. In order to avoid overly large record files, `mprof` compacts its records with a technique slightly more accurate than the GPROF's one.

The Haskell community has been fairly active at exploring heap profiling for lazy functional languages. Some of this work concentrates on studying the graphical means to display profiling information [14]. Other research concentrates on providing a semantics to the evaluation of lazy languages with profiling [17]. Yet other work focuses on issues close to memory leak detection [13, 12].

The graphical display advocated in an early paper by Runciman and Wakeling [14] is used in all other Haskell studies. This representation is totally different from the one we use. It displays, over the whole execution of a program, the heap composition. That is, the heap size and the percentage of pairs, strings, vectors, etc. It could be that such a nice representation enables faster understanding of the memory allocations of a program, in particular when it is used to display a *producer* view. A producer is a function that calls some allocators. This view definitively enables understanding of who's responsible for the allocations of a program and, more importantly, how long these objects live. We think this representation could point out memory leaks. However, we don't think it helps much with fixing these leaks. To fix a leak, one has to understand why objects fail to be reclaimed, which is not reported by heap profiling.

A more recent paper Röjemo and Runciman [11] presents a study that could help in understanding memory leaks. They present the so-called *lag*, *drag*, *void* and *use* phases. They characterize wasted space as that occupied by objects that have passed their last use, or have been allocated but are not yet in use. They present tools to analyze the presence of such objects. Execution has to be completed before any profiling information can be reported. This approach is orthogonal to the one presented here and could be usefully combined with it.

KBDB has been inspired by the work of De Pauw and Sevitski [4], in which the authors present Jinsight, a tool for visualizing reference patterns and tracing memory leaks in Java programs. Jinsight operates on an instrumented JVM that keeps track of *back pointers*, linking all the live objects of the heap. To avoid displaying individual objects they classify objects using their type (their Java class). By successive refined requests, it is possible to determine which objects are responsible for memory leaks. KBDB owes much to Jinsight: the definition of a memory leak and the basic framework for using it comes from that work. However, KBDB differs greatly in its representation of objects and in its runtime overhead. Jinsight creates records describing the profiled heap. These records are very large. Three examples are presented for which the records are respectively 32, 46 and 49 times larger than the profiled heaps (110Kb heap is recorded in a 3.5Mb record file, 20Kb in a 0.9Mb record and 25Kb in a 1.2Mb). Even worse, the memory needed by Jinsight is much larger than the recorded files (in the better case, Jinsight uses 370 times more memory than the heap profiled). To visualize our 17Mb heap (the heap size of the **Bootstrap** of Section 5.1.1) we would have required a 6Gb heap!

A very primitive pointer backtracking facility for leak detection was incorporated in the Xerox Portable Common Runtime by the second author around 1995.

Although it required no per object space overhead, this relied on a full search of the heap to trace back one pointer level in the reference chain. In spite of the obvious performance issues, it proved useful in tracking down real problems in a large system. However the implementation was tricky, in that it tended to suffer from "accidental reflection": It was easy to mistreat the local variables used by the backtracking code itself as roots.

7. FUTURE WORK

KBDB is a heap visualization tool. It displays live cells in the heap. Currently, three different visualizations are implemented: a type based classification, an age based classification and a leak detection view. KBDB could be used to display information *à la* Haskell. That is, we could implement a new KBDB module that displays producer graphs such as the one presented in [14]. Our run time provides enough information for this. In addition, we think it could be interesting to provide the user with a statistical information about the heap, such as the average number of GCs survived by certain objects. We think a GC is a wonderful tool for profiling and debugging because a GC keeps track of all pointers. Since a GC is able to scan an entire heap, it can answer questions such as "how many objects are pointing to that other object". This could be used by the debugger to exhibit, on an on-demand basis, sharing properties.

The memory overhead, and to a lesser extent the time overhead, associated with KBDB can be reduced by specializing the GC so that it keeps only information relevant to KBDB and not other information more suited to C program debugging. We expect to report performance numbers based on such a specialization in the final paper.

As we have reported, for large programs such as Bigloo itself, KPROF supplies rough estimates. We are currently exploring another strategy for allocation profiling. We are developing a technique that computes exact figures. Because it will be slower than the current one it won't replace it. It will be an additional tool that could be deployed when current KPROF fails.

Conclusion

In that paper we have presented two memory profilers for the Scheme programming language. The first one KPROF, reports on allocations that take place in program executions. It acts as a regular profiler. A sampling execution is recorded and, afterwards, allocation profiling information is reported. KPROF can point out which functions consume memory. KPROF imposes a low run time overhead and in particular it does not enlarge memory consumption of profiled executions. The second tool, KBDB acts as a debugger. Programs are stepped and, on a on-demand basis, heaps can be visualized. The heaps are then represented by 2 dimensional pictures in which each live cell of a heap is represented by pixels. Zooming in that picture enables cells selection. KBDB is used to fix memory leaks. It slows down executions of about 10 times and it enlarges heap size of about 5 times. However, KBDB is still spare enough to be practical at inspecting the 17MB heap of the bootstrap of our Scheme compiler.

Acknowledgments

Many thanks to Simon Peyton-Jones, Erick Gallesio, and to Céline for their helpful feedbacks on this work and to Al Demers for his suggestion about the back-pointers implementation.

8. REFERENCES

[1] A. Appel, F. Duba, D. MacQueen, and A Tomach. Profiling in the Presence of Optimization and Garbage Collection. Technical Report CS-TR-197-88, Princeton University, November 1988.

[2] H.J. Boehm. Space efficient conservative garbage collection. In *Conference on Programming Language Design and Implementation*, number 28, 6 in SIGPLAN Notices, pages 197–206, 1993.

[3] L. Cannon, R. Elliot, L. Kirchoff, J. Miller, J. Milner, R. Mitze, E. Schan, N. Whittinton, D. Spencer, H. Keppel, and M. Brader. Recommended C Style and Coding Standards, June 1990.

[4] W. De Pauw and G. Sevitski. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proceedings ECOOP'99*, pages 116–134, Lisbon, Portugal, June 1999.

[5] J. Fenlason and B. Baccala. GNU-gprof: user manual. Technical report, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, 1997.

[6] S. Graham, P. Kessler, and McKusik M. gprof: a call graph execution profiler. In *Compiler Construction, SIGPLAN Notices 17(4)*, pages 120–126, 1982.

[7] R. Kelsey, W. Clinger, and J. Rees. The Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

[8] D. Patterson and J. Hennessy. *Computer Organization and Design* The hardware/software interface. Morgan Kaufmann, 2nd edition, 1998.

[9] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. In *Implementation of Declarative Languages*, Paris, France, September 1999.

[10] N. Röjemo. Generational Garbage Collection without Temporary Space Leaks. In *Int'l Workshop on Memory Management*, 1995.

[11] N. Röjemo and C. Runciman. Lag, drag, void and use – heap profiling and space-efficient compilation revised. In *1fst Int'l Conf. on Functional Programming*, pages 34–41, Philadelphia, Penn, USA, May 1996.

[12] C. Runciman and N. Röjemo. Heap profiling for space efficiency. In E. Meijer J. Launchbury and T. Sheard, editors, *LNCS Vol. 1129, 2nd Intl. School on Advanced Functional Programming*, pages 159–183, August 1996.

[13] C. Runciman and N. Röjemo. New dimensions in heap profiling. *Journal of Functional Programming*, 6, 1996.

[14] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, 1993.

[15] P. Sansom. Time Profiling a Lazy Functional Compiler. In *Functional Programming, Glasgow 1993, Workshop in Computing*, Glasgow, 1994. Springer Verlag.

[16] P. Sansom and S. Peyton Jones. Profiling Lazy Functional Programs. In *Functional Programming, Glasgow 1992, Workshop in Computing*, Glasgow, 1993. Springer Verlag.

[17] P. Sansom and S. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *22nd ACMSymposium on Principles of Programming Languages*, San Francisco, USA, January 1995.

[18] M. Serrano. Bee: an Integrated Development Environment for the Scheme Programming Language. *Journal of Functional Programming*, ???(???):???, ????

[19] B. Zorn. The Measured Cost of Conservative Garbage Collection. *Software — Practice and Experience*, 23(7):733–756, July 1993.

[20] B. Zorn and P. Hilfinger. A Memory Allocation Profiler for C and Lisp Programs. In *Usenix conference*, pages 223–237, 1998.