Garbage Collection and Local Variable Type-Precision and Liveness in Java^{TM} Virtual Machines

Ole Agesen and David Detlefs
Sun Microsystems Laboratories
2 Elizabeth Drive
Chelmsford, MA 01824, USA
ole.agesen@sun.com, david.detlefs@sun.com

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610, USA
moss@cs.umass.edu

Abstract. Full precision in garbage collection implies retaining only those heap allocated objects that will actually be used in the future. Since full precision is not computable in general, garbage collectors use safe (i.e., conservative) approximations such as reachability from a set of root references. Ambiguous roots collectors (commonly called "conservative") can be overly conservative because they overestimate the root set, and thereby retain unexpectedly large amounts of garbage. We consider two more precise collection schemes for Java virtual machines (JVMs). One uses a type analysis to obtain a *type-precise* root set (only those variables that contain references); the other adds a live variable analysis to reduce the root set to only the live reference variables. Even with the Java programming language's strong typing, it turns out that the JVM specification has a feature that makes type-precise root sets difficult to compute. We explain the problem and ways in which it can be solved.

Our experimental results include measurements of the costs of the type and liveness analyses at load time, of the incremental benefits at run time of the liveness analysis over the type analysis alone, and of various map sizes and counts. We find that the liveness analysis often produces little or no improvement in heap size, sometimes modest improvements, and occasionally the improvement is dramatic. While further study is in order, we conclude that the main benefit of the liveness analysis is preventing bad surprises.

1 Introduction

The goal of garbage collection (gc) is to reclaim memory allocated to objects that will not be used again. Knowing exactly which objects a program will not access again is equivalent to the halting problem, and is thus not computable. In response, researchers and implementers have devised gc algorithms spanning a wide range of *precision*. We are concerned with gc algorithms that retain objects if and only if they are reachable from a set of *root* references. Such roots include local and global variables. We focus on precision of that part of the root set resulting from local variables, which we term the *local variable root set*, or *l-roots* for short. At the implementation level, local variables are stored in *slots* in *stack frames*. Since our setting is the Java^{TM} programming language, there will be multiple stacks, one for each thread, but many of the same notions apply to single-threaded languages or those requiring closures rather than stacks.

There are at least four degrees of precision one might apply in calculating l-roots:

1. Treat every local variable as an l-root, without regard to type. This is *ambiguous roots* collection [Boehm & Weiser, 1988; Boehm & Shao, 1993; Bartlett, 1988; Bartlett, 1989; Appel & Hanson, 1988], also commonly called *conservative* gc.[1]

2. Use type information to obtain *type-precise* roots, i.e., only those l-roots whose variable is of reference type. This has also been called "accurate", "precise", and "aggressive" (in contrast with "conservative").

3. Extend type-precision by adding *liveness* information from an intra-procedural live variable analysis; we call this *live-precise* collection.

4. More refined liveness analyses, such as inter-procedural analysis.

One can also analyze liveness of heap objects, and fields of heap objects; this is often called *compile-time gc*. Note, though, that we are concerned with which *root references* will be used again, not with which *objects* will be. We do assume that the pointer-containing fields of heap objects can be determined precisely.

A requirement of precise collection is that one must somehow provide the collector with information about the locations of references. This may introduce both performance overhead and extra implementation complexity. We can provide type-precision information via *tags* that make stack frames and their contents self-describing. Tags may be supported by hardware, but more commonly require the generation of extra instructions to check, mask, and insert tags. If tagging is not used, then the system must associate information with each stack frame to indicate the l-roots of the frame; we call such a data structure a *stack map*.

Sun, Sun Microsystems, the Sun Logo, Java, and HotJava are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. Volano, VolanoChat, and VolanoMark are trademarks of Volano LLC.

[1] The term "conservative" has been applied (ambiguously) to ambiguous treatment of heap contents as well as ambiguous determination of roots.

Here we are concerned with how to generate stack maps for code presented as bytecodes used in the Java Virtual Machine (JVM) [Lindholm & Yellin, 1996], hereafter referred to as "Java bytecode," and thus to implement type-precise collection. We are also concerned with the impact of refining type-precision to live-precision. We describe the problems in generating stack maps for Java bytecode and how to solve the problems. We have implemented both type-precise and live-precise collection in the same system, and offer direct comparison of the resulting heap sizes during the execution of a suite of benchmark programs. We have also measured the cost of generating both type-precise and live-precise stack maps, which is of greater relevance in Java virtual machines than many other systems since Java bytecode is loaded (and stack maps are generated) at run time.

2 Related work

Work most closely related to our topics falls into three categories: tagless garbage collection, compile-time analyses for garbage collection, and experimental results related to precision of garbage collection.

Tagless collection goes back at least to Branquart and Lewi's [Branquart & Lewi, 1971] and Wodon's [Wodon, 1971] collectors for Algol-68 and Britton's [Britton, 1975] for Pascal. Branquart and Lewi's collector is notable in that it updated tables at run time as stack slot contents changed. More recently Appel [Appel, 1989] and Goldberg [Goldberg, 1991] considered tagless collection for Standard ML, which is complicated by the presence of polymorphic functions, where the types of their arguments vary from call to call. In a followup paper [Goldberg & Gloger, 1992], Goldberg and Gloger presented a system that uses type unifications to derive types at gc time. If the collector is unable to determine a type for any given variable or field, then that variable or field will not be accessed in the future, and can safely be ignored by the collector. Baker discussed the general idea of using Hindley-Milner unification in this way a bit earlier [Baker, 1990]. A little later, Fradet [Fradet, 1994] extended this sort of collector to include a certain kind of liveness information, based on the intuitive idea that if a polymorphic function does not depend on a type parameter, then it could not actually use any data items of that type. A simple example of this is *length* on lists, which does not examine the list elements, but only counts how many there are. We observe that Fradet's scheme can in some cases determine that object fields are dead, and that it relies on Hindley-Milner style polymorphism, somewhat different from the type system of the Java programming language. Aditya, *et al.* compared, in the context of the polymorphic functional language Id, the cost of type-reconstruction-based collection and conservative collection [Aditya *et al.*, 1994], and found that run-time type reconstruction can have a significant impact.

Tolmach [Tolmach, 1994] and Tarditi, *et al.* [Tarditi *et al.*, 1996] describe schemes that represent the polymorphic type parameters more explicitly, potentially at run time, but frequently optimized away. The Tarditi, *et al.*, object and stack information is similar to ours, and they use liveness information at gc points (but do not report any experience with it). They also handle callee-save registers, which require traversing the stack to find callers' register information in order to type a register saved by a callee; we also encountered that issue in implementing the scheme laid out by Diwan, *et al.* [Diwan *et al.*, 1992]. We need not do that in a bytecode interpretation implementation of the JVM, but if we produced optimized native code, the issue would arise.

Note that many of these schemes are concerned not only with eliminating reference/non-reference tags in the stack, but also with eliminating type tags in heap objects. In object-oriented languages similar to the Java programming language, objects carry full type information to support run-time type discrimination operations. The availability of full type information makes it possible to identify reference fields in heap objects, and thus we are concerned only with reference/non-reference distinctions for roots.

The Java programming language, as it currently stands, does not have parametric polymorphism, though there is considerable discussion of possible techniques for adding parameterized types and classes. If these were implemented with shared code bodies, then some of the same stack map generation issues would arise as do with Standard ML polymorphic functions [Agesen *et al.*, 1997].

In the area of procedural and object-oriented languages, Diwan, *et al.*, described a scheme for building stack maps for Modula-3 [Diwan *et al.*, 1992], which deals with reconstructing pointers to heap objects from offsets and other optimized representations, which come about at least in part from the ability to pass object fields by reference in calls. They also considered how stack maps might be compressed to save space. A related topic is ensuring that compiler optimizations will not effectively hide live pointers from a collector, and has been considered by Boehm and Chase (at least), separately and together [Chase, 1988; Boehm, 1991; Boehm & Chase, 1992; Boehm, 1996]. The relatively simple and highly constrained model of references presented by the JVM avoids the optimization-induced problems these other works address, such as interior and derived pointers. However, once one considers generating native code from Java bytecode, the optimization issues may arise. In other work, Boehm and Shao considered how to construct a useful conservative approximation of object type information at run time for a conservative collector [Boehm, 1993]. Finally, Thomas, with Jones, built routines for tracing stack frames, moving from an interpretive to a compiled model for stack maps [Thomas, 1993; Thomas & Jones, 1994; Thomas, 1995].

Clearly the notion of tagless collection is now fairly well developed; we certainly do not claim that building stack maps for Java bytecode is a novel idea or even that the difficulties peculiar to this context require deep new approaches.

Turning to compile-time analyses for garbage collection, there has been much work done on such analyses for functional and applicative languages [Barth, 1977; Bruynooghe, 1987; Chase *et al.*, 1990; Deutsch, 1990; Foster & Winsborough, 1991; Hamilton & Jones, 1991; Hamilton, 1993; Hamilton, 1995; Hederman, 1988; Hicks, 1993; Hudak, 1986; Hudak, 1987; Hughes, 1992; Inoue *et al.*, 1988; Jensen & Mogensen, 1990; Jones & le Métayer, 1989; Jones & White, 1991; Jones & Tyas, 1993; Jones, 1995; Mohnen, 1995; Mulkers, 1993; Mulkers *et al.*, 1994; Wadler, 1984]. There are two important ways in which that work does not carry over to our situation. The most obvious difference is that the Java programming language is not functional, so the patterns of allocation, mutation, and heap use in general might be quite different. A more subtle difference is that most of the work on compile-time gc is focused on showing (statically) that certain *objects* are not reachable and can be reused or collected immediately. We are concerned only with whether *references in local variables* will be used again, which is a weaker property.

Some schemes are more similar to our liveness analysis. For example, Appel described a continuation-passing style compiler for Standard ML [Appel, 1992], which effectively removed dead variables from closures. This resulted in there being more closures (one for each set of live variables), prompting some to call for closure combination to save on closure allocation, but Appel has noted that this would risk retaining more allocated heap objects because of dead variables. Shao and Appel devised an arguably better scheme, based on control and data flow analyses, that shares closures heavily but still guarantees that dead variables are unreachable [Shao & Appel, 1994]. Thomas's compiler-generated tracing routines [Thomas, 1993; Thomas & Jones, 1994; Thomas, 1995] take liveness into account for closures, and a given closure may be traced more than once, with different livenesses for the variables, to trace all live references.

Again, we do not claim that the idea of using liveness information is new. However, we found no reports of its use for procedural or object-oriented languages (other than an indication that Chase has built a collector similar to ours for a JVM [Chase, 1997]). On the other hand, it has likely been done before but simply not reported. *Most significantly, we have found no previous measurement of the impact of live variable analysis*, only anecdotal discussions in the context of functional languages, which leads us to the topic of experimental results.

Overall, the improvements obtained with compile-time gc for functional languages have been minor. For example, Jones [Jones, 1995] obtained an 8% reduction in bytes allocated for Haskell, reducing overall execution time by 4.5%. Likewise, Wentworth found that conservative gc generally did well [Wentworth, 1990]. On the other hand, he made a telling observation: *sometimes* conservatism makes a big difference.[2] Similarly, it appears that the primary benefit of liveness analyses is in reducing the likelihood of surprising space retention. Evaluations of conservative gc have been in terms of the incidence of non-reference values looking like references and thus causing garbage to be retained. We note that such evaluations overlook the storage that can be reclaimed by omitting dead variables from the root set; that is, type-precision and live-*precision constitute two separate precision improvements* over ambiguous roots gc.

Our contributions: From this overview of related work, we conclude that our primary contribution lies in reporting measurements of the impact of liveness analysis for a procedural object-oriented language. We previously reported in more detail on the difficulties in producing stack maps for Java bytecode [Agesen & Detlefs, 1997], and summarize that work here, extending it with the liveness analysis.

3 Stack maps and gc points

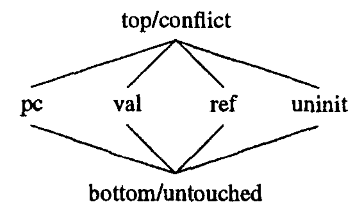
The contents of a stack slot can change during the execution of a Java method. Slots, with the exception of those occupied by arguments to the method, start uninitialized. Thus a simple way in which slot types can change is from uninitialized to containing a value of a particular type. However, a Java compiler is permitted to (and indeed existing ones do) store source variables with disjoint live ranges in the same slot. Thus a slot can contain values of different types at different points in the execution of a method; some of those types may be reference types and others may be non-reference types.

At this juncture, we stress that we are concerned with processing Java *bytecode*, loaded at run time. Thus we are concerned not so much with the Java *programming language* specification [Gosling *et al.*, 1996] as with the Java *virtual machine* specification [Lindholm & Yellin, 1996]. (The instruction set also has been described separately by Gosling [Gosling, 1995].) Java bytecode must pass well-formedness tests performed by a run-time *bytecode verifier*; we assume that we deal only with such well-formed Java bytecode methods. Some of the relevant verified properties are:

> A type can be calculated for each local variable and stack temporary at each instruction of a method, using a straightforward data flow analysis over the lattice of object types, augmented with some non-object types. This implies that types may depend on program point, but *not* on the path by which the program point is reached. We call this the *Gosling property* below, since it was first stated explicitly by Gosling [Gosling, 1995].[3]
>
> The types of the operands for each instruction will satisfy the instruction's type constraints. In particular, integer, floating point, and reference handling instructions are distinct and type checked.
>
> The simple static data flow analysis suffices to show that no uninitialized variable is ever used.

Following Diwan, *et al.* [Diwan *et al.*, 1992], we do not generate stack maps for every instruction. Rather, we restrict the VM implementation so that collection can occur only at certain *gc points*. These must include the allocation bytecodes. In the absence of a whole-program style inter-procedural analysis, which would be difficult in the face of the JVM's dynamic loading capabilities, one must also include calls as possible gc points. Finally, to insure that one can stop each thread if another thread initiates collection, each loop should contain a gc point. This is easily accomplished by making backward branches gc points. Beyond that, the choice of gc points is an engineering decision. Discussion of techniques to advance threads to gc points falls outside the scope of this paper.

Bytecode verification uses a full type lattice, but for stack map generation we need to know only whether a slot contains a reference or non-reference, etc., and not the specific type. We do, however, need to track program counter (pc) values, as will be described in Section 4. We may thus use this lattice:



To start the data flow analysis for a method, we set all variable values to *bottom*, except for the entry point to the method, where we

[2]Proponents of conservative gc argue that cases where it performed poorly for Wentworth can be largely prevented by avoiding allocation in regions of virtual memory whose addresses look like numeric values in use by the program at hand.

[3]As we will see, the Gosling property does not *always* hold; the complete story is more complex.

set arguments to *val* or *ref* according to their incoming type, and remaining variables to *uninit*. Any of the well-known data flow analysis computation algorithms will suffice; we used a simple work-list algorithm.

4 The jsr problem

Unfortunately, we have not yet told the complete story. The JVM specification explicitly allows one exception to the Gosling property. The Java bytecode instruction set includes a pair of operations called `jsr` and `ret`. The `jsr` instruction jumps to an address specified in the instruction and pushes a return address value on the operand stack[4] of the current method. The `ret` instruction specifies a local variable that must contain a return address, and jumps to that return address.

The intended use of these bytecodes is in the implementation of the

`try {` *body* `} finally {` *handler* `}`

construct of the Java programming language, in which *handler* is executed no matter how *body* is exited. The *handler* would be translated as a *jsr subroutine:* a "mini-method" within the method. Every instruction that exits *body*, such as `return` or `throw` statements or "falling off the end", would be preceded in the translation by a `jsr` to the *handler* subroutine, which would store the pushed return address in a local variable, perform the work of *handler*, then perform a `ret`. Although a jsr subroutine resembles a real method, there is a crucial difference: it executes in the same stack frame as its containing method and has access to all the local variables of the method.

The JVM specification for verification of jsr subroutines contains an explicit exception to the Gosling property [Lindholm & Yellin, 1996, p. 136]: the bytecode verifier permits any local variable *v* that is neither read nor written in a jsr subroutine to retain its type across a `jsr` to that subroutine.

This seemingly reasonable weakening of the Gosling property causes serious difficulty for precise garbage collection. Consider a case in which there are two `jsrs` to the same jsr subroutine. At one `jsr`, local variable *v* is being used to hold an integer, and at the other, it holds a reference. Should a garbage collection occur while a thread is in the jsr subroutine, a simple program-counter based stack map scheme cannot determine if *v* contains a reference, since the stack layout is now path dependent. Simply disallowing garbage collections for the duration of the jsr subroutine is not an option since `try-finally` handlers can perform arbitrary computation, including calling methods that may execute indefinitely and allocate an unbounded number of objects.

5 Possible solutions

There are several possible solutions to the jsr problem. An obvious one is to change the JVM specification to remove the Gosling property exception for jsr subroutines, which would simplify bytecode verification as well as stack map generation. However, removing the exception might increase the size of some stack frames, and in any case it would be politically difficult to make changes to the specification that would invalidate any existing code.

A second class of solutions rewrites the bytecodes to eliminate any violation of the Gosling property. One specific technique is to replicate jsr subroutines that are called from sites with different stack maps, so that each replica is called only from sites with the same stack map. Since jsr routine calls can be nested, this can result in exponential code duplication. While the occurrence of the jsr problem is rare for many programs, even one case of exponential expansion might be fatal. Also, we anticipate that exception handling features will be used more in the future than they are now, as programmers become more accustomed to them. Rather than duplicating code, we chose to *split variables*; we describe this in more detail below.

A third class of solutions is to allow the Gosling property violation, and add additional information to stack maps so that one can determine the nesting of jsr subroutine calls in progress, and combine stack map information through the `jsr` call chain. One of us is pursuing this approach, which has the advantages of not affecting the VM specification, of not requiring any bytecode rewriting, and of not imposing any normal case overhead in method execution. It is more complex, and may require slightly more work at collection time (probably not enough to matter), and slightly larger and more complex stack maps (again, probably not significant in practice).

6 Bytecode rewriting to split conflicting variables

Our first step was to refine the lattice used in the data flow analysis that computes stack maps to record not only that a conflict occurs but also the kind of conflict. Essentially, we used the power set lattice, adding cases for *ref-nonref, ref-uninit, val-uninit*, etc. This is easy to implement using bit vector operations.

We need this more detailed information because we resolve conflicts between references and uninitialized values (*ref-uninit* conflicts) differently from conflicts between references and non-reference values (*ref-nonref* conflicts):

> *ref-uninit* conflicts are eliminated by prepending code to the start of the method to initialize the variables to null.
>
> *ref-nonref* conflicts are eliminated by splitting the variables.
>
> *top* (*ref-nonref-uninit*) conflicts are resolved by a combination of the above two actions: we first introduce initializations to null, eliminating the *uninit* conflicts, and then split to eliminate the *ref-nonref* conflicts.[5] Note that the initialization to null will be associated with the part of the split variable that has a *ref* type, and not with the part that has a *non-ref* type.

We extend the stack map computation for a method *m* as follows. We alternate data flow analysis and conflict elimination, iterating until all conflicts have been eliminated. During the data flow analysis, a variable *varsToInit* holds a set of reference-containing variables requiring initialization. It is initially empty. The data flow analysis treats variables in the set as holding initialized reference values at the start of the method. Each iteration initializes a variable *varsToSplit* to the empty set of variables. This set will hold

[4] For the present discussion, it is unnecessary to distinguish between stack frame slots holding operand stack values and stack frame slots holding local variables: we think of them as two separate sets of local variables, one being addressed from the stack frame base, the other being addressed from the top of stack pointer.

[5] We ignore *val-uninit* conflicts since they are irrelevant to garbage collection.

variables that were found to hold a ref-nonref conflict at a point when they were *used*. (Such a conflict can happen only when the Gosling property is violated, i.e., through the type merging that jsr subroutines can induce.)

The stack map computation then proceeds as described previously, except in its handling of conflict values. A use of a variable whose value in the data flow analysis is the *ref-uninit* conflict value causes the variable to be added to *varsToInit*. A use of a variable holding the *ref-nonref* conflict value adds the variable to *varsToSplit*. If a use is of a value having *both* kinds of conflicts, we add the variable to *varsToInit* only.[6]

At the end of an iteration, *varsToSplit* is checked. If it is non-empty, then each variable in it is *split*. To split local variable *n*, we increase the number of local variables allocated in stack frames for method *m* by one; let *nn* be the number of the newly allocated local variable. We then examine the bytecodes for method *m*, modifying them so that instructions that use the original variable *n* to hold references are unchanged, but non-reference uses are changed to use variable *nn* instead.[7] It is a happy property of the Java bytecode instruction set that instructions have sufficient type information encoded in their opcodes to determine locally whether a given instruction uses a local variable as a reference, making the rewriting fairly simple.[8]

There is one more complication. Rewriting instructions can cause instruction positions and lengths to change, so code must be relocated, branch offsets updated, etc., a process we will not describe in detail.

If any uses of conflict variables are detected, at least some are repaired by this variable-splitting process or by addition to the *varsToInit* set. The next iteration of the loop may still find some conflicts in the rewritten code (perhaps a variable has both ref-uninit and ref-nonref conflicts), causing another iteration, or it will detect no conflicts and successfully generate the stack maps.[9]

The rewriting may fail, in the following ways. Allocating new local variables could exceed the limit on the number of locals in a method imposed by the bytecode instruction set. Widening instructions could conceivably cause a method to exceed the maximum method size. In such cases, the VM would have to somehow indicate an error akin to a verification error. Such programs would be exceedingly unlikely to occur in practice.

The performance of the bytecode rewriting part of the process is not a crucial issue since with the most commonly used compiler, javac, very few methods need rewriting. In the benchmark suite we used for the comprehensive measurements we describe in more detail later, we split only six variables (five in one program, one in another). *Ref-uninit* conflicts were somewhat more common. An average of 20 variables per program required initialization. These were concentrated in two programs that used the Java Abstract Windows Toolkit.

On the other hand, the performance of the first iteration of the analysis, the only iteration required by most methods, is of some interest since it will be performed for any dynamically loaded code before that code is executed.[10] Further below we report some measures of the cost of the data flow analysis.

7 Live variable analysis

The additional live variable analysis is straightforward, and requires only a two-element lattice. Since liveness is a backwards flow property, we unfortunately cannot compute liveness by augmenting the forward flow type analysis lattice. We observe, though, that the liveness analysis may eliminate some conflicts at gc points. However, by the Gosling property, except at `jsr` instructions and in jsr subroutines, ref-uninit and ref-nonref conflicts indicate variables that must be dead, so such items should be dropped from stack maps anyway. Still, the liveness analysis will identify dead references.

Since the JVM instruction set is stack-oriented, data movement operations such as the assignment `x = y;` present themselves as pushes and pops. Our live variable analysis propagates liveness information through local variables and stack temporaries. This means that in the data flow analysis, the live/dead values for different variables are coupled, so the height of the lattice is the number of variables; i.e., variables cannot be analyzed separately from one another. It is unlikely that one would see worst case iteration of the algorithm in practice, though.

On the run-time side, what should the collector do with dead reference variables? Obviously it should ignore any dead reference for tracing purposes. Less obviously, if a program is run in the presence of a debugger, the collector has three options. It can trace dead references (so they can still be examined in the debugger for as long as possible); it can set dead references to null (so that the debugger will not try to follow a reference that may be invalid after collection); or it can treat dead references as "weak" references, retaining their value if and only if the referent objects are otherwise reachable.

It also appears that omitting dead references when collecting can expose program bugs. For example, suppose object *x* refers to object *y*, object *x* has a finalizer that uses object *y*, and the program's last reference to *x* becomes dead while the program is still using *y*.[11] If a gc occurs at this point, *y* may be accessed concurrently by the finalizer and by the main program code, a form of concurrency that may surprise the programmer. This does not appear to violate the Java programming language specifications, so we consider it to be legal. We also observe that many optimizing transformations can expose (or hide) bugs, so our liveness analysis is far from unique in this regard.

[6]Late in the game we realized that it is also correct, and probably slightly better, to add the variable only to *varsToSplit*. A later iteration will add it to *varsToInit*, but only if a ref-uninit conflict remains after splitting the variable.

[7]The other choice, where reference uses of *n* are changed to use *nn* and non-reference uses are unchanged, is equivalent.

[8]There is one exception to this property: the `astore` instruction is usually used to pop a reference (an *address*, hence the prefix letter *a*) from the operand stack and store it in a local variable, but it may also be used to do the same with return addresses pushed on the stack by `jsr` instructions. Fortunately, the data flow analysis already maintains sufficient state to determine whether the operand stack top at the point of the `astore` is such a return address, so this complication is easily circumvented.

[9]Again, late in the game, we realized that *varsToSplit* can only be non-empty after the first iteration of the overall process. So we need perform the data flow analysis no more than two times. Furthermore, if we are willing possibly to over-estimate ref-uninit conflicts by allowing variables to be added to both *varsToSplit* and *varsToInit*, we need only perform the data flow analysis once. Doing so might significantly reduce time needed to generate stack maps for methods that require rewriting.

[10]While it is conceivable that one might generate stack maps at gc time, it is problematic because implementations of the data flow analyses will tend to allocate heap storage, which is not generally possible during gc. It is also possible to pre-analyze code and insert gc stack maps into class files as additional attributes, but this is possible only for local trusted class files.

[11]This example was supplied by an anonymous reviewer.

7.1 Why liveness analysis?

Why might it be important to include a live variable analysis? One school of thought is that the dead variable heap object retention problem can be fixed simply by having programmers insert assignments of null at the right places. There are several problems with this view. First, it introduces an overhead all the time to address a situation that occurs relatively rarely (gc). It is more efficient to have the gc treat the slots as containing null than it is to set them to null. Second, why should programmers have to waste their time even thinking about an issue like this when an automated tool can address it? It is not as if programmers are likely to *want* dead objects retained, and thus perhaps desire control over this behavior. Third, even if programmers insert assignments of null, an optimizing compiler might remove them, since they are assignments to dead variables![12] Finally, and we think this is the nail in the coffin, there are cases where it is virtually impossible for the programmer to do the assignment at the critical moment. A good example of this is a method call such as `v.m(x)`, where `x` is the last live reference to some sizeable object subgraph, variable `x` is dead after the call, and method `m` also reaches a point where it no longer uses `x`. This is a particularly disturbing possibility, since a call has indefinite duration and may be deep in the stack, thus retaining garbage for quite a long time.

One of us ran into a concrete example when working on a theorem-proving system in Modula-3. Rewriting the essential part in the Java programming language, the pertinent code was:

```
bool proveTheorem(InputStream is) {
    Sexp sx = Sexp.read(is);
    Pred p = Pred.sxToPred(sx);
    return refute(Pred.not(p));
}
```

The salient feature of this code is that the `Sexp` form was used only because there was a convenient library available to read expressions in LISP S-expression form. The S-expression form was immediately converted to a predicate form and discarded. In realistic situations, `sx` might refer to a megabyte or more of S-expression data that is dead, across a long running call to `refute`. Even more interesting, the actual code was written in a more functional style, and the dead variable was actually a compiler temporary! This made the problem quite difficult to discover and remedy. Furthermore, when the original functional form was rewritten to the form shown above, and then modified by explicitly assigning `null` to `sx` after its last use, the problem still persisted. We speculate that this was because the Modula-3 collector was an ambiguous roots collector, and another copy of the `sx` pointer, which had been passed in the call to `Pred`, was lying in the stack frame for `refute`, or some place even further towards the top of the stack. Our fix was to null out the entire `sx` structure after building the `Pred` form.

Our reason for including this story is to make it clear how difficult it can be to locate and resolve problems of unexpected storage retention.

8 Experiments

In this section we give experimental data obtained on a 296 Mhz Ultra SPARC with 512 Mbytes of memory, running Solaris 2.6. The programs we measured are a collection of benchmarks under consideration for a SPEC suite to measure Java platforms.[13] We had to exclude four of the programs because thread-library issues prevented us from running them correctly; we excluded two more because they allocated too little storage to be interesting in this study. Finally, we added *ellisgc*, a GC stress test program, that John Ellis sent to us. While it bears some relationship to *gcbench*, they react differently under liveness analysis, so we felt it useful to include both.

The VM we used is based on the Javasoft JDK VM, modified to (among other things) support generation and use of our stack maps. Note that since we are comparing the amount of reachable heap data as we vary the stack root set, the actual gc technique is irrelevant (it happened to be mark-sweep).

8.1 Cost of type and liveness analysis

We measured the elapsed time used by stack map generation and liveness analysis while running our benchmark suite on an otherwise idle workstation. For purposes of comparison, we also measured total time, time consumed by class loading, and time for bytecode verification (running the VM in a mode where all classes are verified). Table 1 displays these measurements. The "Stack map/Loading" column divides stack map computation by class loading time, and the "Verification/Stack map" column expresses verification time as a multiple of stack map computation time. The "Average" row gives geometric means for the columns containing ratios.

Just as verification can be done once (off-line) for local trusted classes, one could similarly compute stack map information off-line for such classes, speeding up program startup accordingly. On the other hand, for classes obtained over a network, possibly from untrusted sites, verification is necessary, and as our numbers show, usually dominates stack map computation time by an order of magnitude.

We also measured how much liveness analysis increased the cost of stack map generation. The increase was quite uniform, between 54% and 58% over all the benchmarks. A further breakdown showed that the forward analysis and backward analysis were quite similar in cost. However, the liveness analysis is able to reuse data structures created for the type analysis (basic blocks, etc.), thus decreasing its incremental cost.

8.2 Stack map size measurements

We present a range of stack map size and related statistics in Tables 2 and 3. In these and subsequent measurements, we have added runs of three more programs. The *spreadsheet* program is a prototype financial calculation engine, obtained via private communication. The *hotjava* run is part of a morning's exploration with the HotJava web browser. Both of these programs have elaborate graphical user interfaces. The *volano* run gives the behavior of the server-side program in Volano LLC's VolanoMark 1.0 benchmark, over several invocations of a client-side program provided in the benchmark that imposes a workload on the server. This benchmark is intended to estimate the performance of the real

[12]David Chase brought this to our attention; he mentions the possibility in passing in his dissertation [Chase, 1987].

[13]Those benchmarks selected by SPEC (if any) may have different versions and/or workloads, so our results cannot necessarily be compared meaningfully with any SPEC results. Our purpose was only to compare different gc algorithms on a set of programs, not to compare platforms differing in any other way.

Benchmark	Total run time *(sec)*	Loading *(msec)*	Stack map *(msec)*	Stack map/ Loading	Verification *(msec)*	Verification/ Stack map
compress	82.0	269.6	172.0	0.63	1838.6	10.6
jess	9.9	128.2	130.3	1.01	1147.8	8.8
linpack	89.9	99.1	97.8	0.98	711.5	7.2
newmst	32.2	96.2	91.8	0.95	821.0	8.9
raytrace	21.5	114.0	118.8	1.04	1107.8	9.3
cst	17.9	114.1	113.5	0.99	729.2	6.4
db	7.6	101.8	101.5	0.99	836.8	8.2
si	12.3	108.4	106.0	0.97	632.2	5.9
anagram	3.8	97.2	94.8	0.97	732.7	7.7
gcbench	2.8	98.4	96.0	0.97	769.2	8.0
javac	7.4	143.5	171.2	1.19	4531.7	26.4
deltablue	23.4	105.2	98.2	0.93	723.2	7.3
mpegaudio	91.6	107.2	146.9	1.37	1479.7	10.0
jack	21.4	156.6	175.0	1.11	2102.4	12.0
tsgp	350.4	98.0	93.2	0.95	614.3	6.5
ellisgc	8.8	40.2	62.3	1.54	936.3	15.0
Average				1.02		9.1

Table 1: Comparison of stack map computation with class loading and bytecode verification

VolanoChat chat server on a given Java platform, and is available at `http://www.volano.com/mark.html`.

In Table 2, the "Code size" column shows the number of bytecodes in all methods executed in the run. The "GC points" column gives the total number of *gc points*, bytecode instructions for which stackmaps were computed, for the methods executed in the run. The "Code bytes/gc point" column gives the ratio of these two numbers, an estimate of the interval between bytecode instructions requiring stack maps. The "Slots" column shows the sum of the number of local variable and operand stack slots in use at all gc points, and the average number for each gc point. The "Refs" column shows how many of these slots contained references, and the "Live" columns shows how many of these were live. The last two columns show the fraction of slots that contained references, and the fraction of reference slots that were live. The "Average" row gives geometric means for the rows representing ratios. Table 3 presents the same information, averaged over methods instead of gc points. Again, the "Average" row gives geometric means for the rows representing ratios. Roughly speaking, a little more than half of all slots are references, and approximately 3/4 of these are live. In the particular system in which we did these experiments, stackmaps consumed an average of 57% as much space as the bytecode itself. However, the representation uses no compression, so we believe this overhead could be substantially reduced.

8.3 Run-time heap size measurements

To measure the impact of liveness analysis, we ran the suite of benchmark programs on our modified JVM. This system uses our stack maps to trace stack frames in either a type-precise or live-precise manner. In fact, both levels of precision are available in the same system, so we compared them directly, as follows.

After every 100K words of allocation, we invoked the mark phase of the type-precise collector and determined the number of words of objects marked. We then reset the mark bits and invoked the mark phase of the live-precise collector and determined the number of words of objects it marked. We did a sweep only when the allocation area was exhausted.

For each precision we can construct a function giving the heap size over time, where time is measured in words allocated and is sampled every 100K words. We connect the points of each function and compute the integral under the function's curve, which gives us the space-time product of the run. We report total space-time products, and the ratio of those products, for the two levels of precision in Table 4. We also report for each benchmark the (geometric) mean of the ratios of the heap sizes at each sampled point during the run. Finally we report geometric means of each column of the table, i.e., across all benchmarks.[14] We present some sample curves showing reachable data with and without liveness analysis in Figure 1. The *ellisgc* run shows the most dramatic improvement from liveness analysis of the programs we ran. The *volano* run is more typical. (Each "hump" corresponds to the response of the server-side program to one invocation of the client-side simulated load.)

Overall, liveness information reduces the time-space product by an average of 11%. This result is skewed by the *ellisgc* program. That program is somewhat contrived and is intended to challenge garbage collectors. However, it does not *intentionally* include dead variables. Still, we include averages omitting *ellisgc* and see that the time-space product improves by an average of 3.6%. We note that it is not necessarily reasonable to reject *ellisgc* from the results, since dead variable space retention is likely to be an occurrence that is usually not too bad, but occasionally terrible. It is interesting to see that almost every program we ran shows a measurable difference, so some degree of dead variable space retention appears to be common.

A separate point is that this is a non-generational collector. In a

[14]We use geometric rather than arithmetic means since they are more suitable for comparing ratios. The geometric mean of *n* items is the *n*th root of their product, or, equivalently, the anti-logarithm of the arithmetic mean of their logarithms.

Benchmark	Code size	GC	Code bytes/	Slots		Refs		Live		Refs/	Live/
	(bytes)	points	gc point	#	per pt	#	per pt	#	per pt	Slots	Refs
compress	176206	5762	30.6	40013	6.94	21116	3.66	16576	2.88	0.528	0.785
jess	46909	9146	5.1	72123	7.89	40681	4.45	29282	3.20	0.564	0.720
linpack	29544	5315	5.6	39721	7.47	20066	3.78	15834	2.98	0.505	0.789
newmst	27148	5095	5.3	37006	7.26	19162	3.76	15165	2.98	0.518	0.791
raytrace	100595	7553	13.3	62251	8.24	33301	4.41	25673	3.40	0.535	0.771
cst	98964	7275	13.6	62927	8.65	27117	3.73	20982	2.88	0.431	0.774
db	31132	5834	5.3	41658	7.14	21741	3.73	17038	2.92	0.522	0.784
si	33288	6289	5.3	50410	8.02	24419	3.88	18503	2.94	0.484	0.758
anagram	28302	5336	5.3	38342	7.19	19887	3.73	15709	2.94	0.519	0.790
gcbench	29361	5553	5.3	40092	7.22	20502	3.69	16019	2.88	0.511	0.781
javac	159252	18257	8.7	144700	7.93	86191	4.72	60541	3.32	0.596	0.702
deltablue	90989	5797	15.7	39669	6.84	21678	3.74	16918	2.92	0.546	0.780
mpegaudio	74885	9634	7.8	66239	6.88	33835	3.51	28070	2.91	0.511	0.830
jack	136373	14249	9.6	95938	6.73	52972	3.72	39542	2.78	0.552	0.746
tsgp	27993	5220	5.4	38346	7.35	19492	3.73	15499	2.97	0.508	0.795
ellisgc	9644	1637	5.9	9421	5.76	4921	3.01	3281	2.00	0.522	0.667
quantum	357035	61166	5.8	419349	6.86	240788	3.94	166998	2.73	0.574	0.694
hotjava	399746	67248	5.9	1005937	14.96	597734	8.89	520071	7.73	0.594	0.870
volano	97797	7136	13.7	42357	5.94	24241	3.40	18098	2.54	0.572	0.747
Average			7.9		7.48		3.97		3.04	0.530	0.766

Table 2: Measured stack map size and related statistics

Benchmark	Methods	Bytes/method	GC points/method	Slots/method	Refs/method	Live refs/method
compress	369	477.52	15.62	108.44	57.22	44.92
jess	514	91.26	17.79	140.32	79.15	56.97
linpack	348	84.90	15.27	114.14	57.66	45.50
newmst	335	81.04	15.21	110.47	57.20	45.27
raytrace	499	201.59	15.14	124.75	66.74	51.45
cst	427	231.77	17.04	147.37	63.51	49.14
db	376	82.80	15.52	110.79	57.82	45.31
si	384	86.69	16.38	131.28	63.59	48.18
anagram	346	81.80	15.42	110.82	57.48	45.40
gcbench	361	81.33	15.38	111.06	56.79	44.37
javac	1124	141.68	16.24	128.74	76.68	53.86
deltablue	404	225.22	14.35	98.19	53.66	41.88
mpegaudio	535	139.97	18.01	123.81	63.24	52.47
jack	759	179.67	18.77	126.40	69.79	52.10
tsgp	337	83.07	15.49	113.79	57.84	45.99
ellisgc	182	52.99	8.99	51.76	27.04	18.03
quantum	5843	61.10	10.47	71.77	41.21	28.58
hotjava	4669	85.62	14.40	215.45	128.02	111.39
volano	596	164.09	11.97	71.07	40.67	30.37
Average		117.16	14.91	111.63	59.14	45.28

Table 3: Measured stack map size reported per method

Benchmark program	Time space w/out liveness *(M byte²)*	Time space with liveness *(M byte²)*	Ratio	Mean ratio
compress	61128242	57363160	0.9384	0.9330
jess	868824	825482	0.9501	0.9472
linpack	3355585	3292842	0.9813	0.9497
newmst	78055	73266	0.9386	0.9386
raytrace	37817490	37508324	0.9918	0.9865
cst	12504270	12258149	0.9803	0.9685
db	7978578	7882272	0.9879	0.9736
si	22599024	20103152	0.8896	0.8896
anagram	27918894	27712322	0.9926	0.9858
gcbench	226114414	224068599	0.9910	0.9888
javac	9186308	9007787	0.9806	0.9759
deltablue	3741663	3426734	0.9158	0.9033
mpegaudio	76096	71294	0.9369	0.9369
jack	9716830	9419524	0.9694	0.9640
tsgp	710108	686164	0.9663	0.9626
ellisgc	289690805	61671602	0.2129	0.0904
spreadsheet	154266857	154255015	0.9999	0.9999
hotjava	929502587	929484329	1.0000	1.0000
volano	39938934	38008136	0.9517	0.9523
Average	11778354	10487525	0.8904	0.8462
Average, without ellisgc	9858598	9504470	0.9641	0.9582

Table 4: Heap sizes with and without liveness

generational collector the benefit of liveness analysis may turn out to be greater, since one might reduce the volume of tenured garbage, which takes longer to collect.

9 Conclusions

We found that adding a live variable analysis to a type-precise garbage collector, so as to increase its precision, reduced heap size (time-space product) by an average of 11% for a suite of benchmark programs, with most programs showing some difference and a few showing more dramatic differences. Liveness analysis appears to offer minimal benefit to many programs, but it is important in reducing the possibility of surprisingly large volumes of retained garbage. Preliminary measurements indicate that the cost of generating live-precise stack maps for Java bytecode is about 50% greater than the cost of generating only type-precise stack maps. We also described some technical difficulties in generating stack maps for Java bytecode so as to accomplish type-precise collection, and indicated several solution approaches.

Acknowledgments. The greatly simplifying realization that *ref* and *nonref* uses of variables can be distinguished based only on the opcode of the using instruction came about in a discussion with Boris Beylin, Ross Knippel, and Bob Wilson. John Rose explained to us the behavior of the javac compiler when translating the `try-finally` construct. Hans Boehm, David Chase, and Richard Jones were particularly helpful in referring us to related work. Eliot Moss's participation in this work is supported in part by gifts from Sun Microsystems Laboratories and grants from the National Science Foundation.

References

[Aditya *et al.*, 1994] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In [LFP, 1994], pp. 12–23.

[Agesen & Detlefs, 1997] Ole Agesen and David Detlefs. Finding references in Java stacks. Tech. Rep. SML-E-97-67, Sun Microsystems Laboratories, Chelmsford, MA, USA, Oct. 1997. Presented at the OOPSLA '97 workshop on garbage collection.

[Agesen *et al.*, 1997] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding type parameterization to Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)* (New York, Oct.5–9 1997), vol. 32, 10 of *ACM SIGPLAN Notices*, ACM Press, pp. 49–65.

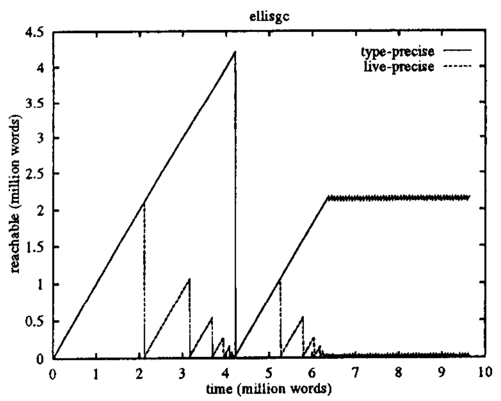
[Appel, 1989] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation* 2 (1989), 153–162.

[Appel, 1992] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992, ch. 16, pp. 205–214.

[Appel & Hanson, 1988] Andrew W. Appel and David R. Hanson. Copying garbage collection in the presence of ambiguous references. Tech. Rep. CS-TR-162-88, Princeton University, 1988.

[Baker, 1990] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming* (Nice, France, June 1990), ACM Press, pp. 218–226.

[Barth, 1977] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM 20*, 7 (July 1977), 513–518.



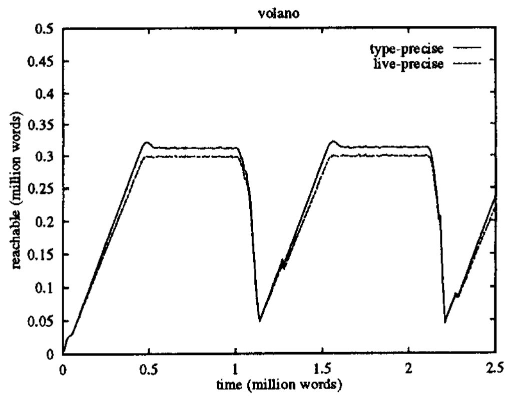


Figure 1: Sample curves showing reachable data over time

[Bartlett, 1988] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, CA, Feb. 1988. Also in Lisp Pointers 1, 6 (April–June 1988), pp. 2–12.

[Bartlett, 1989] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, Oct. 1989. Sources available in ftp://gatekeeper.dec.com/pub/DEC/CCgc.

[Boehm, 1991] Hans-Juergen Boehm. Simple GC-safe compilation. In *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems* (Oct. 1991), Paul R. Wilson and Barry Hayes, Eds.

[Boehm, 1993] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation* (Albuquerque, New Mexico, June 1993), vol. 28(6) of *ACM SIGPLAN Notices*, ACM Press, pp. 197–206.

[Boehm, 1996] Hans-Juergen Boehm. Simple garbage-collector safety. In [PLDI, 1996], pp. 89–98.

[Boehm & Chase, 1992] Hans-Juergen Boehm and David R. Chase. A proposal for garbage-collector-safe C compilation. *Journal of C Language Translation* (1992), 126–141.

[Boehm & Shao, 1993] Hans-Juergen Boehm and Zhong Shao. Inferring type maps during garbage collection. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems* (Oct. 1993), Eliot Moss, Paul R. Wilson, and Benjamin Zorn, Eds.

[Boehm & Weiser, 1988] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience 18*, 9 (1988), 807–820.

[Branquart & Lewi, 1971] P. Branquart and J. Lewi. A scheme of storage allocation and garbage collection for Algol–68. In [Peck, 1971], pp. 198–238.

[Britton, 1975] Dianne Ellen Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[Bruynooghe, 1987] Maurice Bruynooghe. Compile-time garbage collection or How to transform programs in an assignment-free language into code with assignments. In *Program specification and transformation. The IFIP TC2/WG 2.1 Working Conference, Bad Tolz, Germany*, L. G. L. T. Meertens, Ed. North-Holland, Amsterdam, April 15–17, 1986 1987, pp. 113–129.

[Chase, 1997] David Chase, Nov. 1997. Personal communication.

[Chase, 1987] David R. Chase. Garbage collection and other optimizations. Tech. rep., Rice University, Aug. 1987.

[Chase, 1988] David R. Chase. Safety considerations for storage allocation optimizations. *ACM SIGPLAN Notices 23*, 7 (1988), 1–10.

[Chase *et al.*, 1990] David R. Chase, Wegman, and Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices 25*, 6 (1990).

[Deutsch, 1990] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages* (San Francisco, CA, Jan. 1990), ACM SIGPLAN Notices, ACM Press, pp. 157 – 168.

[Diwan *et al.*, 1992] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation* (San Francisco, CA, June 1992), vol. 27 of *ACM SIGPLAN Notices*, ACM Press, pp. 273–282.

[Foster & Winsborough, 1991] Ian Foster and William Winsborough. Copy avoidance through compile-time analysis and local reuse. In *Proceedings of International Logic Programming Sympsium* (1991), pp. 455–469.

[Fradet, 1994] Pascal Fradet. Collecting more garbage. In [LFP, 1994], pp. 24–33.

[Goldberg, 1991] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. *ACM SIGPLAN Notices 26*, 6 (1991), 165–176.

[Goldberg & Gloger, 1992] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming* (San Francisco, CA, June 1992), ACM Press, pp. 53–65.

[Gosling, 1995] James Gosling. Java intermediate bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (IR '95)* (Jan. 1995), pp. 111–118. Published as ACM SIGPLAN Notices 30(3), March 1995.

[Gosling *et al.*, 1996] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Hamilton, 1993] G. W. Hamilton. *Compile-Time Optimisation of Store Usage in Lazy Funtional Programs.* PhD thesis, University of Stirling, 1993.

[Hamilton, 1995] G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In *Proceedings of International Workshop on Memory Management* (Dept. of Computer Science, Keele University, Sept. 1995), Henry Baker, Ed., Lecture Notes in Computer Science, Springer-Verlag.

[Hamilton & Jones, 1991] G. W. Hamilton and Simon B. Jones. Compile-time garbage collection by necessity analysis. In [Peyton Jones *et al.*, 1991], pp. 66–70.

[Hederman, 1988] Lucy Hederman. Compile-time garbage collection using reference count analysis. Master's thesis, Rice University, Aug.

1988. Also Rice University Technical Report TR88–75 but, according to Rice University's technical report list, this report is no longer available for distribution.

[Hicks, 1993] James Hicks. Experiences with compiler-directed storage reclamation. In *Record of the 1993 Conference on Functional Programming and Computer Architecture* (Motorola Cambridge Research Center, June 1993), R. John M. Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Hudak, 1986] Paul R. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming* (Cambridge, MA, Aug. 1986), ACM SIGPLAN Notices, ACM Press, pp. 351–363.

[Hudak, 1987] Paul R. Hudak. A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*, Samson Abramsky and Chris Hankin, Eds. Ellis Horward, 1987, pp. 45–62.

[Hughes, 1992] Simon Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation 2*, 4 (Aug. 1992), 483–509. Special Issue on Abstract Interpretation.

[Inoue *et al.*, 1988] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems 10*, 4 (Oct. 1988), 555–578.

[Jensen & Mogensen, 1990] Thomas P. Jensen and Torben Mogensen. A backwards analysis for compile-time garbage collection. In *ESOP'90 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)* (1990), Neil D. Jones, Ed., Springer-Verlag, pp. 227–239.

[Jones, 1995] Simon B. Jones. An experiment in compile time garbage collection. Tech. Rep. 84, Programming Methodology Group, Göteborg University and Chalmers University of Technology, Jan. 1995.

[Jones & le Métayer, 1989] Simon B. Jones and D. le Métayer. Compile-time garbage collection by sharing analysis. In *Record of the 1989 Conference on Functional Programming and Computer Architecture* (Imperial College, London, Aug. 1989), ACM Press, pp. 54–74.

[Jones & Tyas, 1993] Simon B. Jones and Andrew S. Tyas. The implementer's dilemma: A mathematical model of compile-time garbage collection. In *Sixth Annual Glasgow Workshop on Functional Programming* (1993), Workshops in Computer Science, Springer-Verlag, pp. 139–144.

[Jones & White, 1991] Simon B. Jones and M. White. Is compile time garbage collection worth the effort. In [Peyton Jones *et al.*, 1991], pp. 172–176.

[LFP, 1994] *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming* (June 1994), ACM Press.

[Lindholm & Yellin, 1996] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Mohnen, 1995] Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. Tech. Rep. 95–08, University of Aachen, May 1995. Also in Seventh International Symposium on Programming Languages, Implementations, Logics and Programs, PLILP95.

[Mulkers, 1993] Anne Mulkers. *Live Data Structures in Logic Programs*. No. 675 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[Mulkers *et al.*, 1994] Anne Mulkers, William Winsborough, and Maurice Bruynooghe. Live-structure analysis for Prolog. *ACM Transactions on Programming Languages and Systems 16*, 2 (Mar. 1994).

[Peck, 1971] J. E. L. Peck, Ed. *Algol–68 implementation*. North-Holland, Amsterdam, 1971.

[Peyton Jones *et al.*, 1991] Simon L. Peyton Jones, G. Hutton, and C. K. Hols, Eds. *Third Annual Glasgow Workshop on Functional Programming* (1991), Springer-Verlag.

[PLDI, 1996] *Proceedings of SIGPLAN'96 Conference on Programming Languages Design and Implementation* (1996), ACM SIGPLAN Notices, ACM Press.

[Shao & Appel, 1994] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In [LFP, 1994], pp. 150–161.

[Tarditi *et al.*, 1996] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In [PLDI, 1996].

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

[Thomas, 1993] Stephen P. Thomas. *The Pragmatics of Closure Reduction*. PhD thesis, The Computing Laboratory, University of Kent at Canterbury, Oct. 1993.

[Thomas & Jones, 1994] Stephen P. Thomas and Richard E. Jones. Garbage collection for shared environment closure reducers. Tech. rep., University of Kent and University of Nottingham, Dec. 1994.

[Tolmach, 1994] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation* (Orlando, Florida, June 1994), vol. 29 of *ACM SIGPLAN Notices*, ACM Press, pp. 1–11. Also Lisp Pointers VIII 3, July–September 1994.

[Wadler, 1984] Philip L. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), Guy L. Steele, Ed., ACM Press, pp. 45–52.

[Wentworth, 1990] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software Practice and Experience 20*, 7 (1990), 719–727.

[Wodon, 1971] P. L. Wodon. Methods of garbage collection for Algol–68. In [Peck, 1971], pp. 245–262.