[Muller76]

Muller, K.G. On the feasibility of concurrent garbage collection. Ph.D. thesis, Tech. Hogeschool Delft, The Netherlands, March 1976.

[PJS89]

Peyton Jones, S. L and Salkid, J. The spineless tagless G-machine. *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture,* London, Sept. 1989.

[Rudalics88]

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

[Steele75]

Steele, G.S. Jr. Multiprocessing compactifying garbage collection. In *Comm. ACM*, 18,9 (Sept. 1975), 495-508.

[Wadler76]

Wadler, P.J., Analysis of an algorithm for real-time garbage collection. *Comm. ACM*, 19,9 (Sept. 1976), 491-500.

[Appel89]

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

[Baker78]

Baker, H.G. List Processing in Real Time on a Serial Computer. In *Comm. ACM*, 21,4 (April 1978), 280-294.

[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.

[Cheney70]

Cheney, C.J. A nonrecursive list compacting algorithm. *Comm. ACM*, 13,11 (Nov 1970), 677-678.

[Cohen81]

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

[DLMSS75]

Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M. On-the-fly garbage collection: An exercise in cooperation. E.W. Dijkstra note EWD496, June 1975.

[FY69]

Fenichel, R.R, and Yochelson, J.C. A LISP garbage-collector for virtual-memory computer systems. *Comm. ACM*, 12,11 (Nov. 1969), 611-612.

[Goldberg91]

Goldberg, B. Tag-free garbage collection for strongly typed programming languages. *Proceedings of the ACM SIGPLAN'91 Symposium on Programming Language Design and Implementation*, June 1991.

[Knuth73]

Knuth, D.E. *The Art of Computer Programming. Volume 2: Fundamental Algorithms*, 2nd Ed. Addison-Wesley, 1973.

[Lamport75]

Lamport, L. On-the-fly garbage collection: Once more with rigor. CA-7508-1611, Mass. Computer Associates, Wakefield, Mass., Aug. 1975.

[Minsky63]

Minsky, M.L. A LISP garbage collector algorithm using serial secondary storage. Memo 58, M.I.T. A.I Lab., M.I.T, Cambridge, Mass., Oct. 1963.

[MLH90]

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

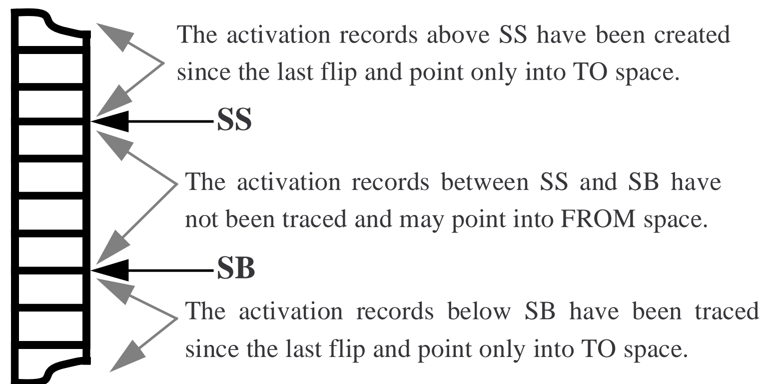

Figure 7. Stack organization for incremental collection for polymorphic languages

- In addition to SS, another pointer, SB, into the stack is used. When the spaces are flipped, SB is set to point to the bottom activation record in the stack. When garbage collection resumes, the gc_routine for the activation record pointed to by SB (not SS) is executed. When that gc_routine is finished, SB is incremented to point to the next activation record. As before, when the spaces are flipped, SS is set to point to the top activation record in the stack. When the activation record that SS points to is popped off the stack, SS is decremented to point to the activation record below.
- At any time, all activation records below SB have already been traced. The activation records above SS have been created since the last flip and can only point into TO space. Thus, the only activation records that need to be traced by the garbage collector are those between SB and SS. Figure 7 shows how the stack is logically partitioned by SS and SB.
- When garbage collection suspends, the type information that has propagated to the most recently traced activation record (i.e. the one that SB points to) is saved. When garbage collection resumes, this type information is available. There is no need to traverse the stack from the bottom each time the garbage collector resumes.
- When SB = SS, all activation records have been traced and no more copying is required until TO space is exhausted and the flip occurs.

[Goldberg91] describes in detail how type information is propagated up the stack during stop-and-copy garbage collection. The method described in that paper needs only be adapted in the ways described here in order to be used in an incremental collector.

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.

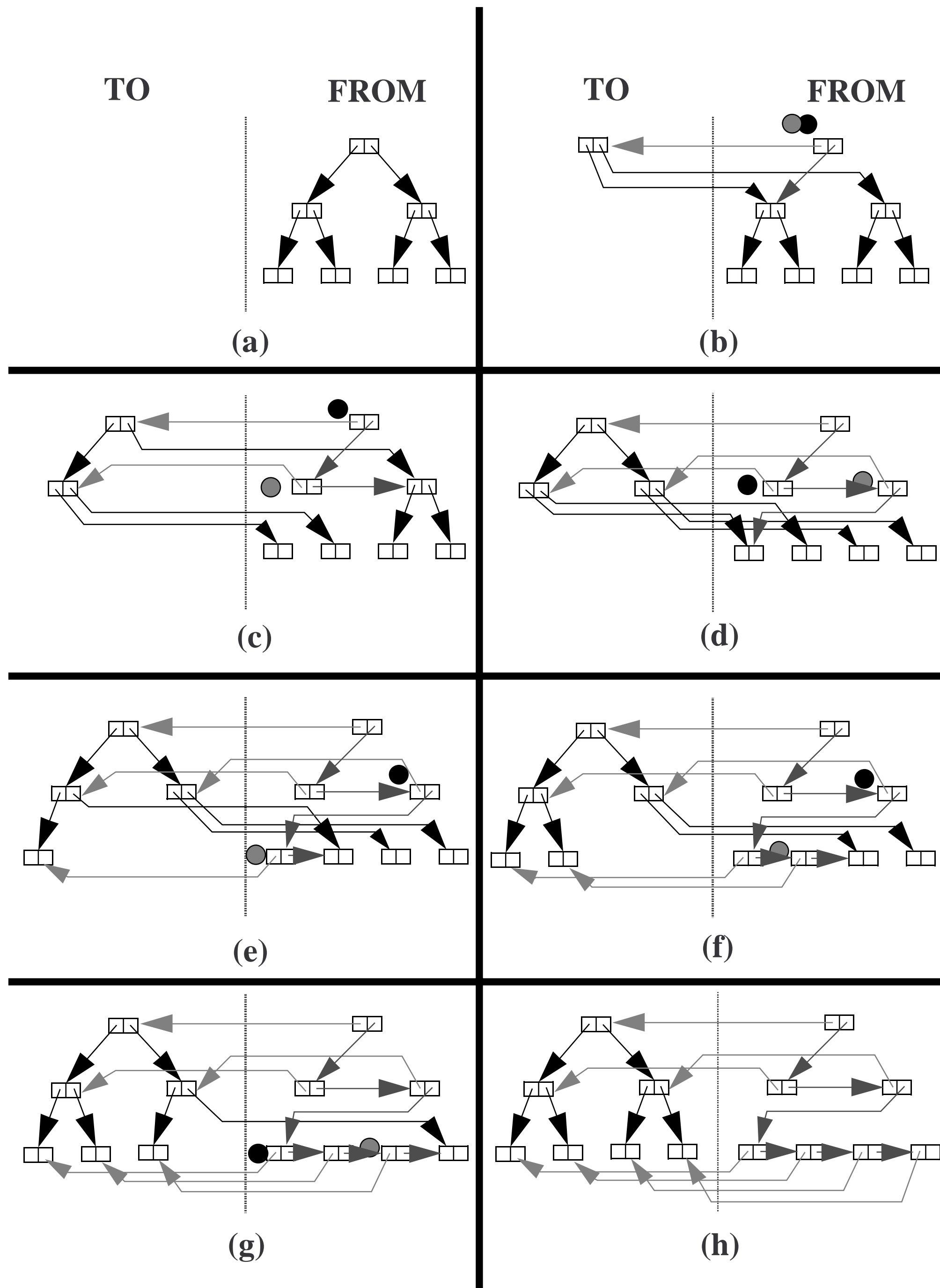

Figure 6. Breadth-first incremental copying garbage collection of trees

greycount is incremented. Whenever **black** is advanced, **blackcount** is decremented. Thus, when **grey** points to the last element of a sublist (i.e. its **cdr** is nil), and **blackcount** is zero, then all the cells of the current level have been copied. At that point, **depth** is incremented, and **greycount** and **blackcount** are reset as described above.

The breadth-first method described for lists is easily extended to trees and other user-defined recursive types. Figure 6 illustrates the method for trees.

4. Incremental Tag-Free GC for Polymorphic Languages

In a polymorphically typed language, different calls to a function may be passed arguments of different types. The same variable in different activation records of the same function may have different types. Thus, the gc_routines for the function (which all activation records for the function share) cannot know the precise types of its variables. This would seem to preclude tag-free garbage collection.

However, Appel [Appel89] suggested a solution to this problem. The types of the arguments to a polymorphic function determine the types of its parameters and local variables. Thus, if the garbage collector cannot determine the type of a variable in a polymorphic function's activation record, then the calling procedure (found by the return address and dynamic link) 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 types of the variables in the outermost function in the program, corresponding to the bottom activation record in the stack, are known. Therefore, each traversal of the stack will terminate successfully.

Since Appel's solution may require many traversals of the stack, [Goldberg91] described a method requiring a single traversal of the stack. The garbage collector starts at the bottom of the stack by calling the gc_routine of the outermost function. As each successive activation record is encountered, its gc_routine is passed encoded type information from the previous (i.e. caller's) activation record's gc_routine. This encoded type information describes the types of the arguments that were passed in the call that created the activation record. Each gc_routine reconstructs the types of the local variables from the encoded type information that was passed to it. It then passes encoded type information to the next activation record's gc_routine.

Because the incremental tag-free garbage collection method described in section 2 starts from the top of the stack after the heaps are flipped, and proceeds down the stack (as SS is decremented), there is no way for the type information to be propagated from the gc_routines of activation records further down the stack. Even if we simply reversed the order of the traversal of the stack, the practicality of the method would suffer. Every garbage collection would require the traversal of the stack from the bottom activation record to the activation record currently being collected in order to propagate the necessary type_gc_routines.

To solve this problem, we modify our method in the following way:

Figure 5. Breadth-first copying of a list with depth=3 (continued)

by following the **cdr** of the last copied cell. If the next cell is actually part of a different sublist, then it can be found using the **black** pointer (actually **car(cdr black)**- see the illustrations). Since **grey** will traverse the list in a breadth-first manner, modifying the old nested lists into one long list, the **black** pointer traverses the modified list (always behind **grey**) by following the pointers in the **car** fields of the cells.

2. The state of the partially copied list is completely captured by the **grey** and **black** pointers.
3. An extra global counter, called **depth**, is used to keep track of the depth of the cell currently being copied. This counter is incremented each time all the cells at the same level have been copied. This is easily determined: There are two other counters, called **greycount** and **blackcount** that count the number of cells traversed by **grey** and by **black**, respectively, at their current level. The number of cells copied at one level of the list is the number of parents of the sublists at the next level. Each time **grey** starts to traverse a new level, **blackcount** is set to **greycount** (i.e. the number of elements of the level above) and **greycount** is set to zero. Whenever a new cell is copied,

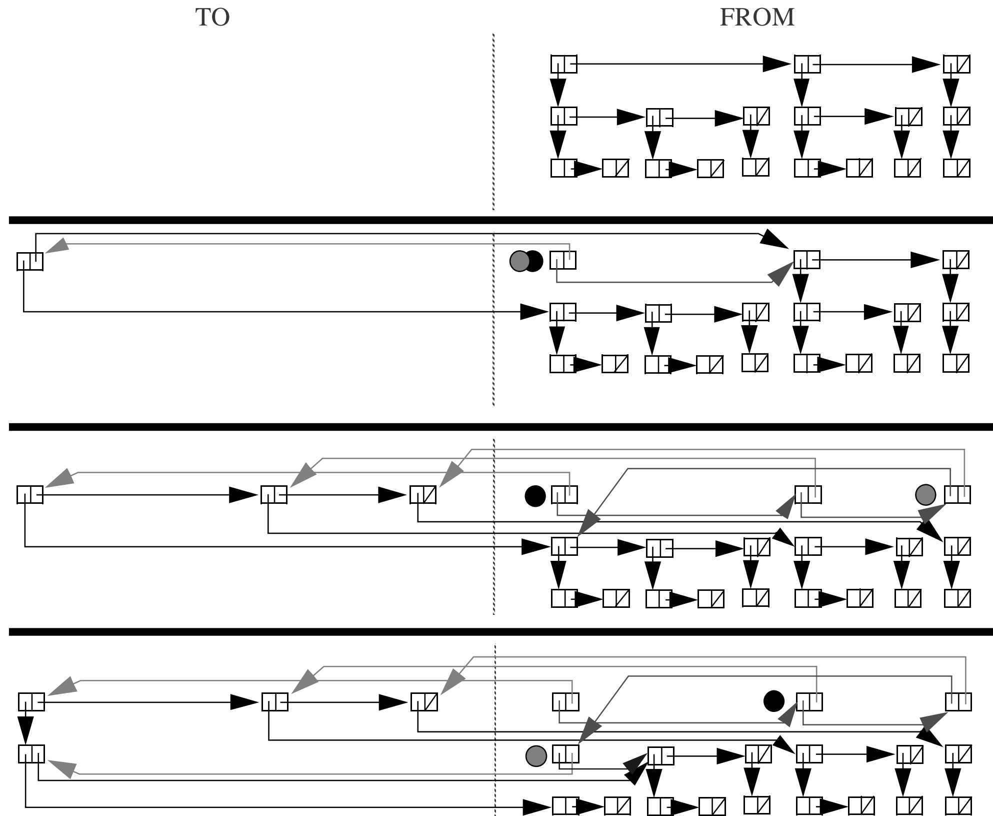

Figure 4. Breadth-first copying of a list with depth=3

2. How the state of a partially copied list is captured efficiently (in time and space).
3. How type information is retained. That is, how does the algorithm keep track of the depth of the cell (which determines the type of the cell) that is currently being copied?

Various stages of a breadth-first copy of a list of depth three are illustrated[1] in figures 4 and 5. There are two pointers, which we call **`grey`** and **`black`** corresponding to the grey and black circles in the illustration, that both traverse the list in a breadth-first fashion. The **`grey`** pointer always points to the cell in FROM space that was most recently copied. The **`black`** pointer always points to the parent cell of the next sublist that **`grey`** should encounter. Thus, when **`grey`** gets to the end of a sublist, the next sublist is found using the **`black`** pointer.

The three issues above are resolved as follows:

1. After each cell is copied into new space, the **`cdr`** field of the old cell contains the forwarding address, as usual. The **`car`** of the cell, however, is modified to point to the next cell in the list to be visited. If the next cell was part of the same sublist, then it is found

1. We have chosen to illustrate the algorithms using figures rather than code (and space limitations precluded doing both). We hope the reader finds the figures enlightening!

- The value of SS must be the same as when the gc_routine suspended.
- The suspended gc_routine is the same as the one for the activation record that SS currently points to.

The first condition means that the activation record *R* being traced previously is still on the stack. The second condition does not mean that the function corresponding to *R* has not executed since the last garbage collection, but only that garbage collection has resumed at the same call instruction within the function. For example, the call instruction may be inside a loop and get executed many times. It is still safe, however, to resume the suspended gc_routine if the conditions are met. The number, stack locations, and types of the variables in the activation record will be the same as during the previous garbage collection. The only possible change could be in the value of those variables. If the value of a variable has changed, this may have two effects:

1. The variable may contain new cells created since the last garbage collection. If so, these cells must already be in TO space and therefore need not be traversed by the garbage collector.
2. Cells in FROM space that will be traversed by the suspended gc_routine may no longer be reachable from the variables in the activation record. At worst, this means that unreachable cells will be copied into TO space (this property certainly exists in Baker's algorithm as well).

In both cases, the garbage collection process remains safe.

3. Breadth-first Incremental Tag-Free GC

Our incremental tag-free method differs from Baker's algorithm in an important way. Recursive structures, such as the nested lists and trees discussed in section 2, are copied in a depth first manner using our method and in a breadth manner using Baker's method. This has two effects:

- During garbage collection the time required to find the next cell to copy into TO space is not constant. In our tree example, this time can be proportional to the depth of the tree. This can be a severe disadvantage in real-time systems, which were the motivation for developing incremental methods in the first place!
- Depending on access patterns to a data structure, a breadth-first copying collector may have desirable data locality properties. If the structures are accessed in a breadth-first manner, then it is beneficial to copy them in a breath first manner to achieve better cache and paging performance.

The problem of breadth-first tag-free garbage collection was left as an open problem in [Appel89]. We now describe an breadth-first copying collection that is both tag-free and incremental.

We first consider homogeneous, but arbitrarily nested, lists. Three aspects of the algorithm must be specified, namely:

1. The transformation of the cells of the list in FROM space after they have been copied into TO space.

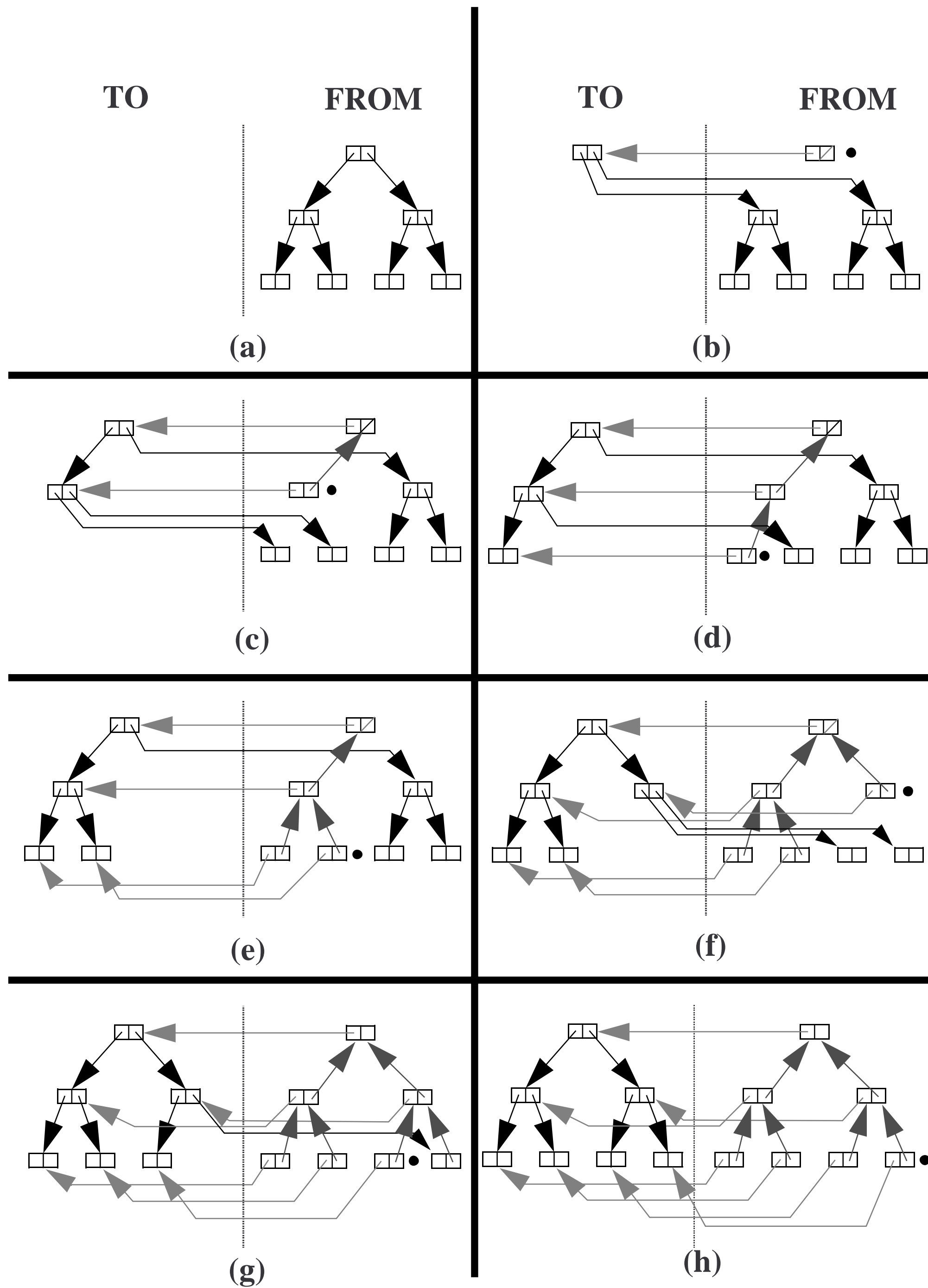

Figure 3. Incremental copying garbage collection using pointer reversal

would simply copy the root node of the tree from FROM space into TO space. While tracing from the newly copied cells in TO space, the rest of the tree would eventually be copied. No stack of visited nodes and no recursion is required.

In tag-free collection, it is certainly also desirable to avoid recursion and the explicit use of a stack to trace the elements of a tree. Not only do we want to avoid using heap space during copying, but also we do not want to save a large data structure representing the suspended tree.

The use of recursion or an explicit stack is avoided through the use of the (well-known) pointer reversal technique (surveyed in [Knuth73]) on the cells of the tree in FROM space. In this depth-first traversal method, as each node *n* is encountered during the traversal, it is modified to point to its parent. Thus, when the subtree rooted at *n* has been traversed, the parent of *n* can be found to continue the traversal. This clearly changes the structure of the tree during the traversal. Most importantly, since garbage collection may suspend in the middle of the tree traversal, it appears that the tree could be left in an inconsistent state.

Luckily, this is not the case. As each node is encountered, it is copied into TO space. Thus, as long as a forwarding address is written into the old node, the rest of the node can be overwritten. For example, in the Standard ML of New Jersey implementation [AM87] a full word is allocated for the value constructor field (**`leaf`** or **`node`** in this case). This field can be overwritten with the forwarding address and one of the remaining fields can contain the pointer to the node's parent. Thus, whenever the gc_routine suspends, the state of the tree remains consistent, being comprised of those nodes that have been copied into TO space and those nodes that have yet to be encountered by the garbage collector. Figure 3 shows the node-at-a-time copying of the graph representing a tree structure. Garbage collection can suspend at any time, and the only information that must be saved is the address in FROM space of the last cell copied into TO space. In each step in figure 3, a cell is annotated by a ● to show which address would have to be saved if garbage collection suspended at that point.

The overhead of this tree copying method is admittedly greater than that of Baker's algorithm. In our method, certain steps in the tree traversal involve following a number of pointers back to an ancestor in the tree (for example, between steps (e) and (f) in figure 3). This is due to our tree traversal being depth first. In the next section we describe a breadth-first copying collector for recursive data structures.

Our algorithm is still incomplete for the following reason: If garbage collection suspends during the tracing of variables in some activation record *R*, there is no guarantee that *R* will still be on the stack when garbage collection resumes. The function call represented by *R* may have returned and, if so, *R* will have been popped off of the stack. Thus the SS register will have been decremented. Another possibility is that *R* is still on the stack, but the gc_routine that suspended is no longer appropriate for resumption. This would be the case if the function represented by *R* has continued executing and the structure (number and types of variables) *R* has changed since the last garbage collection.

Therefore, in order for the suspended gc_routine to resume, the following conditions must be satisfied:

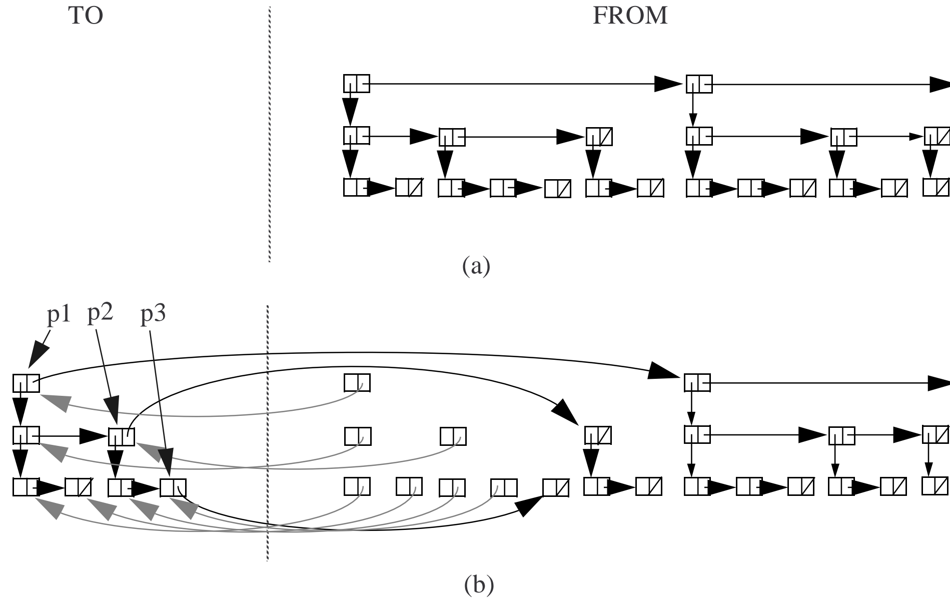

Figure 2. A partially copied list

We must consider the gc_routines that correspond to the types of the variables in each activation record. Since *k* may be less than the number of cells in a single structure, the gc_routine for a given structure may itself have to be suspended. Since the strongly typed languages that we are interested in, like Standard ML, support lists and user-defined recursive data types, we must be certain that a suspended gc_routine can be represented in a small amount of space, independent of the size of the structure.

For the moment, we will use a monomorphically typed version of Standard ML as the target language. The primary predefined data structure is the list. If the gc_routine for a list is suspended, information must be saved describing where in the list the tracing should resume. This information consists of *m* pointers, where *m* is the depth of the list (i.e. number of **`hd`** operations that can be applied to the list before returning a non-list value). The depth of each list is defined by its type and is fixed throughout the computation. The *i*th pointer points into TO space to the last cell at depth *i* in the list that was copied before the gc_routine was suspended. Figure 2 shows a list of depth three that has been partially copied to TO space and the three pointers, labeled **`p1`**, **`p2`**, and **`p3`**, representing the state of the list. The grey arrows represent the forwarding address pointers. When the gc_routine tracing the list resumes, it simply starts tracing at the cells referenced by **`p1`**, **`p2`**, and **`p3`**.

Standard ML (and most other strongly typed languages) allow recursive user-defined types. A common example of this is the definition of a tree in Standard ML:

```
datatype tree = leaf of int | node of tree * tree
```

When encountering a variable of type **`tree`**, a tagged collection algorithm (such as Baker's)

2. Making Tag-Free Garbage Collection Incremental

In order to make our tag-free garbage collection incremental, the following conditions must be satisfied:

- The type information for a heap allocated structure must be available to the garbage collector when the structure is being traced.
- After an incremental collection, memory must be in a consistent state. That is, the value of all reachable structures must remain the same. Furthermore, it must be apparent whether a cell in FROM space has been copied into TO space and, if so, what the address of the new cell in TO space is.
- When TO space is full, all reachable structures in the FROM space must have been copied to the TO space. Thus, the spaces can safely be flipped.

Baker's algorithm satisfies the last two conditions, and the correctness of our method rests on the correctness of Baker's algorithm. However, Baker's algorithm relies on the ability to trace the children (`car` and `cdr` in the LISP case) of the cells that have already been copied into the TO space. The tags of the cells in TO space and the tags of their children are examined to determine how to copy the children. This is not possible in a tag-free garbage collector.

We now describe the incremental version of our tag-free garbage collector. Like Baker, we use a pointer, SS, that, at all times, points to the topmost activation record in the stack that could possibly contain pointers into FROM space (see the description of its use in section 1.1.2). Whenever a new activation record is created (above SS), all cells passed as parameters are first copied to TO space. Thus, the new activation record cannot contain pointers into FROM space. When the spaces are flipped, the SS register is modified to point to the top of the stack. After the flip, all the variables that had previously pointed to the TO space now point to the FROM space.

Instead of tracing from the cells that have just been copied to TO space as in Baker's algorithm, our method must always trace from the variables in activation records on the stack, since the type information is only associated with each activation record. However, the number *k* of cells to be copied by the collector each time may be less than the number of cells comprising the variables of the activation record that SS points to. Thus, it might be necessary for a gc_routine to suspend after *k* cells have been copied. This routine will have to resume during the next collection.

Of primary concern is how a suspended gc_routine for an activation record is represented. A gc_routine for an activation record simply calls the appropriate gc_routine for each variable in the activation record. Since an activation record's gc_routine is non-recursive, its state can simply be represented by a closure containing the address of the next instruction to execute and its local variables. The only gc_routine for an activation record that could possibly be suspended is the one for the activation record that SS points to. Thus, a single closure is sufficient to represent the suspended state of the garbage collection process and can be stored in some special area of memory (i.e. not in the heap). The maximum storage needed to represent a suspended gc_routine is known at compile-time and can be allocated in advance.

In 1991, we [Goldberg91] described a compiled method that, like Appel's method, used the return pointer of an activation record to find the garbage collection routine to trace the variables in the activation record. We then described how common program analyses, like live variable analysis, can optimize the garbage collection process by not copying reachable structures that will not subsequently be referenced.

Details of a tag-free garbage collection algorithm using the compiled method for a monomorphically typed language can be found in [Goldberg91]. For our purposes here, it is sufficient to describe it as follows:

- For each function *f* in the program, the compiler generates a garbage collection routine, or *gc_routine* for short, for tracing the variables in an activation record for *f*. Actually, since the number and types of variables in the activation record may differ at different points of execution of the body of *f*, several gc_routines are generated – one for each function call in *f* during which garbage collection could occur. Garbage collection can only occur during a function call, since the only way to allocate heap space is by calling a primitive function such as **`cons`**.
- In the code for *f*, the address of a gc_routine is associated with each call instruction (at some offset within the code segment). Thus the appropriate gc_routine for an activation record for *f* can be found by examining the return address of the next activation record on the stack.
- If a variable in *f*'s activation record is bound to a closure representing some function *g*, the type of *g* will not necessarily reflect the types of the variables stored in the closure. Therefore, associated with the code for *g* (pointed to by the closure's code pointer) is a gc_routine for tracing the variables in the closure. This gc_routine is found at some offset from the start of *g*'s code.
- When the garbage collector encounters an activation record for *f*, it simply calls the gc_routine associated with the current call instruction in *f*'s code.

In a copying collector, whenever a cell is copied a forwarding address must be left behind. Since pointers are not tagged, how can a forwarding address be distinguished from an integer with the same value? The answer relies on knowing the type of the cell. If it is a cons cell, then in a strongly typed language the **`cdr`** field of the cell must contain a pointer. Thus, the forwarding address is placed in the **`cdr`** field, and if the **`cdr`** of a cons cell contains a pointer into TO space, then it must be a forwarding address.

In all of the tag-free collection algorithms, the garbage collector is invoked when the current space is exhausted, and runs until it has finished copying all reachable heap-allocated structures into the other space. To be incremental, Baker's algorithm relies on tracing the fields of cells already copied into TO space. Since the tag-free collection algorithms trace only from the stack, it is not obvious how they can be made incremental. This is the problem that we have solved.

same as the ratio of the number of variables in the stack to the number of cells in use. The value of k′ can be recomputed at each flip. See [Baker78] for details.

1.2. Garbage Collection without Tags

The fundamental idea behind tag-free garbage collection is that compile-time type information is retained at run-time, but is not retained in the form of type tags. When the garbage collector traverses a data structure, it must be able to find the type information for that structure. How that type information is found, and how it is represented, differs in previously published algorithms.

In 1970, Branquart and Lewi [BL70] described two tag-free garbage collection algorithms for Algol68. The first, called the *interpretive method*, associates a template with each type that describes the structure of variables of that type. As the garbage collector traverses a data structure, it also traverses the corresponding template to determine the appropriate actions. The second method, called the *compiled method*, associates a garbage collection routine (which we shall refer to as a gc_routine) with each type in the user program. The gc_routine is executed when the garbage collector encounters a variable of the corresponding type. These garbage collection routines are produced by the compiler based on the types defined in the user's program.

Both methods described by Branquart and Lewi rely on a run-time table mapping stack locations representing local variables to type templates (interpretive method) or garbage collection routines (compiled method). This table must be updated whenever a new local variable is created. As the garbage collector traverses the stack, it looks in the table to find the type template or garbage collection routine for each variable. Because Algol68 discourages heap allocation of structures bound to local variables, the table seldom needs to be updated. Type templates or garbage collection routines can be associated directly with global variables, since their location is fixed during the computation.

In 1975, Diane Britton [Britton75] extended both the interpretive and compiled methods for Pascal. Instead of a table mapping locations to type templates or garbage collection routines, each activation record in the stack contains an extra pointer to a template or garbage collection routine corresponding to all the variables in the activation record.

In 1989 Peyton Jones and Salkid [PJS89] described a garbage collection scheme for their tagless graph reduction abstract machine. Because they implement a lazy language, each data structure is contained in a closure (that may represent an unevaluated value). The closure contains a pointer to a table of appropriate routines for the closure, including the garbage collection routines. It is not clear how this method would be adapted for strict languages like Standard ML.

Also in1989, Andrew Appel [Appel89] outlined an interpretive method in which the template describing the variables in each activation record is accessed via the return pointer stored in the activation record. The beauty of this method is that no run-time overhead is incurred during normal execution. The only overhead occurs during the garbage collection process itself. Appel also suggested how tag-free garbage collection could be extended for polymorphically typed languages.

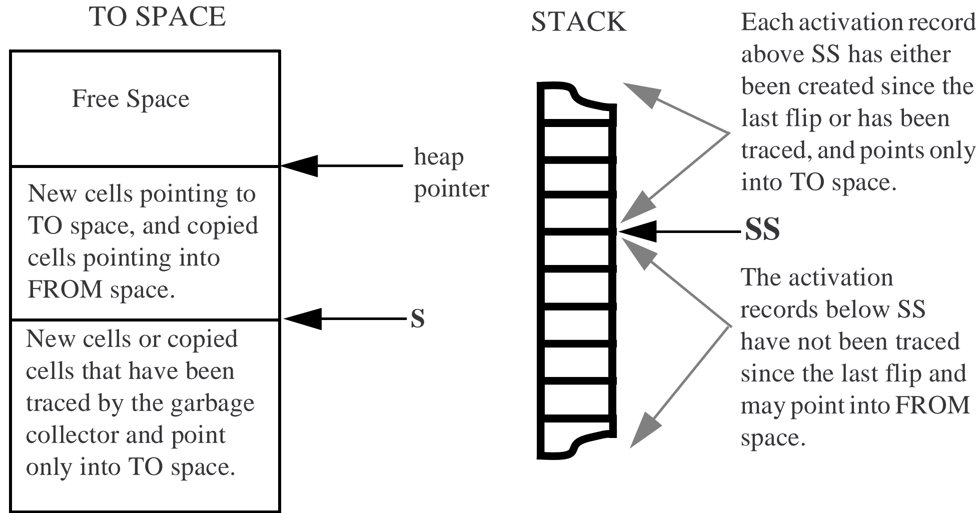

Figure 1. Stack and TO space organization in Baker's algorithm (simplified).

mental collection algorithm did not copy enough cells from FROM space to TO space. The first problem, of course, cannot be solved by the system. The second problem, though, corresponds to question Q3 listed previously.

The solution is to be sure that each time the garbage collector is invoked, a sufficient number of cells are copied to guarantee that they will all be copied by the time TO space is exhausted. However, if too many cells are copied each time, there will be too long a delay for real-time programs. In Baker's method, the garbage collector is invoked at each call to **`cons`** and copies k cells into TO space. Baker's analysis (based partly on [Wadler76] and [Muller76]) is as follows: Assume that when the spaces are flipped, there are N accessible cells in FROM space. In order to guarantee that all N cells can be copied into TO space before TO space is exhausted, it must be the case that

$$k \geq \frac{N}{H - N}$$

where H is the size of each heap. Put another way, if it is required that the program be interrupted for no longer than the time to copy k cells, for some k, then the size H of each heap must satisfy:

$$H \geq N\left(1 + \frac{1}{k}\right)$$

This assumes that the number N of accessible cells stays relatively constant during execution. This situation is called an equilibrium condition, because old cells are discarded at the rate that new cells are created.

Since the garbage collector must also traverse the stack, each time the collector is invoked during a **`cons`** operation it must also trace k′ locations in the stack, where the ratio k′/k is the

The method, illustrated in figure 1, is as follows:

- There is an extra pointer, called SS, into the stack. It points to the topmost activation record which could possibly contain pointers into FROM space. Each time the heaps are flipped, SS is reset to point to the top of the stack. At that time all pointers in the stack point to FROM space. When garbage collection occurs, the activation record that SS points to is traversed and the cells that its variables point to are copied into TO space. The variables in that activation record are updated with the new addresses in TO space. When all the variables in the activation record have been updated, SS is decremented to point to the activation record below. In addition, if the activation record that SS points to is popped off the stack, then SS is decremented.
- When SS reaches the bottom of the stack, all the variables in the stack point to TO space. In order to guarantee this, whenever a new activation record is created (and it must be above SS), all variables in the activation record must point into TO space. Therefore, any cell passed as a parameter to the activation record must first be moved to TO space.
- An extra pointer into TO space, called S, is used. It points to the oldest cell in TO space that could possibly contain a pointer into FROM space. That is, it must point to the oldest cell that was copied from FROM space but whose children have not been copied. Only those cells in TO space between S and the heap pointer (where new cells are allocated) could possibly point into FROM space.
- Each time the garbage collector is invoked, the **`car`** and **`cdr`** fields of the cell that S points to are examined to see if they contain pointers into FROM space. In order to do this, their type tags must be checked (to distinguish pointers from integers). If they do point into FROM space, then the cells that they point to are copied into TO space (if they haven't been already) and the pointers are updated. S is then incremented to point to the next cell in TO space. When S reaches the heap pointer, none of the cells in TO space can point into FROM space.
- When SS reaches the bottom of the stack and S has reached the heap pointer, then all reachable cells have been copied into TO space and all reachable pointers point into TO space. At any time after this, the heaps can safely be flipped.

Figure 1 shows how the stack and TO space are logically partitioned by SS and S. It illustrates a simplified version of Baker's algorithm since, in the actual method, new cells are allocated in a different portion of TO space than copied cells.

Baker's algorithm copies list structures in a *breadth-first* manner. The first cell of the list is copied into TO space. When S points to that cell, it's children are copied into TO space, and so on. This has implications for data locality, which can affect cache and paging performance, since adjacent cells in a list occupy adjacent locations in the heap.

If TO space is exhausted before SS reaches the bottom of the stack or before S reaches the heap pointer, then execution cannot continue. This may have happened for one of two reasons: Either the data accessible by the user program is greater than the size of each heap, or the incre-

operation a few cells are copied from FROM space to TO space as well. Since only a few cells are copied at a time, program execution is never suspended for very long. Eventually, all reachable (i.e. user-accessible) cells in FROM space will have been copied into TO space. At that point the heaps can be flipped.

It will be helpful for our discussion to think of incremental garbage collection as a process that suspends and resumes many times during execution, and can run only for short periods. Of course, the cost of suspension and resumption must be very low.

To understand Baker's algorithm, three questions need to be answered:

Q1. How can one guarantee that the meaning of the program is not changed by the incremental movement of cells?

Q2. How are the appropriate cells in FROM space found in order to be copied each time?

Q3. How can one guarantee that when TO space is exhausted, all the reachable cells in FROM space have been copied to TO space and the heaps can be flipped?

The details can be found in [Baker78]. We will describe as much of Baker's algorithm as is necessary to provide a basis for understanding our method.

Two facets of Baker's algorithm (and the serial and concurrent collection algorithms that Baker based his work on, such as [Steele75], [DLMSS75], [Lamport75], [FY69], [Minsky63] and [Cheney70]) answer question Q1:

- As the contents of each cons cell is copied into TO space, the old cell is overwritten with the new address of the cell in TO space. In other words, when a cell is copied, a forwarding address is left behind.
- When a cons cell in FROM space is referenced, a check is made to determine if the cell has been copied to TO space. If so, its forwarding address is followed to find the correct cell. A forwarding address is always recognizable, because its type tag indicates that it is a pointer and no ordinary pointer in FROM space can point to TO space. In addition, whenever a cell is accessed via a `car` or `cdr` operation, it is also copied into TO space.

Thus, any access to a cons cell will return the same result whether or not the cell has been copied. For this reason, the use of this incremental garbage collection method will not change the meaning of a program.

To answer question Q2, let us assume a stack-based implementation, in which variables in an activation record may point into the heap. The garbage collector must be sure to copy all the cons cells reachable from variables in the stack. Thus by the time the heaps are ready to be flipped, the following cells must have been copied into TO space:

- All cells in FROM space pointed to by variables in the stack, and
- All cells in FROM space pointed to by cells in TO space.

So, each time the garbage collector is invoked (i.e. at every call to `cons`) it copies a few cells in FROM space referenced from the stack, and a few cells in FROM space referenced by cells in TO space.

not require tagged data. However, these methods, like most copying collectors, have the stop-and-collect property in which the execution of the user program may be suspended for a long period of time during garbage collection. This is usually unacceptable for real-time programs.

A copying collector for LISP that does not exhibit the stop-and-copy behavior was described in [Baker78]. The algorithm is *incremental*, that is, the garbage collection process occurs frequently throughout execution, but only copies a few structures at a time. Thus, the user-program is suspended for very short periods of time. However, the incremental garbage collector relies heavily on the use of LISP's run-time type tags.

In this paper, we describe a garbage collection method for strongly typed languages that is both incremental and tag-free. We then extend the algorithm so that it copies in a breadth-first manner, similar to LISP copying collectors, for improved performance and locality. Finally, we describe an incremental, tag-free garbage collection algorithm for polymorphically typed languages like Standard ML.

1.1. Related Work

There are two areas of work related to this paper. The first is the area of incremental copying garbage collection, which is itself based on the more general topic of copying garbage collection. The second area is tag-free garbage collection. Both of these topics have been studied for many years, but, up until now, there have been no incremental tag-free garbage collectors.

1.1.1 Copying Garbage Collection

Algorithms for copying garbage collection are well known and we will spend little time discussing them. The basic idea is that there are two heaps, typically called TO space and FROM space. At any time during execution of the user program only FROM space is in use. When FROM space is exhausted, all the structures (e.g. cons cells in LISP) that can be accessed by the user program are copied into TO space. Exactly which cells are accessible by the user program is determined by a search (either depth-first, breadth-first, or a combination of the two) starting from all locations that correspond to variable names in the program. During the copying phase, the user program is suspended. When the user program resumes executing, the heaps will have been flipped – TO space has become FROM space and vice-versa.

One of the advantages of a copying collector over the simple mark-and-sweep collector is that free storage is compacted. This makes storage allocation very inexpensive, and is accomplished by simply incrementing a pointer (called the heap pointer) that points to the beginning of free space. The disadvantage is, of course, that two heaps are required. There are many varieties of copying collectors, and they are surveyed in [Cohen81] and [Rudalics88].

1.1.2 Incremental Copying Garbage Collection

In 1978, Baker [Baker78] described a method (generally referred to as "Baker's Algorithm") for performing incremental copying garbage collection in LISP implementations. Like the basic copying collector, there are two heaps. However, at any given time both spaces may contain live data structures. New cells are allocated in TO space, and each time a cell is allocated by the `cons`

Proceedings of the 4th European Symposium on Programming (ESOP'92), Springer-Verlag LNCS #582, February 1992.

Incremental Garbage Collection Without Tags

Benjamin Goldberg

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University[1]

Abstract

Garbage collection algorithms that do not require tagging of data have been around since the early days of LISP. With the emergence of strongly-typed languages that require heap allocation, interest in tag-free garbage collection has increased. Several papers published recently describe methods for performing tag-free copying garbage collection by retaining compile-time type information at run time. However, all of these algorithms have the "stop and collect" property, in which program execution is suspended for a significant amount of time during garbage collection. For many programs an incremental garbage collection method, in which the garbage collection overhead is spread evenly throughout the computation, is desirable.

Methods for incremental copying garbage collection have been around since the 1970's. However, these algorithms (the most notable of which is Baker's algorithm) rely on tagged data. In this paper, we present a method for performing incremental copying garbage collection without tags. We then extend this method to work for polymorphically typed languages, and to provide breadth-first copying for improved performance and data locality.

1. Introduction

Copying garbage collectors have been around for a long time, and have traditionally been used for LISP systems. These algorithms typically rely on tagged data to determine the correct handling of each heap allocated structure. In dynamically typed languages like LISP, type tags are required by the run-time type checker and can also be used by the garbage collector.

With the advent of a number of strongly typed languages, such as ML [MLH90], that utilize heap storage, run-time type tags are used solely for the purpose of garbage collection. The overhead required to maintain tagged data can be significant for the following reasons:

- Extra space is often necessary to store the type tag for a datum.
- If a few bits of an integer or pointer are used for the tag, this reduces the size of number that can be represented (leading to more bignums, if supported), reduces the accessible address space, or forces word-alignment on byte-addressable machines.
- Before arithmetic operations or pointer dereferences can occur, the type tags must be stripped from the operands and reinstated in the result.

Recently, several papers [Appel89][Goldberg91] described copying garbage collectors that do

1. Author's address: 251 Mercer Street, New York, NY 10012, USA. Email: goldberg@cs.nyu.edu. This research has been supported, in part, by the National Science Foundation (#CCR-8909634) and DARPA (DARPA/ONR #N00014-91-J1472).