Almost Tag-Free Garbage Collection for Strongly-Typed Object-Oriented Languages

Fah-Chun Cheong
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
(313) 763-2153
fcc@eecs.umich.edu

March 25, 1992

Abstract

Given the heavy use of dynamic storage under the object-oriented paradigm, efficient storage reclamation techniques have become especially important. Previously, Appel [2] and Goldberg [12] have discussed compiler-supported tag-free collection schemes for strongly typed languages. This paper presents a more efficient, almost tag-free garbage collection scheme that takes into consideration the special requirements of object-oriented languages that are strongly typed. Our method requires only a single pass over the run-time procedure-call stack during garbage collection, as opposed to two passes for Goldberg's scheme and a linear number of passes in the worst case for Appel's.

Furthermore, in the case of distributed languages where it is practically impossible to unwind the stack all the way to its origin while trying to identify generic variables, the methods of Appel and Goldberg are not applicable. Our scheme uses a generic tag vector to help identify generic variables and does not require any stack searches beyond the current frame. We have incorporated our proposed scheme into a newly designed distributed object-oriented language Oasis. This paper includes a discussion of our practical implementation experience in the Oasis context.

1 Introduction

Languages like Smalltalk [18], Eiffel [14], CommonLisp [16], and ML [15] have traditionally required each piece of data be tagged with type information. Such tags are used for the dual purpose of dynamic type-checking and garbage collection. Such tags not only exact a toll in the form of space and time overhead during the garbage collection process itself but also during the entire program execution.

As discussed in the papers of Appel [2] and more recently Goldberg [12], run-time tags are not needed at all when garbage collecting for strongly typed languages. They have also shown how their schemes could handle polymorphism in ML [15]. Both of these methods are based upon a much earlier idea first described in the Algol-68 literature, which is simply that when compiling a program, the compiler knows about the type of each piece of data that it encounters and could thus generate the necessary code specific to objects of that type to support run-time garbage collection.

In this paper we shall not discuss tagged garbage collection, for which Cohen [8] has already provided an excellent survey. We will, however, briefly summarize the advantages of a tag-free garbage collection scheme over the tagged methods:

- A tag-free scheme allows for more efficient use of heap space. Without the tag bits, a larger range of integers or floating point numbers can be represented. Furthermore, addressable data objects do not need to be word aligned or be allocated in different areas of the heap, as would have been the case if tags were to occupy the low or high order bits of a word, respectively.

- Manipulation of tagged data, eg. removal of tag bits from integer data before arithmetic operations followed by subsequent re-attachment prior to storage, involves considerable run-time overhead.

- A tag-free scheme allows for more accurate recognition of live data and garbage as the compiler can, to a certain extent, determine whether a local variable is live or dead at a certain point in the program and can thus generate code to guide the collection process.

The earliest example of a tag-free collection scheme can be traced back to the very first implementations of Lisp, which allocate data objects in different areas of the heap based upon types. The high order bits of an address therefore implicitly encode the type information of the addressed data object. However, this simple scheme requires indirection to access integers and floating points and cannot be easily extended to handle user-defined data types. In the context of Algol-68, Branquart and Lewi [5] have discussed two tag-free schemes: an interpretive method and a compiled method. Also, a similar scheme for Pascal was described by Britton [6].

More recently, various garbage collectors have been proposed for the C++ programming language [17]. These include Bartlett's mostly copying collector [3] and Detlefs's generalization of Bartlett's work [9], Boehm's conservative collector [4], Kennedy's reference counting collector [13], Edelson's copying collector [11], and Edelson's mark-and-sweep collector [10]. The primary problem these collectors solve is the identification of pointers on the runtime stack as well as in global data space. Edelson [10] provides a fairly detailed discussion on both the advantages and disadvantages of these collectors.

Our work is more closely along the line of Appel's [2] and Goldberg's [12]. Appel's scheme [2] is an extension of previous methods and supports polymorphically typed languages like ML. He recognized that the return address stored in each activation record can lead to type information for local variables. However, Appel's scheme is mostly concerned with finding which procedure is associated with each activation record and does not take into account the current execution point in the procedure. Goldberg's scheme [12] is an improvement over Appel's in that it is more fine grained and could distinguish between different execution points within a procedure using information gleaned from a compile-time live-variable analysis. By associating different compiled garbage collection routines with various points in each procedure, Goldberg's scheme optimizes the collection process accordingly. Goldberg has also pointed out certain shortcomings in Appel's method, which assigns a fixed descriptor to each procedure definition and therefore lacks the finer granularity needed to deal with local variable initializations at various points in the procedure.

Goldberg's method is potentially faster than Appel's. Unlike Appel's interpretive scheme, Goldberg's method is compiled. Although it may require a larger code size to support garbage collection, it will actually run faster. Furthermore, in the case of polymorphic languages, Goldberg's method requires that the stack be traversed at most twice (linear time), an improvement over Appel's method which requires the number of stack traversals be proportional to the size of the stack in the worst case (quadratic time).

For the rest of this paper, we shall describe a new, almost tag-free collection scheme based upon the work of Appel and Goldberg, which we have modified for use with strongly-typed object-oriented languages. Our method, like that of Appel and Goldberg, relies on compiler generated information to guide the collection process. The difference is that ours is specifically tailored for handling generic classes and methods under the object-oriented paradigm. We recognized that generic methods within a generic class exhibit a more structured form of polymorphism compared to ML's functions. By trading off a little run-time space when dealing with generic classes, our scheme runs faster and is easier to implement than its predecessors.

2 Method

Like Appel's and Goldberg's methods, our tag-free scheme operates in the general framework of a basic two-space copying collector. We have not yet investigated the compatibility of our scheme with respect to generational or parallel garbage collection.

When garbage collection is invoked, it traces the global variables and traverses the stack area looking for references into the heap. It uses compiler generated descriptors to identify atoms, pointers and generics in the relevant areas, prior to tracing and collecting all the reacheable data. The descriptors are static information generated at compile-time, and does not involve any run-time manipulation overhead other than that of the garbage collection itself. Our scheme is tag-free in this regard. However, in the treatment of generic variables, we have resorted to using tag vectors to identify their actual types. Hence our garbage collection method is almost, although not entirely, tag-free.

From an object-oriented viewpoint, ML's notion of function polymorphism is similar to a more structured form of genericity pertaining to classes and methods (not to be confused with the similar notion of method polymorphism related to dynamic method binding). The generic variables occurring in methods are associated with type variables that are bound universally within the scope of a class template, to which these methods are attached. In other words, generic type variables are naturally associated with the enclosing class template, rather than with the individual methods themselves. This is a direct consequence of object-orientation as classes have become the focus of attention and have an existence at the global level, whereas methods have become confined to within the boundaries of a class.

A logical place to store the generic tag information would be in the object itself, i.e. as a vector embedded just below the object header but above the attributes. Our scheme uses the tag vector to determine whether a generic variable occurring in a method attached to this class is a pointer reference. It should be noted that the tag vector only appears in objects instantiated from generic classes. Ordinary, non-generic objects are fully tag-free. Looking up the tag vector is a constant-time operation and does not involve a stack search as in the case of Appel's or Goldberg's methods. This allows our method to be used in situations where it is practically impossible to unwind the stack in search of tag information for generic variables, eg. in a distributed environment. Under our almost tag-free scheme the stack is traversed exactly once during garbage collection.

3 Implementation

In this section, we shall discuss the implementation of our scheme in the context of a strongly typed, distributed object-oriented programming language Oasis [7], which has been designed and implemented by the author at the University of Michigan. Oasis is an object-agent specification and implementation system that embodies an object/agent distributed programming model. The system includes an interactive shell, a front-end compiler, a code transport and distribution mechanism, several back-end translators, and a run-time system that handles, among other things, remote procedure calls, multiple-thread scheduling, and garbage collection. We have successfully incorporated our proposed garbage collection scheme into the implementation of Oasis and will now discuss our experience.

Under the Oasis abstract machine model, the heap space is managed by two pointer registers: the heap pointer and the break pointer. The heap pointer points at the next free heap location while the break pointer points at one word beyond the end of the current heap semi-space. Upon entering a method, a comparison check is made between the heap pointer and the break pointer if the compiler detects any potential use of dynamic memory within the method. Similarly, upon exiting a method, a comparison check is made if any of the output parameters needs to be constructed from dynamic memory. Garbage collection is promptly initiated if the current semi-space is found to be exhausted, or exceeded by not more than a safe margin determinable at compile time. The compiler does not generate any comparison check code if there is no possibility of using dynamic memory within the method.

This safe margin computation is possible at compile-time for the following reasons.

- Firstly, Oasis enforces the rule that the sizes of all dynamic data objects be determinable at compile-time. For example, all arrays are created with dimensions and sizes that can be deduced at compile-time (currently through constant-folding).

- Secondly, Oasis uses recursion, with last-call optimization, as the sole means of control flow. There is no unbounded period of time where an Oasis program could run without either entering a method or exiting from one.

Thus by optionally inserting pointer check code at method entries and exits, the compiler guarantees that the heap semi-space will be checked at regular intervals, between any two of which there could be at most N allocated heap cells, where N is the much sought-after compile-time safety margin.[1] There are two reasons for our decision to associate pointer checks with method entries and exits rather than with heap allocations.

- Firstly, lumping the checks into one place at the start of each method and another at the end requires less overhead than the comparable scheme of checking at each heap allocation, of which there could be several within a method.

- Secondly, the code generated by the Oasis translator back-end is such that the top portion of the stack is always kept in machine registers for fast access. It is only during method entry and exit that the state of the stack is fully synchronized with the registers. If garbage collection is triggered as a result of heap overflow in the middle of evaluating an expression, there could be no way of knowing the precise state of the stack at that point in time.

[1] In practice, we will simply let the user configure Oasis to allow a reasonable margin. The margin computation, although theoretically feasible, is potentially expensive and non-trivial to implement.

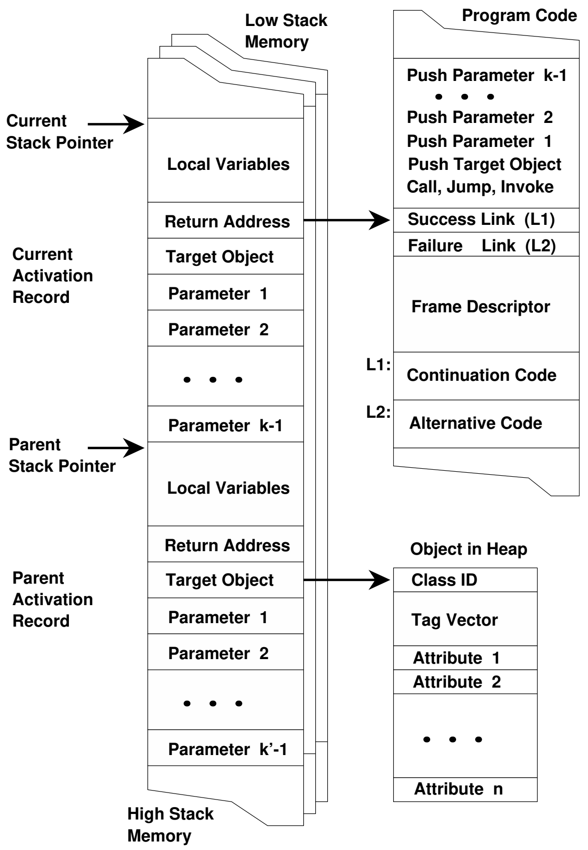

Figure 1: Organization of code, stacks, and heap that supports almost tag-free collection

3.1 Stack Traversal

In this section we shall be concerned with the organization and use of the underlying data structures needed to support stack traversal and pointer identification during garbage collection. Figure 1 is a snapshot taken at run-time that illustrates the layout of the code, heap and stacks of an Oasis agent, implemented as a Unix process, immediately after the invocation of a method. The multiple stacks reflect the multiple concurrent threads within an Oasis agent. At this time, the active thread visible in the foreground has successully pushed the call parameters onto the stack in reverse order, recorded the target object pointer, saved the return address and established an activation record for this call. The shaded regions highlight the data structures required to collect for the parent activation record, assuming that the current activation record has been taken care of in a previous collection step.

Our approach is to compile the offset information needed for tracing local variables and parameters into a frame descriptor to be placed in the code section at a fixed offset below each *call*, *jump* or *invoke* instruction. This is to ensure that the return address can be used to locate the frame descriptor during garbage collection. Figure 2 illustrates the layout of the frame descriptor and how its contents are used to access information in both the stack and the heap.

A frame descriptor is composed of two sub-tables: (1) an array of indices into the activation record to access pointer-type variables and parameters (eg. objects, lists or arrays); and (2) an array of index-pairs of the form $\langle j, g \rangle$, where j is used to index into the activation record to access generic variables and parameters, and g is used to index into the tag vector of the target object to find out the actual type of that particular generic variable or parameter. Since the sub-tables hold only non-negative indices, we can mark the end of each of the two sub-tables with a negative sentinel.

Separating the two sub-tables is an offset S, negated for use as an end-marker. It indexes into the target object slot of the activation record currently under collection. The garbage collection process needs the tag information in the tag vector located at the target object to determine the actual type of any generic variables or parameters in the activation record. Also, since the return address slot is located adjacent to the target object slot, the offset $(S-1)$ can be further used to locate the return address, and hence the frame descriptor for the parent activation record, for use in the next iteration.

We have adopted a virtual frame pointer technique for Oasis' stack discipline in order to speed up method call and return. This explains the absence of traditional dynamic links. The stack pointer is updated by an offset computed at compile-time depending upon the number of local variables and parameters in the activation record. This has less overhead than the comparable scheme of establishing a dynamic link after each call and subsequently using it to locate its parent frame before returning. However, without the dynamic links, the garbage collection process will find it impossible to traverse the stack frames in order to trace reacheable data objects. The offset information that the garbage collection process requires to skip from frame to frame has been hard-coded into the program and cannot be easily retrieved. Our solution is to add this compile-time offset $(S+K)$ to the frame descriptor so that the garbage collection process knows how to update the stack pointer to point at the parent frame after collecting for the current frame.

Observe that the frame descriptor required to collect for the parent frame is accessed through the re-

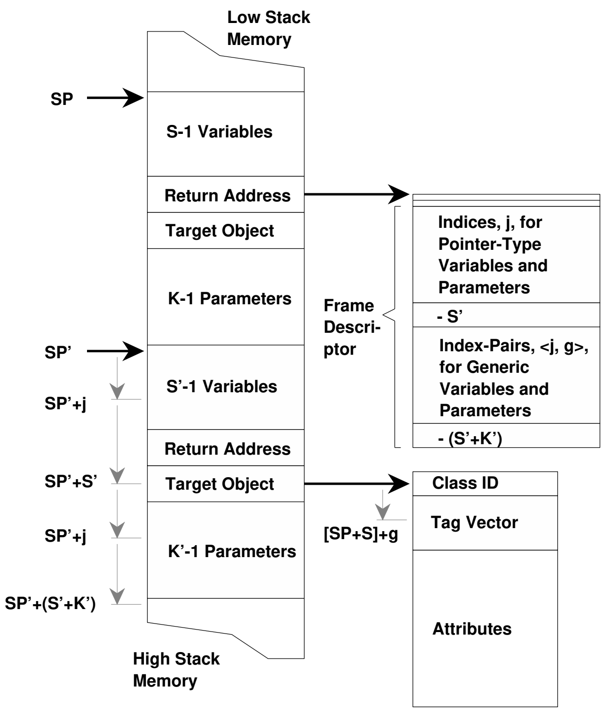

Figure 2: Identifying pointers using frame descriptor

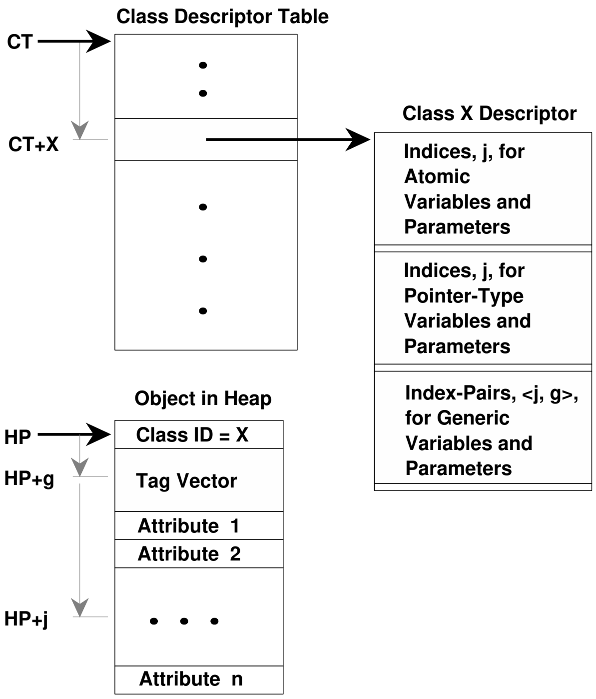

Figure 3: Collecting objects using class descriptor

turn address slot found in the child frame, as is the case for Goldberg's, and not in the parent frame itself. This setup provides us with the finer granularity required to discriminate between different execution points within the parent method and allows us to optimize the collection process accordingly using a live-variable analysis [1].

With all the supporting data structures in place, it is relatively easy to envision how the stack is traversed during garbage collection. When garbage collection is first started, it collects for the frame at the top of the stack associated with the current active thread. Next, it retrieves the frame descriptor via the return address pointer, updates the stack pointer to its parent frame, and continues the collection process on its parent using the information found in the frame descriptor. This inductive step is repeated until the base of the stack is reached. The collection process then continues on another stack associated with a different thread, and this repeats until all the stacks have been traversed. At this point, all the reacheable data will have been copied from the old semi-space into a compact new semi-space and the garbage collection terminates successfully.

3.2 Heap Traversal

We have thus far been concerned mainly with traversing the stack and identifying pointer references in the activation records. We will now discuss how these pointers to objects in the heap can be traced after they have been identified. Figure 3 illustrates the global data structures required to support heap traversal. These data structures are compiled into the program code section and require no run-time overhead for their manipulation. The class descriptor table is indexed by the class identifier that appears in the header word of an Oasis object. This leads directly to the class descriptor for objects which belong to that particular class. The layout of the class descriptor is similar to that of the frame descriptor. The major difference being the addition of a subtable containing indices into atomic attributes (eg. integers, characters and floating-points) of an object. In the case of frame descriptors, only pointers and generics are of interest since atoms stay on the stack and need not be copied. This is not true for class descriptors as heap objects have to be copied in its entirety from one semi-space to another during the collection process.

Our implementation uses breadth-dirst search to trace and copy heap objects during the collection process. The queue is implemented by using the other end of the newly allocated semi-space and thus does not take up extra space.

4 Examples

In this section we shall present two versions, one generic and one non-generic, of a `list_manager` class with the intent of illustrating the finer points of our garbage collection scheme.

4.1 Generic Class

Consider the following class definition in Oasis, which basically defines a generic class called `list_manager` parameterized by the type variable `$a`. It has an attached method `append` similarly parameterized:

```
class list_manager <$a> {
method:
  append ($a* List1, List2, # 2 inputs
             ?Result).     # 1 output
}
list_manager {
  append ([], Ys', Ys).     # base case
  append ([X'|Xs'], Ys', [X|Zs]) :-
    append (Xs, Ys, Zs').   # induction
}
```

In Oasis, a single quote after a variable name denotes its lvalue. For the above class definition, the Oasis compiler generates the following abstract machine code fragment, where the class `list_manager` has been assigned an offset of ten:

```
 1        .data              ;class descriptor
 2 C10:   .word   1          ;generic tag $<a>
 3        .word   -1         ;1st end marker
 4        .word   -1         ;2nd end marker
 5        .word   -1         ;3rd end marker
 6        .text              ;start append method
 7 L10_1:entr     3,4        ;2 inputs, 3 locals
 8        lods    1,5        ;load par #1
 9        bnez    1,L2       ;not nil => goto L3
10        lods    1,6        ;load Ys as out #1
11        retn    3,4,1      ;successful return
12 L2:    lods    1,5        ;load par #1
13        beqz    1,L1       ;nil => goto L1
14        lodm    2,1,1      ;load car of par #1
15        move    2,0        ;assign to X
16        lodm    2,1,2      ;load cdr of par #1
17        move    2,1        ;assign to Xs
18        lods    1,6        ;load Ys
19        lods    2,1        ;load Xs
20        lods    3,4        ;load target
21        call    3,1,L10_1  ;recursive call
22        .addr   L3         ;success link
23        .addr   L1         ;failure link
24        .word   -4         ;target obj offset
25        .word   0          ;X is generic var
26        .word   1          ;index to tag vector
27        .word   -7         ;parent frame offset
28 L3:    move    1,2        ;assign out #1 to Zs
29        chek    L4         ;check heap overflow
30        coll    L5         ;collect garbage
31        .data              ;frame descriptor
32 L5:    .word   2          ;Zs needs copying
33        .word   -4         ;target obj offset
34        .word   0          ;X is generic var
35        .word   1          ;index to tag vector
36        .word   -7         ;parent frame offset
37        .text              ;text section again
38 L4:    brek    1,3        ;allocate cons cell
39        lods    2,4        ;load target
40        stom    1,0,2,1    ;cons cell header
41        stos    1,1,0      ;car = X
42        stos    1,2,2      ;cdr = Zs
43        retr    3,4,1      ;successful return
44 L1:    fail    3,4        ;failed return
```

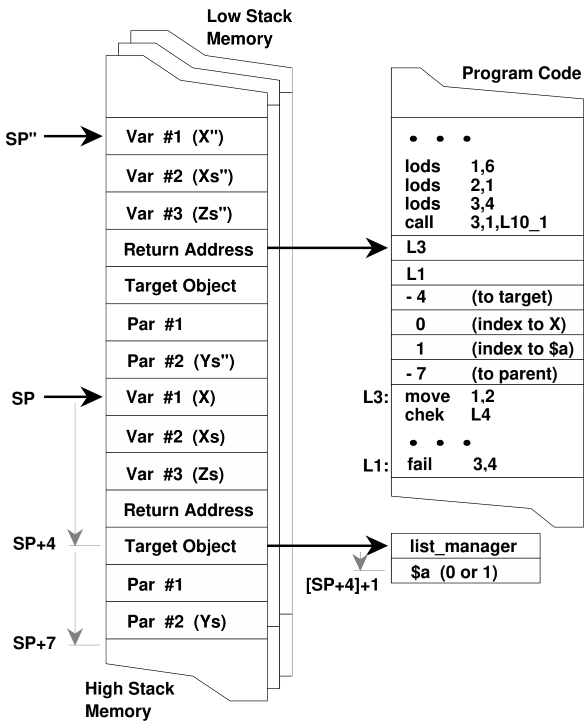

Figure 4: Garbage collecting for the generic `append` method

Graphically, this is illustrated in Figure 4. Observe that after the recursive call (at line 21) but before its return, parameter #1, `Xs` and `Ys` are already dead, while `Zs` is not yet initialized. Therefore, only `X` remains live and its generic type `$a` will determine whether it needs to be traced during collection depending on whether it is a pointer type or not.

4.2 Non-generic Class

If we use a simpler, non-generic class specification which restricts the `append` routine to handling integer lists only (but retains the basic definition of `append`), as in:

```
class list_manager {
method:
  append (int* List1, List2, ?Result).
}
list_manager {
  append ([], Ys', Ys).
  append ([X'|Xs'], Ys', [X|Zs]) :-
    append (Xs, Ys, Zs').
}
```

then the Oasis compiler generates simpler class and frame descriptors instead:

```
      .data              ;class descriptor
 C10: .word   -1         ;1st end marker
      .word   -1         ;2nd end marker
      .word   -1         ;3rd end marker
      .text              ;start append method
      ...                ;...
      call    3,1,L10_1  ;recursive call
      .addr   L3         ;success link
      .addr   L1         ;failure link
      .word   -4         ;target obj offset
      .word   -7         ;parent frame offset
 L3:  move    1,2        ;assign out #1 to Zs
      chek    L4         ;check heap overflow
      coll    L5         ;collect garbage
      .data              ;switch to data
 L5:  .word   2          ;Zs needs copying
      .word   -4         ;target obj offset
      .word   -7         ;parent frame offset
      ...                ;...
```

Observe that what used to be the tag vector (for generic variable `$a`) is no longer present in the non-generic version of `list_manager`, and thus the simpler class descriptor. The frame descriptor is also simpler but for a different reason, i.e. `X` is now an integer and does not need to be traced during garbage collection.

5 Discussion

We have initially contemplated using a compiled scheme not unlike that of Goldberg's, except for the changes made to accomodate object-orientation. But given the complexity of garbage collection, the code space overhead of a compiled scheme is expected to be quite large. A compiled method is also much more difficult to implement than our current interpretive scheme, although it could potentially be much faster. In any case, our decision to go along with the current interpretive scheme yields an important fringe benefit, i.e. class descriptors in Oasis can be reused for the purpose of marshaling and unmarshaling of data during remote procedure calls and replies.

Our scheme represents both the frame descriptor and the class descriptor as a composition of sub-tables, each of which holds index information into relevant areas inside the stack frame and object instance. An alternative design would have been to represent the descriptors as multi-bit vectors mirroring the layout of local variables and parameters in the stack frame, or attributes in the object. The multiple bits would then encode the associated tag information for the local variables, parameters or object attributes. We consider this alternative design unattractive as it introduces an unnecessary level of interpretation to the collection process, i.e. decoding of the tag information. As far as garbage collection is concerned, there are only three kinds of tags worth differentiating: atoms, pointers, and generics. Therefore, by grouping indices for accessing data of the same type into one sub-table, our scheme dispenses with run-time tag decoding altogether (except for generic tags whose values can only be known at run-time when classes become instantiated).

There is a lot of room for improvement in our current representation of the generic tag vector. We are using an array of 32-bit words to represent what is essentially a bit vector, i.e. one full 32-bit word for each single bit. At present, we are constrained by the expressiveness of the Oasis abstract machine language, which handles all data as words and has no notion of bits. In the future, as bit-shuffling instructions are added, we shall begin to represent the tag vectors as true bit vectors. However, there is a space-time trade-off here. At least on stock hardware, indexing, reading and writing a word is faster than the corresponding actions on a single bit within a word, since extra instructions are needed for shuffling or masking remaining bits in the word. As a result, generic class instantiations might just be a little slower.

6 Conclusion

In this paper we have introduced an almost tag-free garbage collection scheme based upon the methods described by Appel [2] and Goldberg [12], which we have modified for use with strongly typed object-oriented languages. Our work is directly applicable to existing object-oriented languages like C++ and Eiffel, although in this paper we have not addressed such language specific issues. Our scheme is both simpler and more efficient compared with its predecessors, requiring only a single pass over the procedure-call stack. Furthermore, our scheme does not require unwinding of the stack in order to identify the actual types of generic variables, and is thus suitable for use in a distributed setting.

This paper also discusses our experience with implementing the proposed scheme in the context of a particular distributed object-oriented language Oasis, whose design and construction has been successfully completed by the author at the University of Michigan. We have shown a detailed layout of the data structures required to support an interpretive tag-free collection scheme, which is absent from previous work. Our proposed garbage collection scheme has been incorporated into the Oasis compiler, code-generator and run-time system. In the near future, we shall be studying the comparative performances of our scheme with respect to other approaches towards garbage collection. In the long run, we are interested in finding out how our method can be modified or extended to work in a generational or concurrent manner.

7 Acknowledgements

The author would like to thank his advisor Quentin Stout, and members of his dissertation committee, Todd Knoblock, Trevor Mudge and Atul Prakash, for their comments and suggestions on early drafts of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and Tools.* Addison-Wesley, 1986.

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

[3] J. F. Bartlett. Mostly Copying Garbage Collector. Technical Report TN-12, DEC Western Research Laboratory, Palo Alto, California, 1989.

[4] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, Canada, June 1991.

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

[6] D. E. Britton. Heap Storage Management for the Programming Language Pascal. Master's thesis, The University of Arizona, 1975.

[7] F.-C. Cheong. *A High-Performance Object/Agent Oriented Programming Language and System for Heterogeneous Distributed Computing.* PhD thesis, University of Michigan, 1992. In Preparation.

[8] J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3):341–367, Sept. 1981.

[9] D. Detlefs. Concurrent Garbage Collection for C++. Technical Report 90-119, CMU Computer Science, 1990.

[10] D. Edelson. A Mark-and-Sweep Collector for C++. In *Conference Record of 19th Annual ACM Symposium on Principles of Programming Language*, pages 51–58, 1992.

[11] D. Edelson and I. Pohl. A Copying Collector for C++. In *Usenix C++ Conference Proceedings*, pages 85–102, 1991.

[12] B. Goldberg. Tag-Free Garbage Collection for Strongly Typed Programming Languages. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, Toronto, Canada, June 1991.

[13] B. Kennedy. The Features of Object-Oriented Abstract Type Hierarchy (OATH). In *Usenix C++ Conference Proceedings*, pages 41–50, 1991.

[14] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

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

[16] G. L. J. Steele. *Common Lisp: The Language.* Digital Press, Burlington, Mass, 1984.

[17] B. Stroustrup. *The C++ Reference Manual.* Addison-Wesley, Reading, Mass, 2 edition, 1991.

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