Tag-Free Garbage Collection for Strongly Typed Programming Languages

Benjamin Goldberg

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

Abstract

With the emergence of a number of strongly typed languages with very dynamic storage allocation, efficient methods of storage reclamation have become especially important. Even though no type tags are required for type checking programs written in these languages, current implementations do use tags to support run time garbage collection. This often inflicts a high time and space overhead on program execution. Since the early days of LISP (and Algol68 later on), there have been schemes for performing tag-free garbage collection. In this paper, we describe an improvement of existing methods that leads to more effective storage reclamation in the absence of tags.

Garbage collection has also traditionally been viewed as being independent of the particular program being executed. This means that results of compile-time analyses which could increase the effectiveness of garbage collection cannot be incorporated easily into the garbage collection process. This paper describes a method for performing garbage collection 1) in the absence of tagged data, and 2) using compile-time information. This method relies on compiler-generated garbage collection routines specific to the program being executed and incurs no time overhead during execution other then the cost of the garbage collection process itself.

We describe tag-free garbage collection methods for monomorphically typed and polymorphically typed languages, and suggest how they might be extended to support parallel languages.

Author's address: 251 Mercer Street, New York, NY 10012.
Email: goldberg@cs.nyu.edu
This research was funded in part by the National Science Foundation (#CCR-8909634) and by DARPA/ONR (#N00014-90-1110).

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

1. Introduction

In the past decade, a number of programming languages with strong typing but very dynamic storage allocation have emerged. These languages require, or could benefit from, run time garbage collection. Such languages include ML [MLH90], Ada [DoD83], and C++ [Stroustrup86].

Traditionally, garbage collection (and dynamic type checking) required each datum to be tagged with type information (see [Ungar86] for description of various tagging schemes). During garbage collection, the tag of each datum is examined in order to determine how the datum should be handled. Naturally, whether the datum is a number, pointer, structure, or closure will determine how the object is treated by the collector.

For strongly typed languages, no run-time tags are required for type checking, since type checking occurs at compile-time. However, current implementations of ML, such as [AM87], retain tags to support garbage collection (Ada and C++ implementations don't have garbage collection). Maintaining these tags inflicts a space and time overhead, not only during garbage collection itself, but during the whole program execution.

This paper describes an method for completely tag-free garbage collection. It is based on the following idea (which was first described in the Algol68 literature, and which we extend):

> When compiling a program, the compiler generates the code necessary to support garbage collection. This code is specific to the program and, since the compiler knows the type of each datum in the program, requires no tagging of data. For each type in the program, there is a garbage collection routine to manipulate objects of that type.

When garbage collection occurs, the variables of each active procedure must be traced (i.e. marked or copied). In strongly typed languages, the types of the variables of each procedure in the program are known, and are the same for all calls to that procedure. Thus the type information can be associated with the procedure instead of the data. The compiler generates, for each procedure, garbage collection routines that know how trace the elements of the procedure's activation record. When garbage collection occurs, the heap-allocated structures rooted in each procedure's activation record are traced by the garbage collection routine corresponding to that procedure.

While a procedure is executing, the number, types, and status (initialized or not) of the variables in its activation record might change. This occurs if local variables are declared in nested blocks within the procedure. To handle this, different garbage collection routines should be associated with different points in the procedure.

The advantages of this approach over tagged garbage collection methods are as follows:

- More efficient use of heap space: Removing the need for tags may save considerable space, even in implementations that use several bits in each word (pointer, integer, etc.) as a type tag. Without such a tag, larger integers can be represented without resorting to multi-word representations and addressable objects do not have to be word-aligned (This is the case if the lowest bits would have otherwise been used as a tag. If the high bits would be used, a tag-free implementation provides a larger address space).
- More efficient execution: During program execution in a tagged data implementation, the manipulation of type tags incurs run-time overhead. For example, in implementations in which integers contain a one or two bit tag, the tag must be stripped off before most arithmetic operations are performed and reinstated in the result. Even the tagged arithmetic instructions provided by some processors do not eliminate the run-time overhead completely.
- More accurate recognition of live data and garbage: Typically mark/sweep and copying collectors trace from all roots, including every variable in every activation record on the stack. However, various variables in an activation record may no longer be needed and should not be traced. The compiler can determine (to some degree) the point in a procedure after which a certain variable will no longer be accessed. Therefore, if garbage collection occurs after this point in the procedure, the garbage collection code that is executed (having been compiler-generated), does not trace that particular variable.

We have already seen that the run-time overhead of our method should be considerably lower than that of ordinary tagged garbage collection. There will probably be an increase in code size, but this effect might be mitigated for the following reasons:

- Programs manipulating simple types will generate simple garbage collection routines: Since the garbage collection routines are program-specific, a program that manipulates mainly simple types (integers, reals, integer lists) will have very simple and short garbage collection routines. In current garbage collection systems, the code of the garbage collector is independent of the program and must be sophisticated enough to handle all possible user-defined types. Not only might this cause garbage collectors to be rather large, but might inhibit optimizing the representation of a complex data type because its structure must be transparent to the garbage collector.
- Recognition of program points that could cause garbage collection: The compiler can determine (to some degree) which sections of a program cannot cause the initiation of a garbage collection. In particular, an analysis can be performed that determines whether or not a given procedure call could ultimately lead to garbage collection. If not, then no garbage collection code need be generated to trace the variables of the calling procedure while it is waiting for the call to complete.

In this paper we provide a detailed description of the algorithms and representations we have developed for tag free garbage collection, and describe some program analyses to make it more efficient. The source programming language that we shall use in our examples is ML. However, we shall defer any discussion of garbage collection in the presence of polymorphically typed functions to section 3.

1.1. Related Work

In the early days of LISP, tags were avoided by allocating objects of different types in different areas of the heap. Thus the type of an object could be determined from its address. Unfortunately, this required heap allocation of integers (using two words due to the indirection involved) and was difficult to extend to languages with user-defined data types.

A number of papers [BL70,Marshall70,Wodon70] were published twenty years ago on garbage collection for Algol68, some suggesting tagged data. In a paper by Branquart and Lewi [BL70] two methods were described for using compile-time type information to avoid tags. One, called the interpretive method, associated with each type an encoding, typically a parse-tree like representation called a *descriptor* or *template*, of the structure of the type. The garbage collector, as it traverses an aggregate structure, must also traverse the appropriate descriptor to determine how to handle the substructures.

The other method which (in principle) is the same as one we are espousing, is the compiled method. Like the name suggests, the garbage collection routines for each type were generated by the compiler. The fundamental difference between our method and the compiled method described in [BL70] is as follows:

- In the Branquart and Lewi method, a table mapping stack locations to garbage collection routines was kept at run-time in order to figure out what garbage collection routine to use to trace each local variable in each activation record. This table had to be updated every time a local variable bound to a heap-allocated structure was created (which was seldom, since Algol68 discouraged the use of heap allocated local variables in favor of heap-allocated global variables). No table entries were required for global variables, since both the location of, and garbage collection routine for, each global variable was known at compile time.
- In our method, no table is required. We are able to determine the garbage collection routines for each local variable by using the return address pointers that are already stored in the stack.

Garbage collection schemes for Pascal, similar to the ones for Algol68, were described in [Britton75]. Instead of using a table, however, an extra pointer was stored in every activation record to point to the descriptor or compiled routine corresponding to the types of the variables in the activation record.

Appel [Appel89] has proposed an extension (although omitting many of the details) to the interpreted methods of [BL70,Britton75] in order to support garbage collection for polymorphically typed languages. In particular, he recognized that the return address stored in each activation record can be used to find the type information (i.e. the descriptor) for the variables stored in that activation record. In the following section, we give a more extensive description of Appel's method.

The use of the return-address pointer has been used to implement exception handling in a number of languages (e.g. Mesa, Ada). As exceptions traverse the dynamic chain, the procedure represented by each activation record is examined to see if an appropriate exception handler is provided.

1.1.1 Appel's Tag-Free Collection Scheme

In [Appel89], an important passage describing his tag-free garbage collection method is the following:

> When the garbage collector is invoked, it searches the stack for references into the heap. From the return-address information on the stack, it can determine which procedure is associated with each stack frame.

Since this is the only passage describing the use of the return address in each activation record, the reader is left to interpret exactly *how* the return address is used. In this section, we provide a straightforward interpretation of the passage in which a single descriptor is associated with each procedure definition. This is implied by Appel since his method seems to rely only on finding out which procedure is associated with each activation record and does not take into account the current execution point in the procedure.

In subsequent sections we describe our algorithm, based on the compiled method, in which we are able to associate different garbage collection routines with various points in each procedure. The garbage collector then determines the current execution point of each procedure and uses that information to select the appropriate garbage collection routine. As we shall show, this allows our scheme to make use of intraprocedural and interprocedural program analyses to optimize the garbage collection process.

In (our interpretation of) Appel's method, when the garbage collector encounters an activation record R, it determines which procedure *f* is represented by R by following the return address pointer stored in R to the call instruction in the procedure that called *f*. The call instruction will contain the starting address of *f*. The type descriptor for the variables in *f* can be associated with *f*'s starting address - either through a table or by placing the descriptor (or its address) at some fixed distance from *f*.

Some problems with associating a single descriptor with each procedure definition are:

- The number, types, and status of variables in an activation record change during the execution of the corresponding procedure. If there are variable definitions in nested blocks in the procedure, then local variables appear and disappear.
- Variables may or may not be initialized at various points in a procedure. Uninitialized variables present a problem to the garbage collector (it may think that an uninitialized pointer contains a valid address).

The solution to this would be to create all local variables defined inside a procedure as soon as the procedure is called, and to immediately initialize the variables. This imposes an additional time and space overhead during execution.

In addition, associating a single descriptor with a procedure definition misses a significant opportunity for optimization. At various points in a procedure, some local variables will be live and others will not. If the garbage collector always performs the same action on the activation record for a given procedure, it will have to assume that all variables are live. This will prevent the reclamation of heap structures pointed to by local variables that are not actually live. The tag-free garbage collection algorithm that we describe in the next section addresses these issues.

2. The Tag Free Garbage Collection Algorithm

In this section we describe our method for monomorphically typed sequential languages. In subsequent sections we extend the method to work with polymorphically typed languages and languages supporting parallelism. Even though we use ML in examples in this section, we restrict ourselves to monomorphic functions.

For consistency throughout this paper, we will assume that a copying garbage collector is used. However, our method will support mark/sweep collection as well (see [Cohen81] for a survey of garbage collection schemes).

2.1. The Basic Method

An important observation is that garbage collection can only be initiated by a call to a procedure (such as **cons**, **new**, or **malloc**, depending on the language) that allocates memory. Such a procedure must be predefined in the language. Therefore, garbage collection can only occur when each active user-defined procedure is waiting for a procedure call to complete.

When garbage collection begins, the activation record of each active procedure *f* contains a valid return address. This is typically the address of the call instruction in the calling procedure (on the SPARC, for example). To return from *f*, the CPU increments the return address (again, on the SPARC) by two words (8 bytes) to the second instruction following the call (the first instruction after the call sits in the delay slot) and then jumps to that address.

We use the return address stored in each activation record to find the garbage collection routine for tracing the activation record below it (i.e. of the caller). The details are as follows:

- Suppose, for a given procedure call, the address of the call instruction is *n*. We use the word at location *n+8* (which is two 32-bit words after the call instruction) to store the address of the garbage collection routine, which we refer to as a *frame_gc_routine*, that will specify how the elements of the activation record (frame) of the calling procedure should be traced. We shall refer to the word at location *n*+8 storing the address of the frame_gc_routine as the *gc_word*.
- During garbage collection, the return address stored in the called procedure contains the value *n*. To trace the calling procedure's activation record correctly, we simply call the frame_gc_routine whose address is contained in the gc_word at location *n+8*.
- During normal execution, when a procedure call finishes, control returns to the instruction at address n+12, where *n* was the return address stored in the called procedure's activation record. On the SPARC, at least, this incurs no additional cost because the **retl** instruction is just a pseudo-instruction representing **jmpl %o7+8,%g0** and can simply be replaced by **jmpl %o7+12,%g0**.

This representation is illustrated in Figure 1. When garbage collection occurs, the collector simply traverses the stack (from most recent activation record to least recent), executing the frame_gc_routine associated with each activation record. Figure 2 shows the main loop of the garbage collector, written in C (in actual implementations it would be written in assembly code).

The frame_gc_routines associated with each procedure take a pointer to an activation record as a parameter and trace the variables in the activation record according to their types.

2.2. Higher Order Functions and Closures

Many languages support higher-order functions, which means that closures must be constructed to represent func-

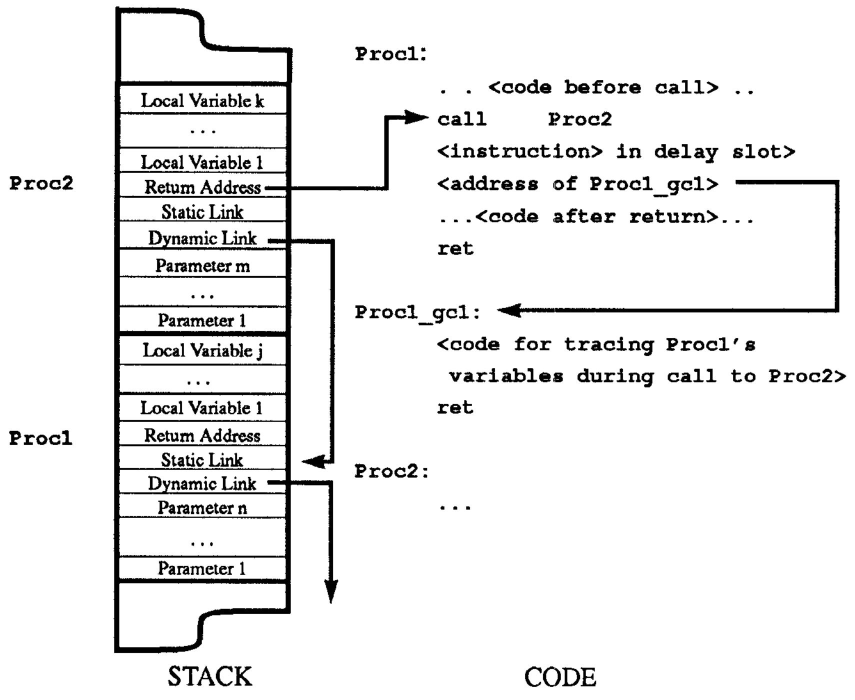


Figure 1. Stack/Code Organization

tions as data. Consider the (monomorphic) ML function **map**:

```
fun map f ([]: int list) = [] : int list
  | map f (x::xs) = f x :: map f xs
```

When **map** is called, the first argument must be represented by a closure. However, the size (number of fields containing the free variables) and shape (types of those variables) of the closure may differ in different calls to **map**. There is no way for **map**'s frame_gc_routines to know how to trace the fields of the closure.

This problem is easily solved. A closure contains a pointer to the code of the function it represents. Suppose that the code starts at address *n*. In location *n-4* (the word preceding the start of the code) the compiler places the address of a garbage collection routine for tracing the closure. This works because all closures representing the same function have the same number and types of fields. When a closure is encountered during garbage collection, the closure's code pointer is followed to find its garbage collection routine.

```
garbage_collect()
{ frame *current_frame;
  void (*gc_routine)();
  current_frame = frame_pointer;                     /* start at the top of the stack */
  gc_routine = *(current_frame->return_addr + 8);     /* get the frame_gc_routine for the next frame*/
  while (current_frame != NULL) {
        current_frame = current_frame->dynamic_link;   /* visit next frame in the stack */
        (*frame_gc_routine)(current_frame);            /* call the frame_gc_routine */
        gc_routine = *(current_frame->return_addr + 8); /* get the next frame _gc_routine*/
  }
}
```

Figure 2. The garbage collector procedure

2.3. Variant Records

Languages like Pascal and Ada support *variant records*, which are records whose number and types of fields are determined by the value of a *discriminant* field at run time. Thus, when garbage collection occurs, the value of the discriminant must be checked in order to determine the types of the current fields. This is easily supported by our tag-free collection strategy, since the frame_gc_routines associated with a procedure containing a variant record can test the record's discriminant and perform the appropriate action. ML has similar objects, whose types are defined by ML's **datatype** declarations, for which the same garbage collection strategy works (again, assuming only monotypes).

2.4. An Interesting Example

For ease of presentation, we write the implementation of the garbage collection routines in C instead of assembly code. We indicate the frame_gc_routine associated with a procedure call as follows:

```
f(x,y,z);          -- this_gc
```

namely, by writing the name of the frame_gc_routine (in this case, **this_gc**) to the right of the procedure call, preceded by "--". This indicates that if garbage collection occurs during the call to **f**, the routine **this_gc** will be called to trace the fields of the activation record of the caller.

Consider the ML append function for integer lists:

```
fun append [] (ys: int list) = ys
|   append (x::xs) ys = x::append xs ys
```

It might be implemented as follows:

```
cons_cell *append(xs,ys)
cons_cell *xs, *ys;
{ int temp;
  cons_cell *res;
  if (xs == NULL) return(ys);
  else {
        temp = xs->car;
        res = append(xs->cdr, ys); -- no_trace
        res = int_cons(temp, res);  -- no_trace
        return(res);
  }
}
```

Notice the following properties of this append:

- If garbage collection occurs during the recursive call to **append**, then the only variable whose value is required later is **temp**. Since **temp** is an integer on the stack, no action needs to be taken.
- If garbage collection occurs during the call to **int_cons**, no local variable or parameter is needed anymore, so again no action needs to be taken (**int_cons** will trace its parameters).

So, garbage collection never needs to trace the elements of an **append** activation record! The frame_gc_routine associated with both calls in the body of append simply returns. Naturally, there is only one such frame_gc_routine, in this case called **no_trace**, and many gc_words will point to it.

The gc_word following many procedure calls will contain the address of **no_trace** for one of two reasons:

- Like the append function above, no heap-allocated structures rooted in the callers activation record need to be traced, or
- The compiler has determined that no garbage collection can occur during the call. Such an analysis is described in section 5.1. Better yet, if both the calling and called procedure are aware that garbage collection cannot occur, then the gc_word following the call instruction can be omitted.

More examples of our garbage collection method are provided in subsequent sections.

It is worth noting that the method we have described can be adapted to the interpreted method. In such a case, the gc_word that currently points to frame_gc_routine would instead point to a descriptor that describes the types of variables in the activation record. Garbage collection would be somewhat slower, since the descriptor would have to be interpreted while traversing the activation record. However, the code size should be significantly less, since the descriptor should take less space then the corresponding frame_gc_routine. What the precise space/time trade-off is remains to be seen from experiments that we are planning to perform in the near future.

3. Garbage Collection for Languages with Polymorphism

In a languages with polymorphic functions, different calls to the same function may supply arguments of different types. For example, the append function in ML

```
fun append [] ys = ys
|   append (x::xs) ys = x::append xs ys
```

is polymorphic, and its type is $\forall\alpha.\ \alpha\ list \rightarrow \alpha\ list \rightarrow \alpha\ list$. This means that for any type, append can take two lists of elements of that type and return a list of elements of that type.

In current ML implementations, there is only one definition of each polymorphic function and all calls to that function execute the same code. While that is not a necessary condition for implementing polymorphically typed languages, it is a convenient way to do so. As we shall see, it is precisely this implementation decision that makes tag-free garbage collection more difficult and more interesting.

Typically, the formal parameters of a polymorphic function, especially those which get bound to arguments of different types in different function calls, are represented by a single word (integer or pointer) and polymorphic functions, such as **cons** and **append**, simply manipulate these words. **cons**, for instance, creates a cell with two one-word fields containing its arguments, no matter what the type of the arguments are.

Since all calls to a polymorphic function execute the same code, there is no way for the function's frame_gc_routines to know precisely the structure of all variables in the activation record during garbage collection. This would seem to make tag-free garbage collection impossible. Appel [Appel89] suggests the following solution:

- Suppose the garbage collector cannot determine the type of a variable in a polymorphic function's activation record. Since the types of the arguments to a polymorphic function determine the types of its parameters and local variables, the calling procedure (found by the return address) is examined to determine the type of the arguments. If the calling procedure is itself polymorphic, then its caller may have to be examined, and so on. This continues (that is, traversing down the dynamic chain) until the precise type of each variable in the current activation record can be determined.

The problem with this is that the tracing of *each* polymorphic function's activation record may involve traversing a fair amount of the stack and testing the identity of activation records many times. Furthermore, determining the type of the parameters by accessing the encoded types of the variables of the caller (and its caller, etc.) quickly becomes very complicated. No details are provided in [Appel89], and just how the garbage collector can determine the types of variables in one activation record by looking at the descriptor for another activation record is far from clear.

Our solution differs in that the stack is traversed at most twice, no testing of activation records is required, and our method is easier to describe and, hopefully, implement.

Our solution is as follows:

- The stack is traversed from the oldest activation record to the most recent (i.e. in the opposite direction of the dynamic chain). In order to do this, an initial traversal of the stack may be necessary, to perform pointer-reversal on the dynamic links.
- The frame_gc_routines associated with a polymorphic function *f* are parameterized by garbage collection routines corresponding to the types of *f*'s parameters We shall refer to these routines as *type_gc_routines* because they trace objects of a certain type rather than entire activation records. During garbage collection, *f*'s frame_gc_routine is passed the type_gc_routines corresponding to *f*'s arguments by the frame_gc_routine of the procedure that called *f*. Thus the structure of each frame_gc_routine will be:

```
f_frame_gc(p, par_1_gc,..., par_n_gc)
stack_frame *p;
type_gc_routine par_1_gc,..., par_n_gc;
{ frame_gc_routine next_gc;
  ... trace variables using par_1_gc,..., par_n_gc ...
  next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next_frame, arg_1_gc,..., arg_n_gc)
}
```

  where **arg$_1$_gc,..., arg$_n$_gc** are the type_gc_routines passed to the next frame's frame_gc_routine. They correspond to the types of the arguments passed by *f* to the function it called.
- Garbage collection starts simply by calling the frame_gc_routine of the bottom (oldest) activation record on the stack. Of course, the frame_gc_routine calls are tail recursive and can be implemented as a loop.
- Closures representing type_gc_routines may be constructed during garbage collection, and reflect the creation of structures during execution. In this way, the process of garbage collection for a polymorphic language greatly reflects the execution process.

Consider, for example, the following program fragment:

```
let  fun f x = let y = [x, x]
               in (y, [3])
               end
in   (f [true], f 7)
end
```

If garbage collection occurs during the creation of **[3]** in **f**, the frame_gc_routine for **f** must call a type_gc_routine to trace **y**. The type_gc_routine for **y** cannot know the exact structure of **y**, since it will vary across different calls to **f** and is dependent on the type of **x**. All **y**'s type_gc_routine can know is that it is a list, and thus must be parameterized by the type_gc_routine for **x**. The frame_gc_routine for **f**, which will have been passed the type_gc_routine for **x**, must pass that routine to **y**'s type_gc_routine. **y**'s type_gc_routine will then apply **x**'s type_gc_routine to each element of **y**, as well as copying the cons cells of **y**.

The function **f** and the "main" function (which computes the result expression) might be implemented as

```
f(x)
unknown x;
{ cons_cell *temp1, *y;
  tuple_cell *temp2;
  temp1 = cons(x,NIL);          --f_gc1
  y = cons(x,temp1);            --f_gc2
  temp1 = cons(3,NIL);          --f_gc3
  temp2 = tuple(y,temp);        --f_gc4
  return(temp2);
}
```

and

```
main()
{ cons_cell c;
  tuple_cell *temp1, *temp2;
  c = cons(true, NIL);             --main_gc1
  temp1 = f(c)                     --main_gc2
  temp2 = f(7);                    --main_gc3
  temp1 = tuple(temp1,temp2);      --main_gc4
  return(temp1);
}.
```

First, consider the code for **f**. If garbage collection occurs during the first call to **cons**, then **x** must be traced. If garbage collection occurs during the second **cons** call, then no variables need be traced. We have already described what must happen during the construction of [3], and no tracing is necessary if the **tuple** call causes garbage collection.

In the main routine, it is only if garbage collection occurs during the second call to **f** that a variable in main's activation record must be copied. This variable is **temp1** and is bound to an object of type *bool list list * int list.*

The frame_gc_routines for **main** are defined as follows:

```
main_gc1(p)
main_stack_frame *p;
{ frame_gc_routine next_gc;
  next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next,const_gc, const_gc);
}
```

```
main_gc2(p)
main_stack_frame(p);
{ frame_gc_routine next_gc;
  next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next, trace_list_of(const_gc));
}
```

```
main_gc3(p)
main_stack_frame *p;
{ frame_gc_routine next_gc;
  temp1_gc(p->temp1);
  next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next,const_gc, const_gc);
}
```

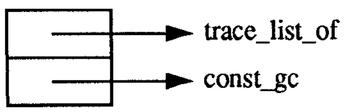
```
main_gc4(p)
main_stack_frame(p);
{ frame_gc_routine next_gc;
  next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next, temp1_gc, temp2_gc);
}
```

where **temp1_gc** is a routine to trace a structure of type *bool list list * int list* and **temp2_gc** is a routine to trace a structure of type *int list * int list.* **const_gc** is the type_gc_routine that can be used for all objects, such as integers and booleans, that are not represented by pointers into the heap, but rather by single word. **const_gc** takes a single-word value and simply returns that value. **trace_list_of** is a function that takes a type_gc_routine **r** and returns a closure representing a new type_gc_routine that can trace a list, each of whose elements can be traced by **r**. For example, Figure 3 shows the closures resulting from **trace_list_of(const_gc)** for tracing a list of integers or booleans, and **trace_list_of(trace_list_of(const_gc))** for tracing a list of lists of integers or booleans.

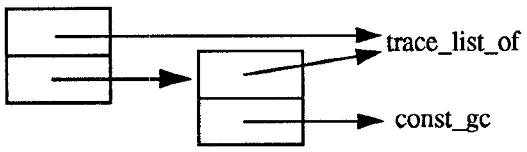
The frame gc routines for **f** are defined as follows:

```
f_gc1(p,x_gc)
f_stack_frame *p;
gc_routine x_gc;
{ next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next, x_gc, const_gc);
}
```

```
f_gc2(p,x_gc)
stack_frame *p;
gc_routine x_gc;
{ next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next, x_gc, trace_list_of(x_gc));
}
```



(a)
Closure for gc_routine for tracing a list of integers



(b)
Closure for gc_routine for tracing a list of lists of integers

Figure 3. The closure representation of gc routines

```
f_gc3(p,x_gc)
f_stack_frame *p;
gc_routine x_gc;
{ p->y = apply(trace_list_of(x_gc), p->y);
  next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next,const_gc, const_gc);
}
f_gc4(p,x_gc)
stack_frame *p;
gc_routine x_gc;
{ next_gc = ... next frame's frame_gc_routine ...
  next_gc(p->next, trace_list_of(x_gc),
       trace_list_of(const_gc));
}
```

Tag-free garbage collection gets more complicated in the presence of higher-order polymorphic functions. Consider, for example, the following function definition

```
fun f g (x::xs) =  let y = (g x)
                   in (y, 1)
                   end
```

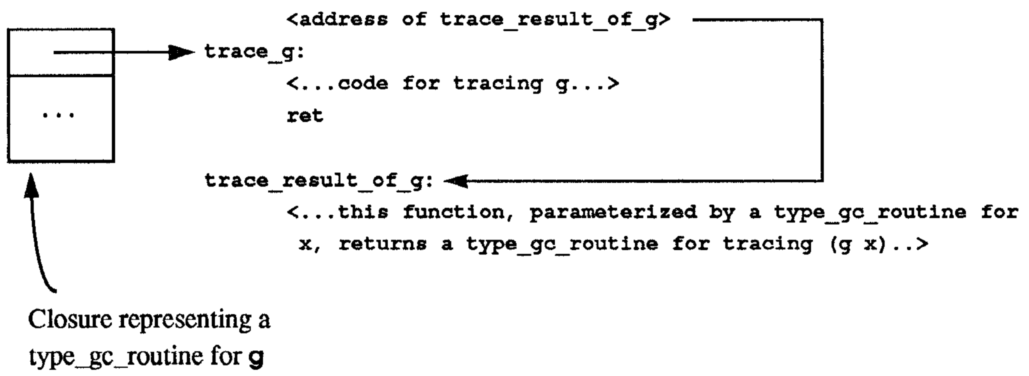
in which case **f** has type $(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta * int$. The difficulty lies in determining, while performing garbage collection on **f**, how to find the type_gc_routines for **x** and for the result of **(g x)**. This is due to **f**'s frame_gc_routine being passed a type_gc_routine for the list **(x::xs)** rather than for **x**, and that the type_gc_routine for **(g x)** is dependent on an interaction between the type_gc_routine for **g** and the type_gc_routine for **x**. The solution to these problems is as follows:

- The type_gc_routine for **(x::xs)**, passed to **f**'s frame_gc_routine, must be a closure resulting from an application of **trace_list_of** to a type_gc_routine appropriate for each elements of **(x::xs)**, and thus appropriate for **x**. Thus, the type_gc_routine for **x** can be extracted from the closure (see Figure 3).
- The type_gc_routine to trace the result of **(g x)** can be created by applying a special higher-order function, called a result_gc_routine, associated with **g** to the type_gc_routine for **x**. The result_gc_routine for **g** is accessed via the code pointer of the closure representing the type_gc_routine for **g**. Figure 4 shows the representation of the closure for the type_gc_routine for **g**, where **trace_g** is the routine for tracing **g** and **trace_result_of_g** is the result_gc_routine for **g**.

4. Tag-Free Garbage Collection for Languages with Tasking

We now describe how our tag-free garbage collection method might be extended to support languages with tasking, such as Ada. We will not attempt to describe all possible tag-free garbage collection techniques for all different classes of parallel languages. Rather, we will choose a simple model of parallelism and describe an extension of our method. The model we choose is that of Ada, namely multiple tasks operating in a shared memory environment. We further simplify the problem of garbage collection in this environment by placing the restriction that all executing tasks must be suspended during garbage collection.

Up to now, we have made the assumption that available heap space can only be exhausted during a procedure call and not at arbitrary times. In a multi-tasking language, such as Ada, this assumption cannot automatically be made. In a shared-memory environment, one process may attempt to allocate space from an exhausted heap while other processes are executing normally. If, upon the attempt to allocate space, all tasks were immediately suspended and garbage collection initiated, then the other tasks might not be in a



```
        <address of trace_result_of_g>
trace_g:
        <...code for tracing g...>
        ret

trace_result_of_g:
        <...this function, parameterized by a type_gc_routine for
         x, returns a type_gc_routine for tracing (g x)..>
```

Figure 4. The closure representation of type_gc_routines for function values

state that would allow the garbage collector to traverse their stacks.

To solve this problem, we enforce our old assumption that a process can only be suspended for garbage collection purposes when the process makes a procedure call. This implies that when the heap is exhausted, some processes will continue executing until they make a procedure call. Depending on how long one is willing to let some processes run while others are suspended, a process could suspend in one of two situations:

- The heap is exhausted and the process calls an allocation routine, or
- The heap is exhausted and the process makes *any* procedure call.

In the first case, only the allocation routines incur the overhead of checking to see if another process has exhausted the heap. This reduces the number of times a test is made to see if the process should suspend, but might allow some processes to run for a long time while others are suspended. In the second case, a test to see if the process should suspend must be made at every call.

To improve the case where a test is made at every call, it may be possible to utilize the addressing modes of some processors to make the test inexpensive (providing processes can share registers). Here is how:

- A register R_{gc}, initially containing 0, is dedicated for the purpose of testing for suspension.
- If the heap is exhausted when a storage allocation procedure is called, the procedure modifies R_{gc} to contain some (probably negative) value n.
- When a procedure call occurs, the target address of the jump instruction is computed by adding the value of R_{gc} to the address of the procedure being called. If the heap has not been exhausted by another process, the value of R_{gc} will be 0 and the call will proceed normally. Otherwise, the address will be a procedure (whose address is offset from the intended procedure by n) that causes the suspension of the process.

When all processes are suspended, garbage collection starts and the stack of each process is traversed in turn. When garbage collection is complete, the processes are resumed.

We have not yet investigated the compatibility of our tag-free scheme with multiprocessor garbage collection schemes described in the literature (a comprehensive survey can be found in [Rudalics88]).

5. Program analysis for improving tag free garbage collection

In this section, we describe some program analyses that would reduce the cost of tag free garbage, and suggest ways in which they could be performed.

5.1. Detecting when garbage collection is possible

Garbage collection can be initiated only when a heap allocation request, such as a call to the **new** operator in Pascal, is made. In a first order language it is easy to determine which calls can ultimately lead to garbage collection. The set S of functions that may ultimately lead to garbage collection can be computed by a simple fixpoint iteration,

$S^0 = \{ \textbf{new} \}$

$S^i = S^{i-1} \cup \{ f \mid f \text{ contains a call to a function in } S^{i-1} \}$

Since there are only a finite number of functions in a program, there exists some j such that $S^j = S^{j-1}$ and therefore $S^j = S$, the set we are looking for.

A similar analysis on programs with higher order functions is more difficult. One way to perform a higher order analysis is via abstract interpretation, as in [BHA85]. Another approach might be to use higher-order analysis based on type-inference methods, as in [MK89].

5.2. Live Variable Analysis

We have already discussed how live variable analysis (see [ASU86]) can be used to reduce the number of structures that are traced during garbage collection. Those variables in an activation record that are not live when garbage collection starts can be ignored by the collector.

6. Summary

In this paper, we have given a method for performing tag-free garbage collection for monomorphically typed and polymorphically typed languages, and for sequential languages as well as languages with tasking. The major contribution of this work has been to:

1. Formulate a method for garbage collection that incurs no run-time overhead in time or heap space, aside from the garbage collection process itself.
2. Show how compile-time analysis can be used to optimize the garbage collection process.
3. Provide a detailed description (missing in previous work) of the code that a compiler could generate in order to perform efficient garbage collection for polymorphically typed languages.

We have implemented, by hand, tag-free garbage collection routines for a number of small programs. In order to gain meaningful statistics on the efficacy of our approach, a real implementation is required. This entails modifying the back-end of a compiler, and we are currently in the planning stages of such a project.

7. Acknowledgments

I would like to thank the National Science Foundation and DARPA for supporting this work. I would also like to thank Robert Dewar, Fritz Henglein, Malcolm Harrison, Konstantin Läufer, Gary Lindstrom, Andrew Appel, and Dianne Britton for their assistance.

References

[AM87]
Appel, A.W. and MacQueen, D.B. A Standard ML Compiler. In *Proceedings of the Conference on Functional Programming and Computer Architecture.* Springer-Verlag LNCS 274, pp 301-324, 1987.

[Appel89]
Appel, A.W. Runtime Tags Aren't Necessary. In *Lisp and Symbolic Computation*, 2, 153-162, 1989.

[ASU86]
Aho, A., Sethi, R., and Ullman, J. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BHA85]
G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects,* LNCS 217. Springer-Verlag. 1985

[BL70]
Branquart, P. and Lewi, J. A Scheme of Storage Allocation and Garbage Collection for Algol-68. In *Algol-68 Implementation,* North-Holland Publishing Company, 1970.

[Britton75]
Britton, D.E. *Heap Storage Management for the Programming Language Pascal.* Master's Thesis, University of Arizona, 1975.

[Cohen81]
Cohen, J. Garbage Collection of Linked Data Structures. In *ACM Computing Surveys, 13(3),* 341-367. September 1981.

[DoD83]
U.S. Dept. of Defense. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A-1983.

[Marshall70]
Marshal, S. An Algol-68 Garbage Collector. In *Algol-68 Implementation*, North-Holland Publishing Company, 1970.

[MLH90]
Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press. 1990.

[MK89]
Mishra, P. and Kuo, T-M. Strictness Analysis: A New Perspective Based on Type Inference. In *Proceedings of the Conference of Functional Programming Languages and Computer Architecture*. pp. 260-271. 1989.

[Rudalics88]
Rudalics, M. *Multiprocessor List Memory Management.* Ph.D. Thesis, Johannes Kepler University, Austria. RISC-LINZ report 88-87.0, December 1988.

[Stroustrup86]
Stroustrup, B. *The C++ programming language.* Addison-Wesley, 1986.

[Ungar86]
Ungar, D.M. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press. 1986.

[Wodon70]
Wodon, P.L. Methods of Garbage Collection for Algol-68. In *Algol-68 Implementation*, North-Holland Publishing Company, 1970.