13. A Proposal for Garbage-Collector-Safe C Compilation*

Hans-J. Boehm and David Chase

Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
(Boehm)

Sun Microsystems, Inc., MS 12-40
2550 Garcia Avenue
Mountain View, CA 94043-1100
(Chase)

Abstract

Conservative garbage collectors are commonly used in combination with conventional C programs. Empirically, this usually works well. However, there are no guarantees that this is safe in the presence of "improved" compiler optimization. We propose that C compilers provide a facility to suppress optimizations that are unsafe in the presence of conservative garbage collection. Such a facility can be added to an existing compiler at very minimal cost, provided the additional analysis is done in a machine-independent source-to-source prepass. Such a prepass may also check the source code for garbage-collector-safety.

Garbage Collection and C

C programs normally allocate dynamic memory using `malloc`, and explicitly deallocate memory by calling `free` when it is no longer needed. This approach is simple to describe and relatively simple to implement. Both `malloc` and `free` can be implemented reasonably efficiently with fairly predictable execution time. However, the need for explicit deallocation often becomes cumbersome for programs that manipulate complicated linked structures. Worse yet, explicit deallocation can substantially complicate interfaces between program units, and is likely to result in less general and thus less reusable code.

To illustrate these points, consider the implementation of a general "stack of `void *`" data type. Assume we implement stacks as linked lists. Each node contains `data` and `next` fields. What does the "pop" operation do? Presumably it returns a pointer to all but the first element of the list. If we assume the operation is allowed to destroy the old stack, it will deallocate the first linked list element `first`. Should it deallocate `first -> data`? If the stack contains pointers to data not otherwise referenced, the answer is "yes." If it points to entries in a statically allocated array, the answer is "no." If it points to separately allocated objects, but the same object may be pushed on more than one stack, the answer is "maybe." Thus even such a simple abstraction cannot be implemented with a clean interface; we must establish some convention for deallocation which will either complicate the interface to the package, reduce its applicability, or force the client to copy objects unnecessarily.

The need for explicit deallocation makes it much more difficult to use data type implementations that scale well across input sizes, such as linked representations of strings, or large integers represented by pointers to blocks of digits. A consequence of this is that C programmers are encouraged to use data representations that involve few pointers. This results in either excessive copying of data (e.g., strings that are copied or `realloc`ed whenever they outgrow a contiguous region of storage) or arbitrary size limitations (e.g., fixed size string buffers).

*Reprinted from *The Journal of C Language Translation*, Volume 4, Number 2, December 1992, pages 126–141.

These problems are aggravated by the fact that explicit deallocation of heap-allocated memory is a major source of time-consuming errors. Premature deallocation bugs are extremely difficult to trace, in that their symptoms usually do not appear until the cause of the problem is no longer visible. An arbitrary program module A can cause arbitrary failures in some other module B by deallocating memory and then writing into it after it has been reallocated by B. This makes it difficult to debug such problems without a person who understands the entire system. For large systems, there is usually no such person.

The situation deteriorates further in a multi-threaded environment, both because it becomes even less clear who will have the last reference to an object, and because the resulting problems are likely not to be repeatable, thus making them even harder to trace.

Explicit deallocation is completely unsatisfactory if C is used as an intermediate language by compilers for languages that require garbage collection (e.g., Common Lisp [16]), Scheme [4, 14], ML (cf. [8]), Cedar/Mesa [2], or Sather [13]).

A garbage collecting storage allocator solves these problems. The client program still allocates memory using a routine equivalent to `malloc`. In this paper, we will still call it `malloc`. However, it is no longer necessary to call `free`. The garbage collector (GC) will run periodically, either as part of a `malloc` invocation, or asynchronously in a separate thread, and reclaim memory that it can determine will no longer be accessed by the client code, because it can no longer be reached by following a chain of pointers. Conceptually, the garbage collector then invokes `free` on all such memory to again make it available for allocation.

There are a number of performance reasons not to implement a garbage collector on top of a standard `malloc/free` implementation. Because garbage collectors deallocate objects in bulk and generate information about the state of the entire heap, they can realize significant performance advantages over object-at-a-time deallocation. Garbage collection is usually competitive with explicit deallocation in overall execution time. Some empirical comparisons of the performance of garbage collection and explicit deallocation are given in [17].[1] But for the purposes of this paper, the reader may assume that the collector is simply a layer on top of a `malloc/free` implementation that automatically performs `free` calls.

Traditional garbage collectors require data structure layout information, so that accessible data structures can be traversed and identified. Conservative garbage collectors avoid this requirement by treating all bit patterns that could conceivably represent pointers as pointers [6].

On a typical 32 bit RISC workstation, a very naive (and very inefficient) collector might work as follows. When `malloc` allocates an object, it adds it to a hash table of allocated objects. When it discovers there is no space available, it sets a mark bit associated with each object that may still be in use. It does this by looking at each of the processor registers, each word in the stack, and each word in the statically allocated data areas. Whenever one of these contains the address of an unmarked object (to be defined more carefully later), the mark bit of that object is set, and each word in the object is recursively scanned for further pointers. Finally the hash table is traversed and unmarked objects are deallocated.

Since the collector doesn't distinguish pointers from other data, it may occasionally mark objects that are in fact unreachable. This may occasionally result in retention of more memory than with a traditional collector. Empirically, this is rarely observable in a well-designed system running in a 32-bit or larger address space [6]. Clever allocators can reduce such retention dramatically by not allocating objects at locations that are likely to be pointed to by integers and the like.

Conservative garbage collection is an appropriate tool for a large fraction of C code. It is particularly appropriate for traditional compilers and similar applications, where the main performance criterion is overall elapsed time. Garbage collection usually does not impose a significant penalty in overall execution time, and may provide a gain, especially when the accounting includes extra copying and other overhead for explicit deallocation. However it does impose two kinds of penalties. It may require more space, since a completely

[1] The reader is encouraged to make his/her own comparisons. A newer version of the collector described there is available for anonymous FTP from parcftp.xerox.com: pub/russell/gc.tar.Z

full heap would cause excessively frequent collections. And it may force the client program to periodically pause for a collection.

Pauses for garbage collection generally prevent the use of conservative collection with programs that involve hard real-time constraints. (But note that `malloc/free` is also problematic in such a context.) It is usually not a problem for interactive applications. Very straightforward collectors can traverse on the order of 3 MBytes of data per second on a SPARCstation 2. With a more sophisticated collector and minimal operating system support, latencies can be reduced to the order of 100 milliseconds on the same machine, largely independent of heap size [5]. Thus garbage collectors can meet response requirements similar to those of virtual memory systems. Still smaller latencies can be obtained (at some cost in convenience) with deferred reference count collectors [9].

Even if explicit storage management is chosen for the final version of a program, garbage collection may be used in tools such as Purify [12] to identify storage leaks [6]. The concerns below also apply, though to a lesser extent, to such systems.

GC-Safe Compilation

Empirically, there is rarely a problem with using conventional C compilers in combination with a conservative collector. Well over a million lines of Cedar/Mesa code have been compiled to C and run without difficulty in this way.

However, the ANSI C standard [1] does not preclude optimizations that are unsafe in this context. Indeed, current C compiler optimizations are often unsafe in the presence of garbage collection, though only under unlikely circumstances.

To illustrate the problem, consider the loop in a code fragment as compiled to a SPARC processor:

```
int f()
{
    int *a, *b;
    int i, sum;
    a = (int *) malloc(100000 * sizeof (int));
    b = (int *) malloc(100000 * sizeof (int));
    ...
    for (i = 0; i < 100000; i++) {
        sum += a[i] + b[i];
    }
    return(sum);
}
```

which could profitably be compiled to something like:

```
diff = b-a;
/* diff reuses b's register, which is now dead. */
for(aptr = a; aptr < a + 100000; a++) {
    sum += *aptr + *(aptr+diff);
}
```

(This is profitable on a SPARC because the `aptr+diff` addition is free, since it becomes part of a doubly indexed load instruction. This stunt saves an increment instruction in the loop, and no shift operations are necessary.) The result is that `b` does not appear accessible to a garbage collection occurring inside the loop. (In a single threaded system, we would need a function call inside the loop to trigger the collection.)

Current SPARC compilers do not appear to produce such code for this example. However, simpler versions of the problem do appear in other contexts and on other machines. Consider the function `f` in this fragment:

```
struct s {
    char space[35000];
    struct s * next;
};

struct s * f(x)
struct s * x;
{
    return(x -> next);
}
```

On an IBM RISC System/6000, this results in the following code (extracted from a compiler produced listing):

```
AIU     r3=r3,1
L       r3=SHADOW$(r3,-30536)
BA      lr
```

The first instruction adds 1 to the upper half of the argument. If a garbage collection is triggered at this point, e.g., by a concurrent thread, the only reference to `x` consists of `x + 65536` stored in `r3`. The following load instruction then supplies an offset of `-30536` to compensate for the initial overshoot.

Optimizations on array index expressions can also result in a situation in which the only reference to a heap allocated array maintained during the array access is an address well outside the object. Typically this does not cause problems, since there are other references to an array. But there is no guarantee that this will be the case.

Given current compilers, the only ways to avoid such problems are to disable optimization, to force pointers such as `a`, `b`, or `x` to memory by declaring them as volatile, or by explicitly storing them into a global data structure, thus effectively registering them as roots for the garbage collector. The first two alternatives can be extraordinarily expensive, especially on modern architectures. The third alternative requires a substantial preprocessor of some sort. It is less costly than the previous alternatives in this case. But it appears difficult to both automate it and to keep the costs down in all cases. Explicit registration of every local pointer variable is too expensive for many applications [11]. Anything else may require nontrivial analysis of the entire program.

This proposal is intended to optionally limit C compiler optimizations to guarantee safety for collectors such as those described in [6], [3], and [5]. The mostly copying collector of [3] further requires either a preprocessor or programmer discipline to identify pointers in heap objects. This proposal should also be sufficient to allow a safe, preprocessor-based implementation of delayed reference counting (cf. [15, 9]). We do not address the more difficult problem of supporting collectors that can move all objects in order to compact the heap. For a discussion of how to accomplish this for a language that provides more type information than C, see [10].

The discussion here assumes C source code. Nearly all of it is equally applicable to C++.

Base Pointers and Derived Pointers

We define any pointer value directly recognizable by the garbage collector to be a *base pointer*. The value returned by an allocation function (e.g., `malloc`) is a base pointer. A base pointer is not necessarily the base address of an object, but the garbage collector must be able to easily convert it to one. (For example, many standard `malloc` implementations store bookkeeping information at the beginning of an object, and return a displaced pointer. A Scheme implementation might add one to a pointer to distinguish it from a 30- or 31-bit integer.)

The rest of this proposal is independent of the precise definition of a base pointer, since that depends on the particular style of garbage collector. As illustrations, we will refer to two possibilities. First, we identify a *restrictive* base pointer definition, in which only pointer values returned by `malloc` (or `realloc`) are considered to be valid base pointers. This has the advantage that there is a low probability of accidentally misidentifying non-pointer data as pointers, and thus unnecessarily retaining memory. Second, we consider a *liberal* base pointer definition, in which a pointer to any position inside an object, or to one past the end of the object, is considered a valid base pointer. This requires somewhat more sophisticated support by the collector and allocator to be practical, and may require more memory. But it has the advantage that otherwise arbitrary C programs that strictly conform to [1] can be used with a garbage collector.

Intermediate base pointer definitions are common and quite useful. It is also common to treat pointers originating in the stack or registers more liberally than the notion of base pointer used elsewhere. This can reduce the chances of a pointer being hidden by a compiler optimization, but it can't eliminate it. Neither of these variations has much effect on this discussion.

We propose that the C compiler remain ignorant of the base pointer definition. It will however be useful to introduce

`# pragma base_pointer` (*list of identifiers*)

to indicate that either the given variables contain base pointers at this point, or that the given functions return base pointer values. This makes it possible to inform the compiler that functions like `malloc` return base pointers with just a header file declaration.

We may perform arithmetic or logical operations on a base pointer to derive another value that may still be used to dereference the original object. The resulting value may still be a base pointer (the usual case under the liberal definition) or it may no longer be recognizable by the collector, and thus become a derived pointer (the usual case under the restrictive definition). Any value computed from a derived pointer and that may still be used to access the object is also a derived pointer. (This is true even if its value becomes identical to a base pointer to the object. This aspect of the definition is unimportant to the client, but simplifies the task of GC-proofing existing compilers.)

GC-Safe C Programs

At any particular point of execution in a C function `f`, we define the *local root set* to be the set of

1. In-scope `auto` and `register` variables declared in `f` and visible at the point of execution.
2. All previously computed values of direct subexpressions of incompletely evaluated expressions. For example, if we are about to perform the addition in `(char *) malloc(N) + 4`, then the value computed by `(char *) malloc(N)` is a local root.

At a particular point of execution in a C program, we define the *global root set* or just *root set* to consist of:

1. All values of `static` and `extern` variables declared in the program.
2. All values in the local root set at the current execution point.

3. All values in the local root set at any of the call sites in the call chain.
4. Values stored in other areas of memory not under the control of the collector, but scanned by the collector. Memory allocated by `sbrk` might be a candidate.[2] The presence of such other roots doesn't affect the rest of this presentation, and will be ignored for the rest of the discussion.

A C program may safely use a conservative garbage collector if it satisfies the following criteria.

(I) Every object allocated through a garbage collecting allocator that may still be accessed is accessible by following chains of base pointers originating with a value in the root set. For purposes of the analysis described below, we assume that all such base pointers are stored in variables or fields of composite data structures declared to hold either C language pointers, or a sufficiently large integral type to hold a pointer.

(II) (This is a technical restriction for concurrent or generational collectors.) Every statement that causes a variable to hold a valid reference to an object involves an assignment to that variable. This is impossible to violate in a program that strictly conforms to [1]. There are no known cases in which it would otherwise be beneficial to violate it. The C block:

```
{       char *x;
        {       char *y = malloc(5);
                if (x != y) x = y;
                y = 0;
        }
        ...
}
```

may violate it, but the comparison may fail in strict ANSI C. If `malloc` happened to return the value of the uninitialized `x`, then the only reference to the newly allocated storage would be through `x`, which had never been written.

Clearly only condition (I) is of real interest. It is automatically satisfied by the liberal base pointer definition, if we restrict ourselves to programs that strictly conform to [1].[3] Recall that in this case, all interior pointers are considered to be base pointers, unless they were derived from exterior pointers.

In practice, more restrictive base pointer definitions are often preferable, even though they impose more complicated restrictions on the source code. The liberal base pointer definition can make it difficult to allocate very large objects without making them likely to be accidentally retained. Nearly all C programs can operate with a more restrictive definition. In most respects, strict conformance to ANSI C is much more than we require, and it is unclear how many strictly conforming programs there are. Hence we don't constrain ourselves to this definition.

Proposal

We propose that C compilers support a flag `-GCSAFE` that forces garbage-collector-safe C source code to be compiled to GC-safe object code. In making this guarantee, the compiler is allowed to assume that all

[2] Typically, the collector must be explicitly notified of such memory.

[3] Based on the most natural interpretation of the Standard, we have to disallow writing the last pointer to an object to a file and then reading it back in, unless `fwrite` and `fprintf` are modified to keep copies of all such pointers. Recall that ANSI C generally restricts pointer arithmetic to yield a pointer inside the same object. Casts to and from integers yield implementation dependent results, and are thus not strictly conforming.

pointers are stored in variables or fields of composite data structures (including unions) declared to hold either C language pointers, or a sufficiently large integral type to hold a pointer.

To ensure GC-safety of the object code the compiler must ensure that conditions (I) and (II) above are maintained.[4] It should further attempt to meet the following somewhat less precise requirements designed to minimize the amount of work and degree of conservativism required of the collector:

(III) Any given pointer is stored in contiguous memory. On a machine with a segmented memory architecture, the segment descriptor and the offset are stored in adjacent locations. This requirement is trivially satisfiable for most architectures. It is unfortunately nontrivial to satisfy on Intel 80X86 machines with $X \leq 2$.

(IV) If feasible on the architecture, pointers should be n-byte aligned, where n is `sizeof(char *)`. Otherwise, there should be as few addresses as possible at which pointers can be stored.

Conditions (II)-(IV) are normally maintained by modern compilers for modern architectures, for reasons other than GC-safety. Condition (I) is much more easily violated. The following discussion addresses (I).

We propose that condition (I) be maintained for all possible definitions of base pointer, subject only to the constraint that `base_pointer` pragmas, if any, are correct. Assuming a liberal base pointer definition would greatly decrease the probability of introducing a problem (and thus make `-GCSAFE` harder to test), but doesn't appear to significantly simplify the compiler's task or improve the quality of its output. Automatic transformations that introduce derived pointers are usually also capable of introducing derived pointers that point outside the object.

Implementation

We assume that the source program guarantees that every accessible object is accessible via a chain of base pointers. Naive C compilers are likely to preserve this property in the object code, as will most sophisticated compilers when asked to produce fully debuggable object code. Our goal is to ensure that this property is maintained at minimal cost by an optimizing compiler.

In order to make this problem tractable, we assume that the compiler satisfies the following properties with `-GCSAFE`. In our experience, these are satisfied by most existing compilers, even in the absence of such a flag:

(A) The C compiler never generates code to recompute the value of a live base pointer from a derived pointer. In other words, all live base pointers are explicitly stored at run-time. Most compilers explicitly store all live values.[5]

(B) At every procedure call or pointer dereference, all heap locations that could possibly contain a pointer and all non-automatic variables either have their intended value, or their intended value is on the stack or in a register.

[4] At the object code level, the root set should be defined as the values scanned by the garbage collector to begin its traversal. This normally includes all values stored in the stack, the registers, and various statically allocated data segments.

[5] There are sometimes reasons not to do so [7]. An optimizer could violate this property as follows. Assume a restrictive base pointer definition. Consider a loop that sums the entries of the heap allocated array `a` in order, and that `a` is live at the end of the loop. On many machines, the compiler is likely to perform induction variable elimination, and instead step a pointer through the array. If there is a lot of register pressure, and the array has known size, it may fail to store the pointer to the base of the array, and instead recompute it by subtracting the size of the array from the final value of the pointer that was stepped through the array.

It follows from property (A) that objects that are accessible via base pointers visible to the collector have always been accessible via base pointers. (Assume that x was not accessible via base pointers, and then becomes accessible via base pointers. This is impossible at the source level, since pointers computed from derived pointers are never considered to be base pointers. Property (A) ensures that the compiler does not introduce a violation of this rule.)

Hence property (B) could fail to hold only if the location or variable were dead. It is extremely rare that a C compiler can safely determine this for anything other than an automatic variable.

Note that the lifetime of a derived pointer ends at an instruction that dereferences it. The pointer may be compared or otherwise manipulated afterwards, but for purposes of GC-safety there is no longer a need to consider it a pointer. Thus, under assumption (A), the real requirement for the compiler is that whenever a derived pointer to x is dereferenced, x must be accessible via base pointers. This can be ensured if we guarantee that:

(I') At every dereference of a possibly derived pointer x, a base pointer (possibly indirectly) pointing to the same object as x is live and visible to the collector.

Requirement (I') can be satisfied by ensuring the following:

(Ia) All local roots have their value visible to the collector (i.e., are stored in a register or on the stack) at every procedure call. (Variables that are known to point to an otherwise accessible object do not have to be made available. It is not necessary to meet this requirement if the called function is known not to dereference derived pointers.)

(Ib) At every dereference of x, either a known base pointer for x is visible to the collector, or all local roots are visible. (If x is known to be derived from either y or z, then it suffices to keep both y and z live at the dereference.)

Both (Ia) and (Ib) require that a few otherwise dead variables may have to be either maintained in registers or spilled onto the stack. We expect that they will typically require that very few variables be kept live. User introduced variables that are dead at call-sites or dereference sites are probably not all that frequent to start with. Many of them can be determined not to contain heap pointers. Others can be determined to point to the same object as a parameter, in which case the caller will already have satisfied the requirement. Stores introduced by (Ia) and (Ib) can usually be moved out of loops, further reducing the cost.

The added cost is likely to be essentially zero on register window architectures with large windows (notably SPARC) where there is typically an ample supply of registers for pointer-intensive code, and there is no incremental cost for saving these registers across procedure calls. It is likely to be more noticeable, but still minor, for architectures with fewer registers, such as the Intel 80X86 processors.

Source-Level Implementation

Under most circumstances, an existing compiler will already satisfy all requirements except (I'). We further assume that an existing compiler will not generate code that introduces pointer dereferences that do not correspond to any source expression. Ideally we would like to ensure that all compiler back-ends be modified to guarantee (I'). This is probably not practical in all cases. Here we present a strategy for implementing this proposal that requires at most minimal modifications to compiler back-ends, at the expense of requiring an additional source-to-source translation pass. This clearly involves a cost in compilation speed when GC-safety is needed. On the other hand, nearly all of the work needs to be done only once.

A more direct implementation in the C compiler itself could use a very similar approach. A compiler that generates C as its intermediate language could incorporate a greatly simplified version of the prepass.

Instead of guaranteeing (I'), the C compiler itself must provide the following:

(C) We must be able to define a macro `KEEP_LIVE`(e, y, t, tmp) whose value is the value of e, but that guarantees that the pointer or integer variable y is accessible to the collector until any dereferences within e are completely evaluated. The parameter t will always be the type of e, and tmp will always be a local register variable of type t that is not otherwise in use during the macro invocation. We assume there is a special version of this macro `KEEP_LIVE_VOID`$(e,\ y)$ for the case in which t is `void`. Both t and tmp will be omitted in some of the following discussion. Evaluation of `KEEP_LIVE`(e, y) should not be appreciably more expensive than evaluating e.

The precise definition of `KEEP_LIVE` will probably be compiler dependent. The following definition of `KEEP_LIVE`(e, y, t, tmp) is likely to suffice for most compilers:

```
( tmp = (e), _SINK = (void *)y, tmp )
```

where `_SINK` is an `extern void * volatile` variable. This makes it extremely difficult for a compiler to permute the assignment to `_SINK` with dereferences in e and still be standard conforming. The extra assignment to tmp can easily be optimized out on a register-based machine architecture. Eliminating the assignment to `_SINK` requires either a postpass over the assembly code, or a corresponding facility built into the compiler back-end.[6] In addition to the increased register pressure mentioned above, this implementation of `KEEP_LIVE` may inhibit instruction scheduling by adding spurious dependencies between memory reference instructions. Some scheduling can be performed after removing assignments to `_SINK`, but careless scheduling is unsafe.

This definition of `KEEP_LIVE` assumes that e may terminate. To verify this, consider the expression `KEEP_LIVE(f(), y)`, where f contains dereferences inside an easily recognizable infinite loop. The compiler would be fully justified in optimizing out the assignment to `_SINK`, since it is provably never executed. In practice, this may happen if the call to `f` is in-lined. It is hard to define `KEEP_LIVE` without such an assumption. To make it possible for the prepass to ensure that all function calls are viewed by the compiler as possibly terminating, we need the following:

(D) We must be able to define a macro `DISGUISED_ZERO` that always returns the constant 0, but is treated by compiler optimizations as returning an unknown value. The one exception should be that all instructions that will never be executed as a result of `DISGUISED_ZERO`'s value, and all branches that are never taken, should be eliminated. This could be implemented by defining `DISGUISED_ZERO` to be `_zero` where `_zero` is an external volatile variable initialized to 0. A simple postpass over the compiler output could again eliminate the resulting dead code.

Our prepass then transforms the code in the following ways:

1. All immediate subexpressions e of function calls and dereferences whose result may contain an otherwise inaccessible base pointer are replaced by $(tmp = (e))$, where tmp is a newly introduced temporary register value of the right type. For our purposes, a dereference is an expression that uses `*`, `->`, or `[]` as its outer operand.

2. All functions that may directly or indirectly reference derived pointers and cannot be guaranteed to always return are augmented with the initial statement

[6] Use of a reserved identifier such as `_SINK` ensures that conflicts with other identifiers are unlikely.

`if (DISGUISED_ZERO)` { *build C* ; `return` *C* ; }

where C is some expression of the correct type. This ensures that the compiler may not eliminate code immediately following a function call.

3. Whenever requirement (Ia) implies that variables x_1, x_2, ..., x_n must be live at a call to f, we replace the call by:

   `KEEP_LIVE(...KEEP_LIVE( KEEP_LIVE(`f `(...),` x_1 `),` x_2 `),...,` x_n `)`

   If the type of e is `void`, we use `KEEP_LIVE_VOID` instead. Note that step 1 guarantees that all local roots are either in local variables or, in very unusual cases, in fields of local union or structure variables.

4. Whenever requirement (Ib) implies that variables x_1, x_2 , ..., x_n must be live at a dereference expression e, we replace e by:

   `KEEP_LIVE(...KEEP_LIVE( KEEP_LIVE(` e, x_1 `),` x_2 `),...,` x_n `)`

5. We add declarations for the local variables required by any of the above steps.

Further optimizations are possible. For example, nested `KEEP_LIVE`s for the same variable can usually be coalesced.

Our first example might be annotated as:

```
int f()
{
    int *a, *b;
    int i, sum;
    register int * tmp1;
    register int tmp2;

    if (DISGUISED_ZERO) return(0);
    a = (int *)malloc(100000 * sizeof (int));
    b = KEEP_LIVE( (int *)malloc(100000 * sizeof (int)),
        a, int *, tmp1);
    ...
    for (i = 0; i < 100000; i++) {
        sum += KEEP_LIVE(a[i], a, int, tmp2) +
               KEEP_LIVE(b[i], b, int, tmp2);
    }
    return(sum);
}
```

This assumes that `malloc` appeared in a `base_pointer` pragma. It otherwise assumes that the prepass is not particularly clever. The output would be simpler if it could determine that the body of the function always terminates, or that `malloc` does not dereference derived pointers, or that the second `KEEP_LIVE` implies the first. (Note that there are possible definitions of `malloc` such that the first `KEEP_LIVE` is necessary. Also the other statements would be falsified if the code in the ellipsis provably failed to terminate.)

Although we have added a significant amount of clutter to a version of the program that the programmer will hopefully never see, none of this would normally add any runtime overhead. The conditional return could be easily eliminated. The `KEEP_LIVE` calls force the variables `a` and `b` to be kept in registers throughout the loop. Since they would need to be there at some point anyway, this adds instructions only if too few registers are available. The optimization we discussed at the beginning could still be performed, if it were found to be desirable.

Summary

Ideally C compilers would provide a `-GCSAFE` flag that ensures that only garbage collector safe transformations are performed on the code. However, this is not always practical. Here we repeat the minimum set of essential and interesting requirements on a C compiler such that GC safety can be guaranteed by an additional prepass over the source code. We consider a requirement to be interesting if we are aware of a violation by an existing compiler, or we feel it could be violated in the interest of a legitimate optimization. We would expect that most compilers could meet these requirements for all compilations, and in fact, that many already do. However, it suffices that these be met when an appropriate compilation flag is specified.

(III) Any given pointer is stored in contiguous memory, or in locations that can be recognized by the collector as related.

(A) The compiler never generates code to recompute the value of a live base pointer from a derived pointer. In other words, all live base pointers are explicitly stored at run-time.

(C) We must be able to define the macro `KEEP_LIVE(` e, y, t, tmp `)`.

(D) We must be able to define a macro `DISGUISED_ZERO`.

On standard RISC architectures with existing compilers, we expect that only (C) and (D) may be problematic. We claim that they are easy to support directly in a compiler back-end, or can be implemented with a postpass on the compiler output.

Acknowledgments

Revisions to earlier versions of this proposal were inspired by [10] and discussions with various others, including Alan Demers, John Ellis, and Thomas Breuel.

References

[1] ANSI. *Programming Language C, X3.159-1989.* American National Standards Institute, 1989.

[2] Russ Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. Experiences creating a portable Cedar. *ACM SIGPLAN Notices*, 24(7):322–329, July 1989. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[3] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. WRL Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, February 1988. Also in *Lisp Pointers* 1, 6 (April-June 1988), pp. 3–12.

[4] Joel F. Bartlett. Scheme ⇒ C: a portable Scheme-to-C compiler. WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, January 1989.

[5] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.

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

[7] P. Briggs, K. Cooper, and L. Torczon. Rematerialization. *ACM SIGPLAN Notices*, 27(7):311–321, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[8] Regis Cridlig. An optimizing ML to C compiler. In *ACM SIGPLAN Workshop on ML and Its Applications*, San Francisco, June 1992.

[9] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9), September 1976.

[10] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, number 7 in SIGPLAN Notices 27, pages 273–282, July 1992.

[11] Daniel Edelson. A mark-and-sweep collector for C++. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 51–58, 1992.

[12] Reed Hastings and Bob Joyce. Fast detection of memory leaks and access errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136, 1992.

[13] Stephen M. Omohundro. *The Sather Language*. ICSI, Berkeley, 1991.

[14] John R. Rose and Hans Muller. Integrating the Scheme and C languages. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 247–259, 1992.

[15] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, 1984.

[16] W. F. Schelter and M. Ballantyne. Kyoto Common Lisp. *AI Expert*, 3(3):75–77, 1988.

[17] Benjamin Zorn. The measured cost of conservative garbage collection. Department of Computer Science Technical Report CU-CS-573-92, University of Colorado at Boulder, 1992.

Hans-J. Boehm is a member of the research staff at the Xerox Palo Alto Research Center, in the Computer Science Laboratory. Previously he taught at the University of Washington and at Rice University. He has a Ph.D. in Computer Science from Cornell University. He can be reached as boehm@parc.xerox.com *or at +1 415 812 4435.*

David Chase is a staff engineer working on compilers at SunPro, a business of Sun Microsystems, Inc. in Mountain View. He has a Ph.D. from Rice University. He can be reached at +1 415 336 1587 or dchase@eng.sun.com.

∞