Compile Time Garbage Collection Using Reference Count Analysis

Lucy Hederman

Abstract

Storage management overhead accounts for a considerable proportion of execution time in straightforward implementations of languages with dynamic storage allocation. Our approach to reducing this overhead is to shift some of the work to compile time. With static reference count analysis, we determine when an object becomes inaccessible, and insert explicit deallocation code into the program at those points. This will reduce the frequency and number of garbage collections at run time.

We discuss previous approaches to storage analysis and show that reference count analysis can allow more precise modeling of storage accessibility than other approaches.

The analysis is extended to the interprocedural case. It is formulated as a set of path problems on a specially designed summary graph. The effectiveness of our implemented optimization on a variety of programs is presented.

Acknowledgments

I wish to thank Hans Boehm for suggesting that I write a Master's thesis, for providing the idea, and for endless advice and assistance. David Chase deserves thanks for taking the time to clarify some points about his work; Keith Cooper, David Callahan, Lori Pollock and Preston Briggs for assistance with the interprocedural analysis algorithm, and Uli Kremer for profitable discussions about it. Scott Comer and Lori Pollock provided editorial comments on this dissertaion.

Ivy Jorgensen, Leah Stratmann, and Vicky Riffle assisted with administrative details throughout my time at Rice. Scott Comer, Bill LeFebvre, Dave Johnson and others patiently instructed me in the vagaries of Unix, Emacs, Rmail, LaTeX, Suntools, etc.

Special thanks are due to Scott, David B., Frank and Kim, Soroor, Laura, the Green family, and everyone at Rice and in Houston who made this two years an invaluable, memorable, and mostly pleasant experience.

Back in Ireland, Robert, Catherine, my siblings (Linda, Wendy, William and Simon) and many friends provided encouragement and an incentive to finish this dissertation. I am eternally grateful to my parents, Billy and Carmencita, for having always known the ideal balance between pushing me upwards, onwards and outwards, and spoiling me rotten!

Contents

Chapter 1

Introduction and Motivation

In early programming languages, such as ForTran and Cobol, storage for variables is allocated statically. This scheme is simple and cheap to implement, but imposes some severe restrictions on programming flexibility. For example, recursive procedures cannot be accommodated, and all data objects must be of fixed size and shape.

Greater flexibility is available with dynamic stack allocation. Algol-60 is the standard example of this scheme. PL/1 and C also use stack allocation for most objects except those that the programmer explicitly specifies should be statically allocated. Typically all allocation takes place at procedure entry and deallocation takes place at procedure exit. Stack allocation allows implementation of recursive procedures. It is more expensive than static allocation since some amount of execution time is spent allocating and deallocating.

Dynamic heap allocation is the method of choice in modern high-level programming languages. Dynamic heap allocation is "the simplest, most natural and most powerful method from the programmer's viewpoint" [MJ76]. However it is also the most expensive. Unless the programmer is expected to explicitly deallocate storage, special techniques are needed to recover it, since storage is generally not an unlimited resource. Storage reclamation contributes a sizable proportion of execution time in languages such as LISP or Russell where heap deallocation is implicit. Reclamation techniques are discussed in the next section.

In Russell [BDD85] a programmer has no control over the allocation policy used for the various kinds of values. The semantics assume that all storage is heap allocated. But a correct implementation need not actually allocate everything on the heap. To do so would be wasteful. In general, since Russell functions are first class objects, function closures and function activation records must be allocated on the heap. Variable cells must also be heap allocated since they may be returned to a surrounding function. For other objects the flexibility of heap allocation is not required. The following chapter looks at the problem of wasted flexibility in more detail, and discusses techniques for switching to cheaper allocation policies where possible.

1.1 Storage Reclamation

In dynamic heap schemes a storage allocator hands out storage cells, from a limited supply. When this supply runs out, it is necessary to reclaim cells which are clearly no longer needed, that is cells which are not accessible from the user's program. These are "garbage" cells and the reclamation of them is called "garbage collection".

There are basically two approaches to garbage collection. For a detailed survey see [Coh81]. *Mark-sweep collectors* maintain free lists of storage cells. Garbage collection proceeds in two phases. First accessible storage is identified and marked in a depth first search from a program's variables and from pointers on the stack. Then the collector sweeps over the heap and places all unmarked storage onto a free list. Generally, storage allocation cannot be resumed until collection is finished, making this scheme unsuitable for real-time applications. *Copying collectors* [Bak78] divide memory into *newspace* and *oldspace*. Storage cells are allocated from newspace. When this fills up the roles of the spaces are flipped and the copying phase begins. Accessible

objects in oldspace are identified, in a breadth first search, and copied to newspace. A tagged forwarding address is left in the old cell. When other pointers to the cell are encountered they are replaced with the forwarding address. Storage allocation can resume as soon as the spaces have been flipped, making copying collectors suitable for applications with real-time constraints.

Generation scavengers [LH83, Ung84, Moo84] are copying collectors which optimize garbage collection for short-lived objects. They are based on two observations: short-lived objects are much more common than longer-lived ones in LISP-like environments, and it is wasteful to copy relatively permanent data on every collection. Generation scavengers place objects which have been copied some number of times in a special area, which is collected rarely, if ever. These collectors are proving very fast and eliminate much of the overhead heretofore associated with dynamic heap allocation. In certain circumstances, given a large amount of available memory, heap allocation can become even more efficient than stack allocation [App87]. In these environments most storage allocation optimizations (see chapter 2) become unimportant.

1.2 Conservative Garbage Collection

Unfortunately copying collectors, including generation scavengers, are not suitable for all environments. They require that pointers be distinguishable from other data. It may not be desirable or feasible to maintain enough run time information to distinguish pointers with certainty. If storage allocation is relatively infrequent in comparison with other operations, the overhead of maintaining sufficient information to identify pointers at run-time may dominate other storage management overhead. If

the client program was, at least partially, generated by a conventional (say C, or Pascal) compiler, it will not maintain the necessary information. Even if the compiler was designed to support automatic storage management, similar problems may occur if a general purpose code generator is used. Maintaining tag information to identify pointers may introduce incompatibilities in data representation that are viewed as unacceptable. In these situations a "conservative" collector is needed [BW88, Rov85]. A conservative collector treats any bit pattern that represents the address of a valid object as though it were a pointer. It must retain any object which could conceivably be accessible, and thus may fail to reclaim some garbage.

As a consequence of possibly treating an integer as a pointer, the collector cannot move any object. Suppose some accessible bit pattern is assumed to be a pointer, and the referenced cell is moved. The supposed pointer must be changed to refer to the new cell location. But if the bit pattern is actually an integer it is not correct to change its value. Copying collection is not possible. Conservative collectors therefore use mark-sweep collection. These collectors can incorporate various techniques to speed up recognition of valid heap objects, and to reduce the probability of misidentification of integers as pointers.[1] However it is unlikely that the performance can be brought in line with that of generation scavengers. Thus it remains important to minimize the number of heap allocations and garbage collections.

1.3 Russell Storage Management

The Russell run time environment uses the conservative collection scheme described in [BW88]. The storage allocator maintains a separate free list for objects of different

[1] With the scheme described in [BW88], the collector fails to reclaim garbage exceedingly rarely.

sizes. It is therefore feasible to return a cell to the storage allocator.[2] Explicit deallocation of an object of specified size costs 3 or 4 machine instructions. Garbage collection is relatively costly, in a typical case taking perhaps 100 instructions per reclamation of a small object. This is the key to the optimization which we propose in this dissertation and which has been implemented in the Russell compiler.

Straightforward implementations of Russell produce very storage allocation of intensive code. Various storage optimizations have already been implemented in the Russell compiler [BD86, BH88]. These include stack allocation of activation records and variable cells where possible. But some obvious "waste" of heap space remains. Russell is used for many semi-numerical computations. The bignum (unbounded integer) facility is used as the foundation for a Russell implementation of the constructive reals, which is the subject of on-going research at Rice [BCOR86, Boe87]. Since the size of bignum objects is unknown at compile time, in the absence of further analysis they must be heap-allocated. This means that all bignum temporaries, as well as those mentioned explicitly in a program, are on the heap. Most of these temporaries are very short-lived. Yet they consume heap-space very quickly.

There is a similar problem with floating point temporaries. They are double precision numbers, and do not fit into the standard registers. To allocate them on the stack would involve less work than for bignums, but it has not yet been looked into.

This work is targeted at these and other temporary heap objects. We set out to slow down consumption of the heap by explicitly freeing such objects when it is clear that they are no longer in use.

---

[2]This is not the case for copying collectors.

The analysis which we developed has also been used for improving the efficiency of Russell's implementation of streams [ASS85]. In Russell a stream is represented by a linked list of cons nodes, with a function closure as its tail node. These nodes are differentiated by a cons/closure tag. When evaluation of the tail is forced, a new cons node is created, with a new closure node as the tail. The old function closure almost always becomes inaccessible at that point. But sometimes it won't, and therefore, in general, it cannot be explicitly deallocated. If it can be ascertained that the closure node has no references to it other than from the preceding stream node, this fact can be included with the cons/closure tag. The stream implementation uses this information to discard the node when once it has been evaluated. The analysis which will be described in this dissertation has allowed this optimization to be implemented.

1.4 Compile Time Garbage Collection

Garbage collection is the process of identifying and reclaiming inaccessible storage cells. At compile time data flow analysis techniques [ASU86] can be used to identify points in a program where the last reference to a heap object is removed. By inserting code at these points to free the object, we shift some of the overhead of the marking phase of garbage collection from run time to compile time. This is generally an optimization. By returning storage to the free list as it becomes garbage, we postpone the moment when the free list becomes empty, and reduce the total number of garbage collections needed. We call this optimization "compile time garbage collection".[3] It is the approach to storage optimization which we have pursued. The compile time analysis we use to identify when storage becomes inaccessible is based on Hudak's

[3]Perhaps the optimization should be called "compile time marking" since it is only marking overhead which is shifted to compile time.

reference count model [Hud86], which is an abstraction of an early automatic storage reclamation technique, outlined here.

1.5 Reference-Count Based Storage Management

Reference counting is a conceptually simple technique. An object must be garbage if there are no references to it. In a reference count based reclamation system, each run time object has an associated *reference count*, which is the number of references to it (from variables, the stack, and pointers nested inside other objects). This count is initially zero, and must be updated each time a pointer to the object is created or destroyed. When the reference count of an object becomes 0 the object may safely be freed. Any pointers nested in the object are thereby destroyed and the corresponding reference counts must be decremented.

Other than its simplicity, the chief advantage of reference count based reclamation is that the overhead costs are distributed over the entire execution. (With mark-sweep and simple copying collectors program execution is halted while collection takes place). The disadvantages are that extra space is needed for the counts, there is an overhead associated with every pointer assignment, and circular objects are not reclaimed.

1.6 Outline of the Dissertation

Although modern generation scavenging garbage collectors can be very efficient, they are not suitable for all run time environments. Therefore it remains important to re-

duce garbage collection overhead, whether by reducing the number of heap allocations or assisting the collector in its operations. The following chapter discusses these kinds of storage optimizations. All are based on the idea that it is possible to use static (compile time) information about objects and pointers to implement more efficient memory management. Some previous techniques for gathering static information are presented, for comparison with our analysis.

The remainder of the dissertation presents our approach to reducing garbage collection costs, which is to introduce explicit deallocation code wherever it is determined to be safe. The optimization is based on the assumption that the storage allocator maintains a free list to which inaccessible objects can be explicitly returned. Chapter 3 describes the reference count analysis algorithm, which computes the information needed to identify when deallocation is safe. The algorithm is presented in a data flow analysis framework [ASU86, KU77]. We show that compile time reference counting can sometimes provide a more precise analysis of storage accessibility than other approaches.

Chapter 4 extends the reference count analysis of chapter 3 to compute and use interprocedural information. The interprocedural analysis requires flow-sensitive information about procedures. But since the context of calls can be ignored the problem remains tractable [Cal88]. The algorithm differs from traditional interprocedural analysis algorithms. Instead of solving a data flow framework over a program call graph, it solves path problems on a specially designed "flow summary graph".

The algorithms of chapters 3 and 4 have been implemented as an optional optimization phase of the Russell compiler. In chapter 5 the results of applying this optimization to a selection of Russell programs are presented.

The final chapter summarizes the contributions of this work, and mentions possible directions for future work.

Chapter 2

Memory Management Optimizations

Storage allocation policies range from static allocation to implicit global heap allocation. Static allocation represents very early (compile time) binding of variables to storage. Stack allocation binds at procedure entry. Heap allocation is fully dynamic, allowing binding to storage at every assignment statement. Static allocation is cheap, but inflexible. At the other end of the spectrum, dynamic global heap allocation allows flexibility, but this benefit is offset by significant memory management costs. Straightforward implementations of languages in which global heap allocation is the default allocation policy, or in which the heap is implicit, use the heap for all objects, even those that don't need it.

This problem was first identified by Muchnik and Jones, in [MJ76], as the *storage allocation optimization problem*, which they summarize as the problem of determining for each variable in the program the most appropriate storage allocation policy. [MJ76, Sch75, Ste78, BH88, RM88] all describe techniques to switch to static and stack allocation where possible. Ruggieri proposes a new run time memory organization and an associated allocation policy, which falls between stack and global heap policies on the spectrum described earlier [Rug87]. She proposes that each procedure have a local heap and that objects be allocated in the local heap associated with a

procedure whose lifetime contains the lifetime of the object. An entire local heap can then be reclaimed when its procedure terminates. While it will occasionally be necessary to garbage collect a local heap, most garbage will be deallocated in bulk at procedure termination. This scheme and the "lifetime analysis" used to implement it are discussed in section 2.2.

The other way to avoid the costs of dynamic heap management is to attack the garbage collection overhead directly. Efforts in this direction fall into two classes. One of these is to reuse allocated storage without letting it become garbage ("overwriting update" or "destructive update") [HB85, Gop88, Sch75]. This kind of optimization is particularly useful for functional or value assignment style programming languages, for which straightforward implementations copy objects before updating a component, even if the original value is never needed. The other class consists of optimizations which assist the garbage collector in its operations [Sch75, Bar77]. Our insertion of explicit deallocation code falls into this category. Barth's work is described in section 2.3.

In [MJ81], Muchnik and Jones describe analysis aimed at what might be called "storage reclamation optimization". They classify objects according to the most suitable reclamation technique for each object, as follows:

- Objects whose reference count is never greater than one — When a pointer to one of these objects is to be destroyed the object can be deallocated. No other reclamation strategy is needed.

- Objects that never appear in a cycle. These can be reclaimed with reference counting.

- Objects that can appear in cycles. These need to be garbage collected, but the overhead of reference counting is not needed.

Typically a large proportion of objects are never pointed to more than once (92 to 98% in LISP programs [SG77]). Not having to reclaim these cells can result in substantial savings. The analysis constructs finite approximations to the run time data structures. It is considerably more general than the similar methods of Barth (section 2.3).

Most storage optimizations require some static modeling of a program's data structures. In [Sch75], Schwartz presents analysis of SETL programs which can be used to implement various optimizations. Chase extends Schwartz analysis with what he calls "containment analysis" [Cha87a], which was specifically designed for storage optimization. The "lifetime analysis" which Ruggieri uses to implement her local heap memory organization, is closely related Chase's analysis.

These two analyses are described in detail in the following sections. This will provide the reader with a base from which to evaluate the reference count analysis of the following chapter. It is also hoped that it will help readers who are interested in the general area of storage optimizations. However it is not necessary to understand these sections before going on to the next chapter.

Structure graph analysis has been developed for other optimizations. Larus and Hilfinger's alias graph is a conservative summary of the program's structure graph [LH88]. The alias graph is used to detect interprocedural aliases, for computation of conflicting structure accesses.

2.1 Containment Analysis

Chase proposes an analysis which can be used to compute lifetimes of storage which, in turn, is used to decide when destructive updates and stack (or static) allocations are safe [Cha87a]. The analysis computes containment relationships between storage allocated at different definition points. In conjunction with live variable analysis, this information is used to compute lifetimes of storage. The analysis extends the analysis of the SETL compiler [Sch75] by handling more complex containment relationships.

The notion of "containment" is a little vague. Intuitively, b contains a if the storage for a is used to implement b. The essential idea is that if changing the value of an object a causes the value of object b to change then b contains a.

Chase's work was designed for languages in which there are no explicit pointers and copying is "by reference". It seems best suited to value-assignment languages since the analysis becomes very costly in the presence of side-effects.

2.1.1 The Storage Containment Graph

A storage containment graph, or SCG, approximates the containment relationships of a (side-effect free) program. Graph nodes come in two types — *store* nodes represent allocated storage, *def* nodes represent the variable definition and modification statements in the program. *Store edges* go from a def node representing an allocation to the store node for the storage allocated by that definition. An edge $\langle a, \sigma \rangle$ indicates that the storage for the object defined at a might be σ. All storage arising from a particular allocation is represented by a single store node. This form of abstraction is common to most analyses of storage [Hud86, RM88]. Some such abstraction is needed to bound the size of the analysis framework (in this case the size of the graph).

Consequently storage from different executions of an allocation operation cannot be distinguished, which may result in some loss of precision. *Containment* edges join def nodes and are labeled with selector operations. An edge $s : \langle d, e \rangle$ indicates that an object created at e can be extracted from an object created at d by applying an s selection operation to it. The object created at d *contains* the object created at e.

The SCG deals naturally with aliasing. Two variables are *aliased* if they refer to the same storage. In the SCG representation the values resulting from definitions d_a and d_b may be aliased only if the def nodes each have a store edge to the same store node. Furthermore the SCG can express the fact that while a and b may be aliases and b and c may be aliases, it is not possible for a and c to be aliases (see figure 2.1). Such a situation might result from combining the effects of alternate branches.

Incomplete information is expressed with a special def node $\perp$ and a special store node $\sigma_\perp$. There is a store edge from $\perp$ to $\sigma_\perp$, and for each possible selector s there is a selector edge $s : \langle \perp, \perp \rangle$. If there is compile time type-checking, incomplete definitions may be expressed with different $\perp_t$ and $\sigma_{\perp_t}$ for each type t, with the appropriate selector edges. Differently typed $\perp$ nodes can safely use different store nodes because it is assumed that two variables of different types will never simultaneously use the same piece of storage.[1]

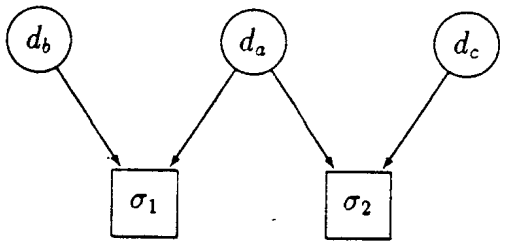


Figure 2.1 Representation of Aliasing in an SCG

[1]This assumption does not hold for certain uses of union types which violate any reasonable concept of type correctness, but are commonly used by C hackers, and the like. More seriously, languages which allow polymorphic types may not fit this assumption.

The construction of an SCG assumes that simple use-definition chains have been computed. For each program statement there is an associated graph transformation, which adds some set of nodes and edges to the SCG. The graph is constructed by iteratively applying these transformations until no more edges may be added.

In Chase's language model five definition types affect the SCG. The graph transformations associated with each type are described here. Value-assignment semantics are assumed. Remember that there may be many def nodes for a variable at its use, corresponding to the set of definitions for the variable which reach the use. The use-definition chains are used to identify the set of def nodes for a variable at a point.

$d : x \leftarrow new[]$

Create a new store node and an edge from d to it.

$d : x \leftarrow new[y_1, \ldots, y_n]$

Create a new store node and an edge from d to it. For each initializer y_i add selector edges s_i from d to the nodes for the definitions that reach y_i. (s_i is the ith field selector for the type of storage allocated.)

$d : x \leftarrow y$

Copy to d all edges from def nodes for y.

$d : x \leftarrow y.s$

Find the set of def nodes for $y.s$. They are the nodes at the end of all s-edges from all def nodes for y. Copy to d all edges which leave these $y.s$ nodes.

$d : x.s \leftarrow y$

Depending on the semantics and implementation, this could be either a copying update, in which case new storage is allocated for x, or an in-place update.

Also s could be an imprecise selector (such as an array index whose value is unknown at compile time) or a precise one. In all (four) cases, the node d should represent the new x value. The non-s fields of the new x value could have any of the values that any existing definitions for x have: copy to d all selector edges from def nodes for x; if the selector is precise do not copy those labeled s — the new x will not have old s values. Also in all four cases, add selector edges labeled s from d to all def nodes for y, to represent the new s field of the definition.

In the case of a copying update, new storage is allocated: create a new store node with a store edge from d. For an overwriting update the storage used by definition d could be any storage used by any definition for x: add edges from d to all store nodes adjacent to def nodes for x.

Note that all the edges added by a transformation for a definition originate at that definition's node.

The derivation of the graph falls into a monotone data flow analysis framework ([KU77], see section 3.2). Since all the transformations are monotonic, increasing and finite the derivation has the Church-Rosser property. Thus transformations can be applied in any order. The construction is potentially an $O(|defs|^2 \times S)$ process (where S is the maximum number of selectors applicable to an object), since this is the maximum number of edges the graph may have.

If analysis of an SCG allows a copying update to be changed to an overwriting one the SCG must be changed accordingly before further analysis can be done. This involves making all definitions that were previously adjacent to the storage allocated by the copying update, adjacent to the storage overwritten. The transformation is of comparable complexity to those defined above.

2.1.2 Use of the Storage Containment Graph

The storage containment graph represents all the relationships which may exist between value definitions and storage. Questions about storage containment and sharing translate into path questions on the SCG. For example "what storage can the results of definition d contain ?" translates to "what store nodes are reachable from d ?", and "what definitions produce results that can contain storage σ (allocated at a particular definition)?" translates to "which def nodes can reach σ ?". This latter question is asked when computing an approximation to the lifetime of storage allocated at a particular definition d. Let Δ be the set of def nodes that can reach the single store node σ_d associated with the allocation d. Using simple use-definition chains on the program's flow graph, find all uses of the definitions in Δ. The live range of storage σ_d is the set of (flow graph) nodes and edges on paths from these definitions to their uses.

The storage containment graph was not designed to identify when storage becomes inaccessible, with a view to inserting deallocation code. But since the graph does express all value-storage relationships it could be used to identify inaccessible storage.[2] Storage "accessibility" is a different criterion from storage "liveness". Storage is no longer *live* past the last use of any reference to it. It becomes *inaccessible* once the last reference to it is killed. In either case the storage can safely be freed. However with the SCG representation there appears to be no way to specify at compile time the address of the storage to be freed. The run time address(es) corresponding to a store node are not known at compile time. Unless some variable or component is

[2]Note that inaccessible storage is not expressed by store nodes which have no incoming edges. There will always be an edge to a store node since no edge is ever removed from the SCG by any transformation. Storage inaccessibility is expressed by the non-existence of a path to the store node from any definition for a *live* variable.

known to definitely refer to the storage which is to be freed, there is no way to specify it.

2.1.3 Some Algorithmic Details

For languages with value-assignment semantics (i.e. no side-effects) a single storage containment graph provides precise information for an entire program. To understand how this is possible, imagine having a separate storage containment graph at each program definition. Each precisely represents the containment information resulting from execution of all paths to that point (and not paths in other branches). Chase has proved that merging any two such SCGs preserves the containment information of both ([Cha87a] section 5.4). The fundamental intuition is that for all transformations, the transformation applied at a definition d only affects the containment information for that definition's node, and no other. As an example of how this allows "containment-preserving" merging of graphs, consider a piece of code where the **then** branch of a conditional incorporates an object a into b, and the **else** clause incorporates the same object into c. The graph for the **then** branch (figure 2.2(i)) shows the containment of a in the definition of b, and the graph for the else branch

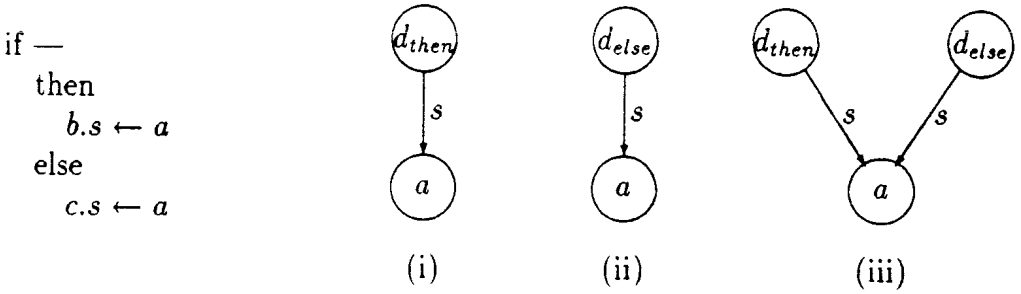


Figure 2.2 A code fragment and the relevant parts of (i) the graph for the **then** branch (ii) the graph for the **else** branch (iii) the merged graph. Since the containment of a definition is not relevant if that definition is not live, (iii) is no less precise than (i) and (ii).

(ii) shows the containment of a in c. In the graph produced by merging these graphs (figure 2.2(iii)) it appears that in both clauses a is contained in both d_{then} and d_{else}. This appears to be less precise than the relationships expressed in the separate graphs. However, since information about what is contained by a definition is only relevant if that definition is live, and since d_{then} is not live within the **else** clause, the apparent imprecision does not matter.

If the language semantics allows side-effects the graph construction algorithm changes. Side-effects occur when an assignment to a field of x could change the value of any def nodes aliased to x. The transformation associated with a side-effect assignment adds more edges to the graph in addition to those added for a side-effect free assignment. For $d : x.s \leftarrow y$, add edges labeled s to definitions for y from all definitions adjacent to store nodes adjacent to d. (Chase does not allow removal of any edges since this would make the construction process non-monotonic and would complicate the sharing of graphs described below. However it appears that the set of edges labeled s from all definitions for x could correctly be removed when $x.s$ is assigned something new.)

The extra edges introduced for a side-effect assignment do *not* originate at that assignment's def node. Thus they may affect the containment relationships of definitions other than the assignment. Therefore it is no longer the case that a single SCG expresses the same containment relationships as the SCGs at each point. In the example of the previous paragraph, not only is the containment of d_{then} and d_{else} affected by the transformations, but perhaps that of some other definition outside the conditional. In a single SCG for the entire program the effects of both branches will be seen in each branch. Now that definitions outside the scope of the branch are affected, a single SCG for an entire program may be imprecise.

So it would seem that we need a separate SCG for every node. Note however that the SCG of a successor of a node P is no smaller than the SCG of P. With suitable data structures an SCG can be used as a base for all its successors. Furthermore all points in a strongly connected component (SCC) must have the same storage containment graph: all points are predecessors of each other and thus for all i, j $SCG_i \subseteq SCG_j$, which implies that for all i, j $SCG_i = SCG_j$. By constructing the graphs for each SCC in topological order, an SCG can be built up from graphs of the SCC's predecessors, saving space and time.

2.1.4 Discussion

This work placed very little emphasis on interprocedural analysis. The strategy which Chase suggests involves substituting a specially "reduced" version of the SCG of the called procedure at each call site. This special version is called a PRUG (partially reduced update graph). The PRUG contains information about storage reachable from returned values, global variables and reference parameters. All other information is irrelevant in an interprocedural context. For recursive procedures the PRUG is generated by successive approximation. This strategy is potentially very expensive.

The work deals with languages with linked data structures where all pointers are implicit; there are no "address of" operations. In languages with explicit pointers some optimizations are missed. Chase is working on some ideas for handling explicit pointers and dereferencing operations [Cha87b].

2.2 Lifetime Analysis

The work of Ruggieri [RM88] is also concerned with the lifetime of storage. But the intended application of her analysis is a little different from Chase's. She proposes an allocation policy in which objects are allocated on a local heap associated with a procedure whose lifetime is guaranteed to be greater than the lifetime of the objects. All the objects in a local heap are deallocated in one chunk when the procedure terminates.

Ruggieri and Murtagh point out, in [RM88] that this policy groups objects of similar lifetimes together, which is the strategy that is proving so successful in generation scavenging collection schemes (see chapter 1 and [Ung84, LH83]). The benefit of Ruggieri's proposed policy is that short-lived objects do not incur the expenses associated with a global heap. She hopes that the local heaps will rarely need to be garbage collected. Even if one of them does, the overhead will be minimal since each heap will be small compared to a global heap. It remains to be seen whether Ruggieri's local heap scheme is practicable.

To implement this allocation policy, detailed information about the lifetimes of objects is needed. The analysis algorithm focuses on procedural languages with dynamically allocated objects and destructive updates. Other than requiring that the language support static type-checking, the techniques are widely applicable. Type information is used to avoid having to make worst-case assumptions about aliasing. The algorithm emphasizes inter-procedural analysis, unlike Chase's. However problems of inter-procedural variable aliasing need to be handled explicitly rather than being implicit as in the SCG representation.

Expressions which allocate storage are referred to as *resolved sources*. Sources approximate run time objects. Each source represents all the objects allocated by the corresponding expression. (Compare with Chase's approximation of all objects allocated by a particular definition, by a single definition node). The problem is to associate each of these with some procedure. During intra-procedural analysis formal parameters and procedure calls must be treated as possible sources of objects. They are represented by *unresolved sources* which serve as place-holders for sets of resolved sources.

The data flow analysis problem, referred to as the *reaching sources* problem, is to determine for each procedure the set of all possible resolved sources that might produce objects created local to the procedure or returned to the procedure by procedures it calls. The algorithm proceeds in three phases. The first, the intra-procedural phase, computes summary information for each procedure. This consists of two parts: the set of all sources which the procedure may return to its caller, and for each argument to each call within the procedure, the set of all sources which could produce that argument. A monotone data flow analysis framework is used. The functions associated with program statements trace the bindings of objects to variables through assignment statements. They also handle the effects of aliasing which accounts for their considerable complexity.

The second, inter-procedural phase replaces each unresolved source by the appropriate set of resolved sources. For example an unresolved source representing a formal parameter will be replaced by the set of sources which could reach the actual parameter. Again a monotone data flow analysis framework is used. On each edge of the call graph a function models the effect of the call on the set of sources returned by the calling function. Each edge has a slightly different function because of the

different association of actual and formal parameter sources. The purpose of the inter-procedural phase is to compute for each procedure the sets of resolved sources returned to the procedure by procedures it calls and by the procedure to its callers.

The final phase uses this information in conjunction with the call graph to determine in which procedure's local heap the objects represented by each resolved source should be allocated. In a single pass over the nodes of a dominator tree for the call graph, it finds the shortest-lived procedure whose lifetime is guaranteed to include the lifetime of the objects created by the source.

Note that ultimately Ruggieri is not concerned with what goes on inside procedures. The behavior of individual assignments, definitions and modifications is of no interest after the intra-procedural phase. This is not the case for Chase's analysis nor ours. Note also that sources are associated with variables rather than with definitions.

The analysis traces the bindings of objects not only to variables, but also to subcomponents of objects. Each variable of a structured type induces *subvariables* for each of its fields. For example the subvariable associated with the s field of variable v is $v.s$. For recursive types the set of subvariables would be infinite. So approximations have to be made at some point. Higher "order" approximations will generally lead to more precise information and to slower convergence. But in many cases the higher order will yield no useful information. Chase has pointed out that by associating information with definitions instead of with variables he automatically achieves information as precise as Ruggieri can achieve by tuning the order.

While Chase's storage containment graph was not designed specifically to solve Ruggieri's reaching sources problem, I believe it contains all the necessary information.

We will see in chapter 4 that the first two phases of Ruggieri's algorithm correspond very closely to the first two phases of our inter-procedural algorithm. Indeed

her algorithm might provide a good starting place from which to extend our analysis to handle nested pointers.

2.3 Shifting Garbage Collection Overhead to Compile Time

Barth, in [Bar77], describes an algorithm which reduces the run time costs of storage reclamation. Some of the work required to maintain reference counts in the collection scheme described by Deutsch and Bobrow [DB76], is shifted to compile time.

Deutsch and Bobrow's collector is intended for a LISP environment, and uses a combination of reference counting and mark-sweep collection. Only nested references (i.e. from the heap) are counted, not those from program variables. The collection phase reclaims objects if they are not referenced from a program variable (i.e. the stack) and their reference count is zero. Counts are recorded in two tables, rather than with the objects. The Zero Count Table (ZCT) contains the addresses of storage cells to which there are no references. The Multiple Reference Table (MRT) records the addresses of cells with more than one reference, and maintains the counts with the addresses. Cells with exactly one reference are not recorded. Since most accessible cells in LISP programs fall into this last category, a considerable amount of space can be saved.

Changes to reference counts are recorded in a transaction file. There are three possible transactions — ALLOC enters a cell address in the ZCT, CREATEREF removes a cell address from the ZCT if it is there, increments its count if it is in the MRT, or enters it in the MRT with count 2 if it is in neither table. DELETEREF is the inverse of CREATEREF. These transactions can be considered to be part of

the program code. For example an assignment $p.a \leftarrow q$ can be considered to be the sequence

DeleteRef $(p.a)$
$p.a \leftarrow q$
CreateRef $(p.a)$

It is intended that the tables be maintained on backing store. Periodically they are brought into memory and updated by processing the transaction file. Certain pairs of transactions on the same cell address cancel each other (ALLOC—CREATEREF, CREATEREF—DELETEREF, DELETEREF—CREATEREF). During the processing of the transaction file this fact can be exploited to reduce the number of operations on the tables.

Barth's optimization involves identifying such canceling pairs at compile time and removing the transactions from the program, thereby avoiding the generation and processing of the transactions at run time. His analysis also allows other transformations as well as the three above. Notably, when an ALLOC transaction is followed by a loss of all variable references to the allocated cell, the ALLOC can be removed and inline code inserted to return the cell to the free list at the point where the last reference is lost (ALLOC—CANCELLATION transformation). This transformation is exactly the optimization with which our work is concerned. However our analysis will find more opportunities for applying this transformation than Barth's does.

The analysis traces the flow of references to newly allocated objects through variables. An algorithm which detects ALLOC—CREATEREF and ALLOC—CANCELLATION transformations is described. The algorithm watches each allocated cell in turn, from the time it is allocated until it is first pointed to from the heap, keeping track, as precisely as possible, of references to it from program variables. If the cell becomes referenced from the heap then the CREATEREF at that point cancels the

ALLOC transaction, and both are removed from the program. If the set of references becomes empty (and the cell is never pointed to from the heap) then an ALLOC—CANCELLATION transformation may be performed.

On each edge of a program's flow graph two bit vectors are computed, representing those variables which may reference the allocated cell (Maybe Set, MS) and those which definitely reference it (Definitely Set, DS). The algorithm which Barth proposes appears to be excessively conservative. For a given allocation site, some of the nodes of the flow graph are assigned a numbering such that no node, other than the allocation node, is numbered before all its predecessors have been. The numbering gives the order in which nodes are visited. (It is not clear how the nodes of a cycle would be numbered). A node is visited by intersecting incoming DS sets and unioning MS sets, and then, if the node represents an assignment that affects the references to the cell in question, the appropriate changes are made to the sets. These changes basically make the membership of the LHS variable in the sets match that of the RHS variable. As described the algorithm does not iterate to find a fixed point. This presumably is related to the ordering of nodes, but seems very conservative.

In the final pass over the nodes, "ALLOC owed" markers are propagated through the graph. The edge out of the first node, which is the allocation node, is marked. At other nodes, if all the edges into the node are marked then the algorithm seeks a canceling CREATEREF or a loss of all references. Markers are updated accordingly. Once all the nodes have been visited, ALLOC transactions are inserted on each marked edge.

The algorithm given in [Bar77] for finding and handling the ALLOC—CANCELLATION transformation does not seem correct. If all the MS sets leaving an assignment are empty then the last reference to some cell is lost at that node. An ALLOC—

CANCELLATION transformation is performed, which inserts code to deallocate the cell which has become inaccessible. It is unclear how Barth manages to get a handle on the cell to be freed, in order to specify it in the inline deallocation code. Its address is not known at compile time. Barth refers to it as "the last allocated cell"; but there is no reason why there could not be another allocation site between the one under consideration and this assignment node. If LHS definitely refers to the cell before the assignment (if LHS is in all incoming DS sets) then the cell to be freed can be specified as "the cell to which LHS refers". However if LHS may refer to other cells, cells about which we know nothing, this won't work. Recall that the same problem arises when trying to use Chase's graph to insert deallocation code.

This work places no emphasis on interprocedural analysis. It is intended for a very specific application, namely a transaction-oriented reference-counted collection scheme in a LISP-like environment. It probably works well at finding canceling pairs of transactions that are close to each other, and very short-lived cells.

Chapter 3

Reference Count Analysis

As outlined in the introduction the purpose of our work is to find, at compile-time, points in the program when the last reference to a heap-allocated object is removed, and to insert explicit deallocation code at those points, which will return the inaccessible object to the heap, thereby assisting with garbage collection. This chapter presents the static analysis which we use to implement this optimization. It is based on reference counting. Our algorithm avoids most of the complexity of previous work by ignoring nested pointers. While this clearly compromises the precision, it allowed us to easily implement the algorithm and to discover that reference counting can, in certain cases, produce more precise information than any of the previous methods. The next stage of our research would be to extend our analysis to handle nested pointers, using techniques from the previous chapter.

3.1 Introduction

Our compile time analysis is based on Hudak's semantic model of "reference counting" [Hud86]. In a run time context the reference count of an object at a given time is the number of references to that object in existence at that time. At run time it is possible to maintain accurate reference counts for all objects. When an object is created (storage is allocated for it) and assigned to a variable (e.g. $x \leftarrow alloc$) it is given a reference count of 1. A pointer assignment, such as $x \leftarrow y$, kills the old

reference from x and copies the reference from y. The reference count of the object to which x refers (before the assignment) is decremented by 1. The reference count of the object to which y refers is incremented by 1. When a reference count becomes 0, the object is freed — returned to the heap. Any references within the object are thereby killed and the appropriate reference counts must be decremented.

The reference count analysis which we have developed approximates this scheme. At compile time we cannot hope to maintain accurate counts. We do not know which paths will be taken through the program, nor what objects will be produced, nor to which object a variable will be referring at an assignment statement. We will need to use global data-flow analysis techniques [ASU86] and suitable abstractions for run time information [CC77] in order to feasibly compute useful information at compile time.

For our purposes a safe approximation to a reference count is one which is at least as great as the true reference count. A safe approximation for the object referred to by a variable at a point is a set of objects which includes the one referred to. To be safe, when a reference (a variable's value) is copied, the reference counts of all the objects to which the variable may refer must be incremented. When a reference is killed, no reference count may be decremented unless it is clear exactly which object is involved; that is only if the set of objects to which the variable may refer contains only one object. For example if at a point where x is assigned y our best approximation to the object referred to by x is $\{i, j\}$ and by y is $\{j, k, l\}$, the best we can do is to say that there may be new references to j, k and l; we don't know what reference is removed. These abstractions follow Hudak's abstract interpretation model of reference counting [Hud86].

This chapter formulates our algorithm for reference count analysis as a *distributive data flow analysis framework* and describes how reference count information is used to allow insertion of deallocation code. For now we assume that there are no procedure calls, or we make worst case assumptions at call sites. Chapter 4 describes an interprocedural analysis algorithm.

3.2 Monotone Data Flow Analysis Frameworks

Global data flow analysis [ASU86] generally seeks the *meet over all paths (MOP)* solution of a data flow problem. The *MOP* solution for a program is the calculation for each point in the program of the maximum ("best") information, relevant to the data flow problem, which is true along every possible execution path from the starting point of the program to that point. "Bit-vector" data flow problems have been studied extensively. A detailed survey of algorithms for these problems can be found in [Ken81].

For problems in which the data to be computed at each point is more complex than a bit-vector, a *data flow analysis framework* provides a structured (lattice-theoretic) way to formulate the problem [Kil73, KU77, ASU86]. A broad class of these problems are *distributive*. For distributive problems an instance of the corresponding framework implicitly defines a set of simultaneous equations whose maximum fixed point solution is the desired MOP solution. Given an instance of a framework, Kildall's algorithm [Kil73] computes the maximum fixed point solution of these equations, yielding the MOP solution if the framework is distributive. A more general class of problems are *monotone*. For these the maximum fixed point solution to the corresponding set of equations may differ from the MOP solution. However it is a safe, and frequently

adequate, approximation. There exists no single algorithm which will compute the MOP solution for all monotone frameworks [KU77].[1]

A monotone data flow analysis framework (MDFAF) $\mathcal{F}$ consists of

- a *bounded semilattice* L, whose elements are the values propagated through a flow graph,
- a *meet* operation $\wedge$, on L, which defines the effect a program "join" has on the values of L, and
- a *monotone function space* F of transfer functions from L to L which are used to define the effect of flow graph nodes on values of L.

Refer to [KU77] for a fuller description.

The following definitions will be useful later in the chapter.

Monotonicity A function f on L is *monotone* iff

$$(\forall x, y \in L)[f(x \wedge y) \leq f(x) \wedge f(y)].$$

Distributivity A function f on L is *distributive* iff

$$(\forall x, y \in L)[f(x \wedge y) = f(x) \wedge f(y)].^{2}$$

A function space is monotone/distributive if all the functions in the space have the corresponding property.

An *instance* of a MDFAF consists of

---

[1] [KU77] gives a variant of Kildall's algorithm which computes a better approximation than Kildall's, but one which may still differ from the MOP solution for monotone non-distributive frameworks.

[2] Clearly all distributive functions are monotone.

- a program flow graph: the nodes of the graph are the basic blocks of the program and the edges represent the control flow; one node is the start node: and
- a mapping from each node of the flow graph to a function in F; the function associated with a node represents the effect of the basic block on the incoming data values.

The following is a description of Kildall's algorithm (based on the one in [KU77].

Kildall's Algorithm:

Input An instance (flow graph, start node n_0, and associated transfer functions) of a framework (bounded semilattice L, meet operation $\wedge$, top element $\top$, bottom element $\bot$, and monotone function space)

Notation $Paths(n)$ is the set of paths in the flow graph from the start node n_0 to n. $Preds(n)$ is the set of predecessors of n.
If n is a node, f_n is the transfer function associated with that node.
If P is a path, f_P is the composition of transfer functions for nodes on the path.

Output $A[n] \in L$ for all nodes n that are the maximum fixed point solution to the simultaneous equations

$$X[n_0] = \bot \tag{3.1}$$
$$X[n] = \bigwedge_{p \in Preds(n)} f_p(X[p]) \tag{3.2}$$

Method Initialize all nodes

$$A[n] = \begin{cases} \bot & \text{if } n = n_0 \\ \top & \text{otherwise} \end{cases}$$

Visit all nodes, iterating to a fixed-point, where a node is visited by

$$A[n] \leftarrow \bigwedge_{p \in Preds(n)} f_p(A[p])$$

[KU77] proves that the algorithm will eventually halt, that it is correct and that the solution approximates the MOP solution :

$$(\forall n) \left[A[n] \leq \bigwedge_{P \in Paths(n)} f_P(\bot) \right].$$

For many problems it is unnecessarily conservative to use the bottom value for the start node. The value used should represent the best safe estimate of the situation before the program begins execution.

The solution gives information about the beginning of a node which is generally a sequence of instructions. Information at other points is obtained by applying the transfer functions for the instructions between the start of its basic block and that point.

3.3 Language Model

The proposed optimization is performed on Russell intermediate code. This is 3-address code which assumes an unbounded number of virtual registers. In this analysis we trace references to heap objects from virtual registers only. When a reference is stored to memory we give up and will never explicitly free the referenced object. In other words we make no attempt to statically model containment nor to trace nested pointers. Clearly this will prevent us from introducing as much deallocation code as we might otherwise be able to. But it is in keeping with our initial concern, which is compile time collection of floating-point and bignum temporaries. It has also allowed us to refine basic "accessibility analysis" in a direction orthogonal to Chase and Ruggieri without producing unwieldy algorithms.[3] An optional optimization in the Russell front-end which allocates as many variables and identifiers as possible to virtual registers greatly enhances our analysis.

Four kinds of intermediate code instruction are of interest.

[3]Of course, the initial reason we did not model containment is that in the intermediate code there is little evidence of data structures.

$x \leftarrow alloc$

Allocates a new heap object and creates one reference to it from register x.

$x \leftarrow nil$

nil represents all values for which deallocation makes no sense (constants, static and stack objects, and miscellaneous values) and objects which we will never deallocate because we have lost track of references to them (all values loaded from memory).

$x \leftarrow y$ (assume $x \neq y$)

Assigns the value of register y, which may be a reference to a heap object, to x.

$mem \leftarrow x$

Stores the value in x to memory.

Note that the first four of these "kill" the LHS value, x, and may cause an object to become garbage. Virtual registers are declared and undeclared explicitly. The intermediate code also contains explicit information about liveness of registers. This information is important for the effectiveness of our analysis. The instruction $x \leftarrow nil$ is used to model x becoming dead or being undeclared.

3.4 Simplest Accessibility Analysis

In this section we will present an analysis which corresponds to a restriction of Chase or Ruggieri's analysis that makes no attempt to model containment of data structures. We will show that this approach is imprecise for reasons unrelated to static modeling of data structures. The information computed by this analysis will be used in the reference count analysis of the next section.

Some opportunities for deallocation can be uncovered with a simple analysis which approximates the set of objects to which virtual registers may refer. When an object no longer appears in any set it must be inaccessible and can therefore be deallocated.

The set of virtual registers in a program, $\{x, y, \ldots\}$, is augmented with a special register *mem* which represents memory, giving $V = \{mem, x, y, \ldots\}$. The objects of interest for our analysis are heap objects, which are generated by *allocators* — instructions which allocate heap-storage. Without running a program we cannot identify the objects which it produces. Instead we approximate heap objects by identifying them with their allocation sites [Cha87a, Hud86, Rug87].[4] This will prevent us from distinguishing distinct objects created by a single allocation instruction, and thus may hide opportunities for deallocation. For convenience we will assume that allocators are numbered 1, ..., *numlocs* and that each carries its unique number with it, as in $x \leftarrow alloc_i$. Values represented by *nil* in the language model are represented by a single object *nil*. $O = \{nil, 1, \ldots, numlocs\}$ is the set of objects. Consider the lattice $\mathcal{P}(O)$, the power set of O, with set union ($\cup$) as its meet operation. This reflects the fact that at a program join a register may refer to any of the things it could refer to on all incoming paths. The top element of the lattice is $\{\}$; the bottom element is O, the set of all objects. The approximation ordering on elements of the lattice is $\supseteq$ (superset ordering). The lattice is bounded since *numlocs* is finite.

Let A map registers to sets of objects: $A = (V \rightarrow \mathcal{P}_O)$. The data flow problem is to compute $a_p \in A$ for each program point p. The interpretation of a_p is that at point p in the running program the register x will not refer to any object other than those (represented by elements) in the set $a_p[x]$. In the framework $\mathcal{F}_A$ for this problem

[4]These object approximations correspond to Ruggieri's "sources" [Rug87]. We will use the word "object" to denote both real, run time objects and object approximations. Occasionally the distinction will become important, and will be pointed out.

the transfer functions associated with basic blocks are composed from the following functions for individual instructions. Throughout these function definitions assume that components which are not redefined are simply copied.[5]

$$\langle x \leftarrow alloc_i \rangle_A(a) = a' \quad \text{where} \quad a'[x] = \{i\}$$

$$\langle x \leftarrow nil \rangle_A(a) = a' \quad \text{where} \quad a'[x] = \{nil\}$$

$$\langle x \leftarrow y \rangle_A(a) = a' \quad \text{where} \quad a'[x] = a[y]$$

$$\langle mem \leftarrow x \rangle_A(a) = a' \quad \text{where} \quad a'[mem] = a[mem] \cup a[x]$$

These functions are distributive with respect to A. The proofs are trivial. Thus $\mathcal{F}_A$ is distributive and Kildall's algorithm computes the MOP solution. The initial value (at the start node) for all registers, including *mem*, is $\{nil\}$.

Once the solution has been computed it is used to uncover instructions which definitely cause an object to become inaccessible, and thus "deallocatable". All instructions which kill a register are considered.

> **Deallocation Criterion:** If the value in register x is about to be killed it's referent may be deallocated if $a \in A$ is the value computed for that point, and if
>
> 1. $nil \notin a[x]$ and
> 2. $(\forall\, i \in a[x])[(\forall\, y \neq x)[i \notin a[y]]]$.

That is x must refer to a heap-object, and none of the objects to which x may refer may have other references (including references from memory). Alternatively, letting $n(i, a)$ be the number of occurrences of i in the sets of a, with an occurrence in *mem*'s set contributing ∞, the second condition can be rewritten:

2′. n(i,a) = 1.

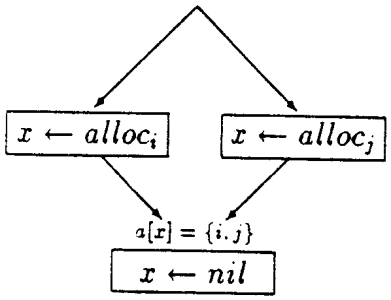


Figure 3.1 Accessibility analysis allows deallocation of x's referent.

For the annotated program fragment represented by figure 3.1, $a[x] = \{i, j\}$ and $\forall\, y \neq x, a[y] = \{nil\}$ at the join.[6] Thus we may deallocate x's referent, whichever object it may be. However accessibility analysis is not general enough to show that in figure 3.2, in which one branch swaps the values of x and y, the referents of both x and y may be deallocated when the register values are killed. When x is about to be

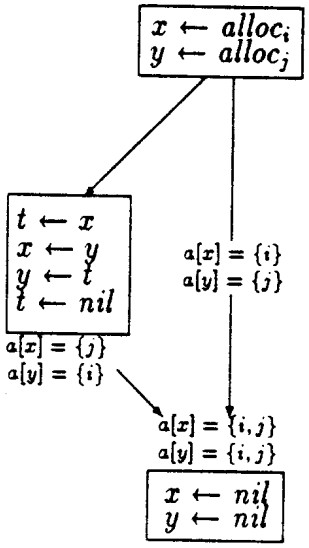


Figure 3.2 Swap program segment: accessibility misses the opportunity to deallocate x's referent.

[5]The notation "$\langle instr \rangle_L$" denotes the transfer function corresponding to the instruction *instr* in the framework $\mathcal{F}_L$ associated with the semilattice L.

[6]In all annotated flow graphs, assume that components of an a value which are not mentioned in an annotation have the value $\{nil\}$.

killed it appears that the objects to which it may refer may have another reference (from y). So x's referent is not deallocated. But in fact no matter which branch is taken there can be only one reference to each object. (This analysis does allow deallocation of y's referent.)

As was mentioned, this is essentially Chase or Ruggieri's analysis restricted to simple objects. Neither of them would uncover both opportunities for deallocation in the example of figure 3.2.

3.5 Adding Reference Counts

For an improved analysis we need to record *how many* references to an object may exist. The scheme was outlined in the introduction to this chapter. It represents an improvement over the previous analysis because it allows us to infer a reference count for an object which is less than the number of occurrences of it in the sets associated with the registers. Since we need to bound the lattice for the reference counting framework (to ensure termination), we must bound the set of values which an inferred reference count can assume. The bounded set of values is $C = \{0, 1, \ldots, maxrc, \infty\}$. Since most objects are not shared very widely even a small value for $maxrc$ will give accurate results.[7] (We have been using 3). The domain is "sticky" — once a reference count reaches ∞ it stays there [Hud86]. Addition and subtraction operations are redefined accordingly.

$$x \oplus n = \text{if } (x + n) > maxrc \text{ then } \infty \text{ else } (x + n)$$
$$x \ominus n = \text{if } (x = \infty) \text{ then } \infty \text{ else } (x - n) \qquad (n \neq \infty, x > n)$$

[7]Often, run time reference-counting implementations, in an effort to save space, use a small fixed-size reference-count field, and periodically collect the garbage thereby missed with a mark-sweep type collector.

The meet operation for C is arithmetic *max*. This reflects the fact that at a join an object can have no more references than it has on any one incoming path. As a consequence of the meet operation, C is ordered by arithmetic $\geq$ and ∞ is *less* than 0 in the lattice. The top element is 0. The bottom element is ∞.[8]

This meet operation explains why reference counting can be an improvement over simple accessibility. Consider figure 3.2 again. Assume that we can infer that on each branch i and j have one reference each. When we meet this information we (correctly) infer that i and j still have one reference each, though we do not know whether the reference is from x or from y. Now when x is about to be killed if x actually refers to i then it must be the only reference, since the reference count analysis tells us that there is only one reference to i. Likewise if x actually refers to j. Therefore the reference from x is the only one to its referent and the referent can safely be deallocated.

Let R map object approximations to inferred reference counts: $R = (O \rightarrow C)$. The data flow problem is to compute $r_p \in R$ for each program point p. $r_p[i]$ is an upper bound on the number of references, to any and all objects created by $alloc_i$, which may exist at point p during program execution. In the framework $\mathcal{F}_R$ for this problem, the transfer functions associated with basic blocks are composed from the functions for individual instructions defined below. The functions depend on the A values which are computed with $\mathcal{F}_A$ of the previous section. The function $kill(x, r, a) \in R$ encodes how killing x affects the reference count lattice element r, given that a records what objects registers may refer to at that point. If x definitely refers to some one object then, and only then, that object's reference count is decremented.

---

[8]For further clarification, the meet of 2 and 1 is 2. That is the *max* operation is the regular arithmetic one. A meet operation on a lattice must return the greatest lower bound of its operands, that is a value on the bottom side of the operands.

$kill(x, r, a) = r'$ where

$$r'[i] = \begin{cases} r[i] \ominus 1 & a[x] = \{i\} \text{ (a singleton)} \\ r[i] & \text{otherwise} \end{cases}$$

Here are the transfer functions:

$\langle x \leftarrow alloc_i \rangle_R(r, a) = r''$ where

$$r' = kill(x, r, a) \text{ and } r''[i] = r'[i] \oplus 1$$

$\langle x \leftarrow nil \rangle_R(r, a) = kill(x, r, a)$

$\langle x \leftarrow y \rangle_R(r, a) = r''$ where

$$r' = kill(x, r, a) \text{ and } (\forall\, i \in a[y])[r''[i] = r'[i] \oplus 1]$$

$\langle mem \leftarrow x \rangle_R(a) = r'$ where

$$(\forall\, i \in a[x])[r'[i] = \infty]$$

See section 3.5.1 for further discussion of the function $\langle x \leftarrow alloc_i \rangle_R$. For $\langle x \leftarrow nil \rangle_R$, note that it is not necessary to count copies of values that are not references to heap objects. If *nil* is actually a heap object being loaded out of memory, the reference count for that object is already ∞.

The transfer functions are distributive and hence Kildall's algorithm computes the MOP solution for this problem. All reference counts are initialized to 0, since this represents the situation before the program starts when there are no heap objects in existence yet. The computed solution is used to decide which instructions must remove the last reference to a heap object.

> **Deallocation Criterion:** If the value in register x is about to be killed it's referent may be deallocated if $a \in A, r \in R$ are the values computed for that point, and if

1. $nil \notin a[x]$ and
2. $(\forall\, i \in a[x])[r[i] = 1 \text{ or } n(i,a) = 1]$

At all instructions satisfying these conditions we may safely assert that the object to which x refers may be freed.

Figure 3.3 shows the "swap" program of figure 3.2 annotated with inferred reference counts. The deallocation criterion for reference counting allows both x and y's referents to be deallocated. Without reference counts this was not possible.

Recall that $n(i,a)$, in the final clause of the deallocation criterion, is the number of occurrences of i in the sets of a (∞ if i is a member of $a[mem]$).[9] That is $n(i,a)$ is the cardinality of the set of registers which may refer to any and all objects allocated by $alloc_i$.[10] The final clause of the deallocation criterion is necessary since it is quite

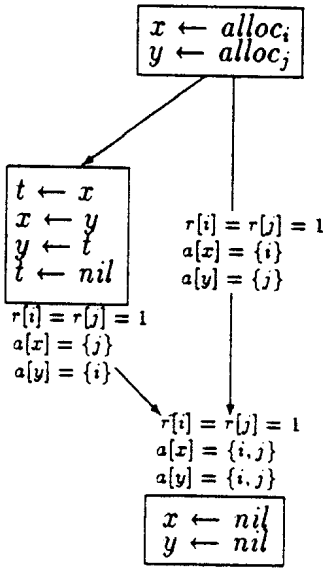


Figure 3.3 Swap program segment annotated with inferred reference counts. Reference count analysis allows both x and y's referents to be deallocated

[9] $n(i,a)$ should be an element of C, the set of "sticky" reference counts.

[10] $n(i,a)$ is NOT the same as $r[i]$. If this is not clear, return to figure 3.3. At the final node $n(i,a) = 2$ (since i occurs in $a[x]$ and in $a[y]$), while $r[i] = 1$.

possible that the inferred reference count of an object will be greater than the number of occurrences of that object in sets associated with registers. The anomaly arises as a result of our being frequently unable to decrement reference counts when a register is killed (c.f. the auxiliary *kill* function above). As an example consider what happens when x is assigned *nil*, with $a[x] = \{i, j\}$. While one occurrence of each of i and j is removed from the sets of a by $\langle x \leftarrow nil \rangle_A$, the reference count of all objects remains unchanged by *kill*. Consider figure 3.4 where the annotations suggest that any solution must satisfy $n = m + 1$ and $m = n$. Since n and m are elements of C, $n = m = \infty$ satisfies. These values are clearly excessive since there is never more than one reference to either object. Even the simple accessibility analysis can infer this since $n(i, a)$ and $n(j, a)$ are never greater than 1. The "or $n(i, a) = 1$" clause of the deallocation criterion allows x's referent to be deallocated at the $x \leftarrow nil$ instruction. In section 3.7 we will present a more precise solution to the anomaly.

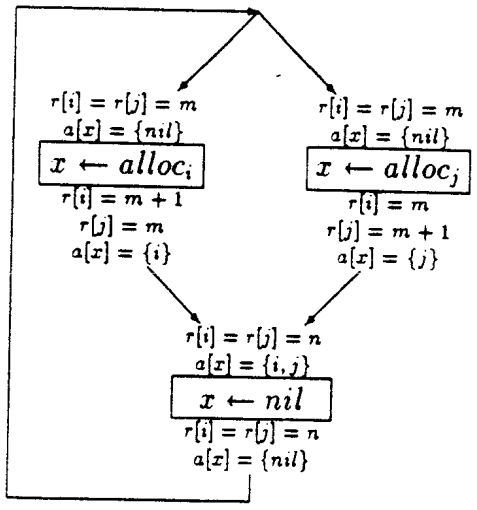


Figure 3.4 Sketch of how $\mathcal{F}_R$ can produce excessive inferred reference counts. (Missing reference count components are 0.)

3.5.1 The Consequences of Collapsing Many Objects into One

It may not be immediately clear why $\langle x \leftarrow alloc_i \rangle_R$ should set $r[i]$ to $(r[i] \oplus 1)$. Because of the abstraction which collapses all run time objects created by a given allocator into one, it is clearly not correct to set the reference count of i at $x \leftarrow alloc_i$ to 1. This would disregard any references to objects previously allocated here. However it is tempting to set $r[i]$ to $max(r[i], 1)$, arguing that the allocation makes no new references to old run time objects. This would be wrong. Consider the program of figure 3.5. The fixed point solution that would result from setting $r[i]$ to $max(r[i], 1)$ at the allocation site is shown. (Remember that *kill* first decrements $r[i]$ from its value of 2, since the register being killed (x) must refer to an object represented by i.) After the $x \leftarrow nil$ instruction y may refer to an object allocated at the allocation site. Therefore $r[i]$ should not be 0 at that point.

The problem is that in arguing that an allocation makes no new references to *old* objects we are differentiating objects which were collapsed into one. Decisions were

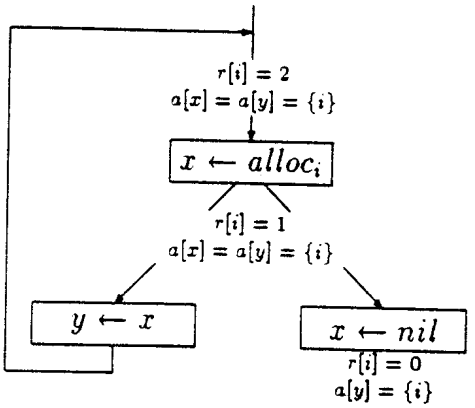


Figure 3.5 Annotated using incorrect definition for $\langle x \leftarrow alloc_i \rangle_R$. Since y may refer to an object allocated by $alloc_i$, $r[i] = 0$ at the end is wrong.

made based on the assumption that all run time objects created by a single allocator are represented by one object. They must not be differentiated. The meaning of $r[i]$ is that there are no more than $r[i]$ references to *any and all* objects allocated by $alloc_i$. It is an upper bound on the *sum* of the number of references to each object allocated by $alloc_i$. The allocation instruction certainly makes a new reference to an object created there, and thus increments this sum.

Returning to figure 3.5, consider what happens when the correct function is applied at $x \leftarrow alloc_i$. The value of $r[i]$ is 2 everywhere, representing the possible references from x and from y. Just before the $x \leftarrow nil$ instruction these refer to different run time objects. The referent of x does become inaccessible here but our analysis cannot determine this, as a direct consequence of the approximation of objects.

Of course different abstractions are possible, and could be investigated if this one proves to be a hindrance. For example, a first improvement might be to associate two locations, one old and one new, with each allocation site. The functions would become correspondingly more complicated. We have not pursued this approach.

3.6 Monotonicity and Distributivity

We have formulated reference count analysis in two separate phases, the first of which computes the sets of objects to which registers may refer, and the second of which uses this information to compute upper bounds on reference counts for objects. Let us denote the MOP solution to a problem specified by some framework $\mathcal{F}_L$ on an instance I by $\mathcal{F}_L(I)$. Then the reference count analysis, which uses the $\mathcal{F}_A$ solution, might be denoted $\mathcal{F}_R(I, \mathcal{F}_A(I))$. Since we need to see the solution of both frameworks

(for the deallocation criterion) we might denote the result of the entire algorithm by the pair $\langle \mathcal{F}_R(I, \mathcal{F}_A(I)), \mathcal{F}_A(I) \rangle$.

The algorithm could instead be presented in a single framework $\mathcal{F}_{R \times A}$, that combines the information gathering process into one. The lattice for this framework is $R \times A$. The meet operation is obvious :

$$\langle r, a \rangle \wedge \langle r', a' \rangle = \langle r \wedge r', a \wedge a' \rangle.$$

This induces definitions for bottom, top, and the approximation ordering. The transfer functions are easily derived from those for the separate frameworks. As an example:

$\langle x \leftarrow alloc_i \rangle_{R \times A}(\langle r, a \rangle) = \langle r'', a' \rangle$ where

$$a' = \{i\} \text{ and } r' = kill(x, r, a) \text{ and } r''[i] = r'[i] \oplus 1$$

The $\mathcal{F}_{R \times A}$ formulation is *not* equivalent to the original. It is more accurate. That is $\mathcal{F}_{R \times A}(I) \geq \langle \mathcal{F}_R(I, \mathcal{F}_A(I)), \mathcal{F}_A(I) \rangle$, and the inequality is strict for some instances. The difference is in the A-values used to compute the R-values. For $\mathcal{F}_{R \times A}$, the MOP solution at the point p in figure 3.6, is the R component of $f_{R \times A}(\langle r, a \rangle) \wedge f_{R \times A}(\langle r', a' \rangle)$ whereas for the original framework the MOP solution at p is $f_R(r, a \wedge a') \wedge f_R(r', a \wedge a')$. This latter value may be less precise if $a \wedge a'$ is less precise than a or a', as it will frequently be. For the same reason, $\mathcal{F}_{R \times A}$ is *not distributive*. Figure 3.7 shows a specific example of the non-distributive behavior. The solid lines and boxes represent

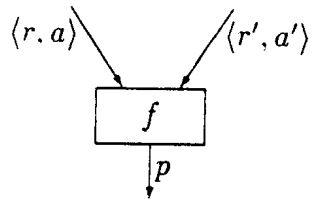


Figure 3.6

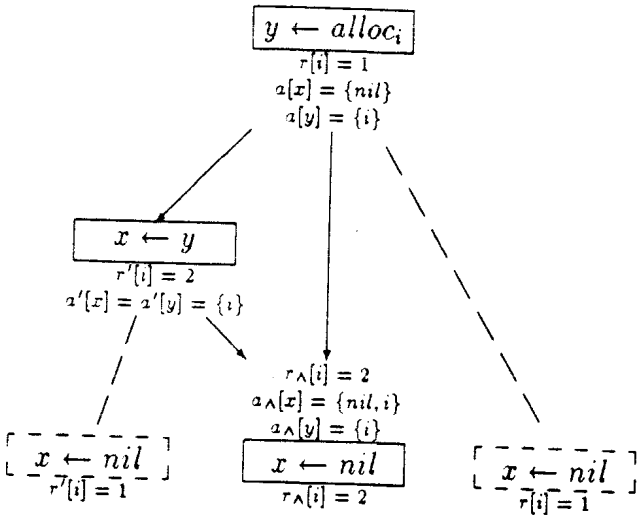


Figure 3.7 Non-Distributivity of $\mathcal{F}_{R \times A}$

the calculation of $\langle x \leftarrow nil \rangle_{R \times A}(r' \wedge r, a' \wedge a)$. It's i component is 2. The dotted lines and boxes indicate computation of $\langle x \leftarrow nil \rangle_{R \times A}(r', a')$ and $\langle x \leftarrow nil \rangle_{R \times A}(r, a)$. Both produce 1 as the i component, so that the i component of their meet, $1 \wedge 1$, is 1. Since $1 \neq 2$ in R, $\langle x \leftarrow nil \rangle_{R \times A}$ is not distributive.[11] As a result, Kildall's algorithm does not compute the MOP solution. So, although the simultaneous equations specified by the combined framework are more precise than those for the separated frameworks, the precise solution is not obtained with Kildall's algorithm. In fact it seems that the solution obtained for $\mathcal{F}_{R \times A}$ is equivalent to the MOP solution to the less precise separated frameworks formulation, $\mathcal{F}_R(I, \mathcal{F}_A(I))$. The improved algorithm presented in [KU77] would recover some of the precision of $\mathcal{F}_{R \times A}$. But it is more expensive.

[11]This example does exhibit monotonicity since 1 is greater than 2 in the ordering on R.

3.7 A Better Analysis

Recall that in the deallocation criterion for reference count analysis the final "or $n(i,a) = 1$" clause is needed because $\mathcal{F}_R$ allows inferred reference counts to become unnecessarily high. Unfortunately these "elevated" counts can hinder the data flow analysis so that it is not sufficient to only worry about them after the data flow algorithm has terminated. The benefit of reference count analysis is in being able to infer an $r[i]$ which is less than $n(i,a)$. If an elevated reference count hinders such an inference the "or $n(i,a) = 1$" clause cannot help out. Figure 3.8 outlines an example of this problem. It is a modification of figure 3.3. The extra instructions on the right hand branch should not have any effect on the deallocation of x and y's referents later. However, they do result in "elevated" counts for i and j. Deallocation in figure 3.3

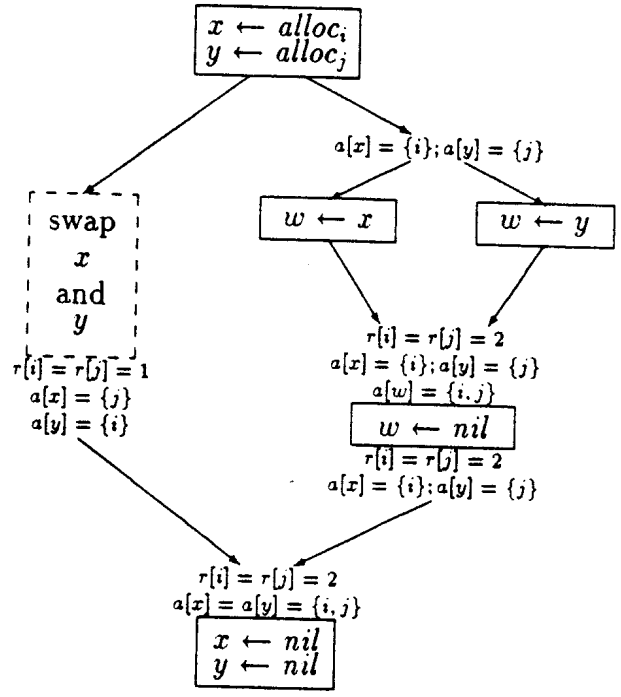


Figure 3.8 Elevated Reference Counts Hinder the Analysis and Prevent Deallocation.

depended on the inferred reference counts being less than $n(i,a)$ and $n(j,a)$. In this example they are not, as a direct result of the elevated counts. Consequently, x's referent cannot be deallocated.

Eliminating elevated reference counts is not difficult. We create a new framework, $\mathcal{F}_{R'}$, which is modification of $\mathcal{F}_R$. At all points p, this framework will maintain $r_p[i] \leq n(i, a_p)$. Since the anomaly only arises from the handling of registers being killed, the only change is to the auxiliary *kill* function:

$$kill'(x, r, a) = r' \text{ where } r' = r \text{ except}$$

$$\begin{cases} r'[j] = r[j] \ominus 1 & a[x] = \{j\} \text{ (a singleton)} \\ \forall\, i \in a[x], r'[i] = \min \begin{cases} r[i] \\ n(i,a) - 1 \end{cases} & \text{otherwise} \end{cases}$$

Note that a is the A-value before the killing instruction. Therefore the number of occurrences of i in the sets of the A-value after the instruction will be $n(i,a) - 1$ if $i \in a[x]$. If $i \notin a[x]$ the membership of i in sets of a is unchanged at the instruction, so it is not necessary to change $r[i]$.

Our implementation uses this framework.

3.8 Implementation Details

3.8.1 Handling Function Calls

If there are function[12] calls in a program their possible effect on the accessibility and reference counts of objects must be taken into consideration. In the absence of interprocedural analysis (which is the subject of the following chapter), we are

[12]In Russell, all "subroutines" are called functions. All return some value (though it may be a "void" value). Russell "functions" may, and frequently do have side-effects, and are therefore not functions in the mathematical sense. We stick with the well-established terms "intraprocedural" and "interprocedural".

forced to make worst case assumptions. In general a function call may return a new object, one of its arguments, or a *nil* value. It may also store its arguments to memory. It is safe to ignore any new object created by a call, by treating it as a value for which deallocation makes no sense. If it is assumed that all objects which are passed to a function are stored to memory, their reference counts are set to ∞. It is then safe to ignore the fact that one of them may be returned by the function. Thus we can assume that a function call returns a *nil* value. In Russell intermediate code, a function call looks like $r \leftarrow f[a_1, \ldots, a_n]$ where r, $a_1, \ldots, a_n$ are registers. In the absence of interprocedural information, it is treated as the following sequence of instructions : $mem \leftarrow a_1, \ldots, mem \leftarrow a_n, r \leftarrow nil$.

Calls to functions which may save a continuation wreak havoc on the analysis: every accessible object could be stored to memory if a continuation is saved. We depend on the compiler's front end to indicate, where possible, that a called function does not save a continuation.

The front end also helps greatly by providing information about the effect of the many calls to built-in functions. Where possible it indicates that arguments are not saved (stored in memory) by the called function. They can then be completely ignored. It indicates when a built-in function returns a specific parameter without saving references to it. This is typical of printing functions, whose parameter is passed back as the result. The call to these functions can be treated as an assignment ro r of the specified argument. The front end also indicates when a function call can be treated as an allocator. For example, most of the arithmetic operations on bignums are implemented by calls to built-in functions. These generally return the only reference to a newly created object, and do not save any of their arguments. Such functions produce many of the small temporary objects for which this compile-

time garbage collection was devised. By treating the call as an allocator the object it returns is included in the analysis and it might be found to be deallocatable.

3.8.2 Deallocation of Objects Whose Size is Known

In many cases the size of allocated objects is known at compile time. Explicit deallocation of an object of known size is very inexpensive. The algorithm therefore maintains a table of sizes of objects. If all the objects to which a register might refer when it is to be deallocated are of one size, then the fast deallocation sequence is used.

3.8.3 "Deallocating" Non-heap Objects

Russell's garbage collector provides a deallocation routine which recognizes and behaves intelligently when it is passed a pointer to non-heap memory, for example to the stack or static areas. Therefore, while it does not make "sense" to deallocate static and stack objects, it is safe to pass them to the deallocation routine. Thus if a register which is being killed may refer to stack or static objects as well as otherwise-inaccessible objects, explicit deallocation is possible. Therefore stack/static object values are separated out from *nil*, giving an extra instruction type: $x \leftarrow \mathit{stack/static}$, and O is augmented with s, an object to represent these values. With appropriate changes to the transfer functions and the deallocation criterion, some extra opportunities for deallocation might be uncovered.

3.9 Discussion

We have compared accessibility and reference count analysis and shown that reference count analysis can provide more precise information and allow more compile time garbage collection. Accessibility is effectively what Chase or Ruggieri would compute if they ignored "containment".[13] By giving up on nested pointers, we avoided the complexity of previous analyses. This allowed us to implement a usable algorithm and to investigate the issues and advantages of inferred reference counts. We discovered that they result in greater precision in some cases, allowing deallocation where accessibility analysis can not. There is no reason why our analysis could not be fully extended using techniques from previous analyses. Reference count analysis depends on an accessibility information. In our algorithm it is provided by $\mathcal{F}_A$. Chase and Ruggieri provide much more precise information about accessibility. By building reference counting on top of their analysis, the benefits of both schemes could be obtained. However, the resulting algorithms might be very expensive.

Our optimization differs from many data flow analysis-based optimizations in that it does not use information about "future behavior". This ability to use "future behavior" is one of the principle sources of other compile time optimization. In deciding that an object may be allocated on the stack or that storage may be overwritten (destructive update), Chase uses information about what may happen to the object or storage after the allocation or update. Ruggieri decides where to allocate heap objects based on how long they will live. Barth removes reference count transactions if he is sure there will be a canceling transaction. None of these optimizations could be replaced by run time decisions. In contrast, our optimization simply shifts a run time

[13]We do not wish to make little of containment analysis. It is at least as deep a subject as static reference counting.

operation to compile time. During execution, no object will be freed with explicit deallocation code, which could not at that same point be collected by the garbage collector. Insertion of deallocation code is an optimization only because it shifts garbage collection overhead to compile time. An optimization which sought to explicitly deallocate objects when they will not be referenced again (based on live-variable analysis) would use information about the future. Our implementation does use information from a limited live-variable analysis performed in an earlier pass of the compiler.

Chapter 4

Interprocedural Analysis

For reference count analysis to be effective for real programs, it is necessary to make better than worst-case assumptions at call sites. This requires analysis of the effect that function calls may have on references. A function may return a newly allocated object, or an object passed to it as a parameter,[1] or a non-heap object value. It may also change the accessibility and reference counts of any objects passed to it.

In Russell intermediate code, a function call looks like $r \leftarrow f[a_1, \ldots, a_n]$ where r is a register, f denotes a function, and $a_1, \ldots, a_n$ are registers. Russell functions are first class objects and Russell strongly encourages full use of this feature. Thus f may be a function-valued variable whose value is unknown at compile time, which would prevent us from building a complete call graph. The front end makes a fair attempt at computing the identity of the called function, but is hindered by the separate compilation facility of Russell. We make no attempt to analyze such calls but treat them as described in subsection 3.8.1. In what follows we assume that f is a known function which is defined in the same file as the call. We also assume that the front end has indicated that the call will not save a continuation, even indirectly.

Each function definition ends with a return instruction *return* x, where x is a register. A function's virtual registers are private to it. Thus a function can only

[1] I have tried to reserve "parameter" for the formal parameters of a function (i.e. on the called side), and to use "argument" for the values on the calling side. However the distinction is not always clear.

access objects through memory and its parameters. The only changes that can be seen after a function returns are in the returned value and in memory.

Our interprocedural data flow analysis algorithm is different from standard ones in that it does not use the program call graph as the flow graph. The graph used here is specially designed for this problem. Callahan's program summary graph [Cal88] is an earlier example of this move away from call graphs.

The problem only requires "bottom-up" analysis. That is, the information to be computed for a function f depends on functions it calls, but not on the context of calls to it. Context ("top-down") information does not add to the precision of the analysis. The only information which context can give is of the flavor "every time f is called its ith parameter may refer to any of the following objects, and the inferred reference count of these objects, at all calls, is bounded by ...". As already stated the overall (relevant) effect of a function call on the arguments can only be to copy them to memory, or return one of them. Other copies which the function makes disappear when the function terminates. The function certainly cannot destroy any references in a virtual register of the calling function, since it does not have access to them. In any case, separate compilation makes top-down analysis impossible, since it prevents us from identifying all calls to f at the time when f is being analyzed (c.f. section 4.4).

4.1 What We Need to Know About Called Functions

In the absence of interprocedural information we must assume that every argument in a call may be stored to memory by the called function. The passing of an argument,

a_i, is treated as $mem \leftarrow a_i$. This is seriously restrictive since it prevents any object which may be passed to a function from ever being deallocated. If a function does not store its ith parameter to memory nor return it as the result object then the corresponding argument in calls to that function can be ignored : the call has no permanent effect on the accessibility of any object passed to the function as one of these arguments. As already noted a function cannot retain any references to its parameters other than in memory, since its virtual registers disappear at the function return.

The other source of imprecision at a call site is not knowing what kind of object is returned by the called function. Without interprocedural information the call is treated as $r \leftarrow nil$. (Recall that *nil* represents all values which cannot be considered for deallocation). If one of the values which a function may return is a value represented by *nil* then this is the best we can hope for. But if it can be ascertained that the function definitely returns a newly allocated object,[2] and does not store any references to that object, then the call should be treated as an allocator. If the function definitely returns its ith parameter without storing any references to it, the call is effectively an assignment: $r \leftarrow a_i$. (The argument can then be ignored instead of being treated as $mem \leftarrow a_i$.) If stack and static objects are distinguished from *nil* (as described in subsection 3.8.3), and if the result of the called function is definitely one of these objects then the call can be treated as $r \leftarrow stack/static$.

The treatment just described does not cover all possibilities. It may only be possible to ascertain that a function returns some combination of the values described. For example it may return a static or stack object or a newly allocated object, in which

[2]By "newly allocated object" we mean a heap object which did not exist before the call. It is an object which is allocated by the called function, or by a function it calls.

case it can be treated as an allocator. Other combinations cannot be handled with the basic instruction types. Our implementation gives up, but it would not be difficult to extend the set of instructions to include such things as $r \leftarrow \{a_1, a_3, alloc_i\}$ (meaning that r may be assigned a reference to any of the objects to which a_1 and a_3 may refer or to a new object).

The purpose of the interprocedural analysis will be to compute the following *summary information* for each (n-ary) function f

1. the set $result[f] \subset \{newobj, nil, static/stack, param_1, \ldots, param_n\}$, which represents the smallest set of values that f may return;

2. for each of the n parameters, two boolean values, $stored_i[f]$, and $returned_i[f]$, which denote whether f *may* store and/or return its ith parameter.

The information is computed in two phases. Each function is first analyzed intraprocedurally, and its dependence on other functions is established. Then the interprocedural phase analyzes the dependencies of all the functions, and produces the summary information. The information is used to decide how to handle function calls during the intraprocedural reference count analysis, as outlined earlier in this section.

4.2 Flow-Sensitivity and Aliasing

The summary information is "flow-sensitive". That is the flow of control within a function must be taken into account in order to compute its summary. This is because of the "propagation" nature of the data: reference values are propagated from parameters by instructions within a procedure. Computation of summary information requires an interprocedural accessibility analysis of each function.

In general, flow-sensitive problems are intractable in the presence of aliasing [Mye81]: intra and interprocedural information are not independent and must be computed together to a fixed point. However, since our analysis ignores the context of calls, aliases are irrelevant, and the interprocedural problem can be solved independently.

4.3 Computation of Summary Information

Computation of interprocedural summary information is a data flow problem since one function can affect the summary of another. For example, if f returns the result of a call to g, then $result[f]$ depends on $result[g]$. If f's ith parameter is passed as the jth argument to g, g might store the value to memory, so that $stored_i[f]$ depends on $stored_j[g]$.

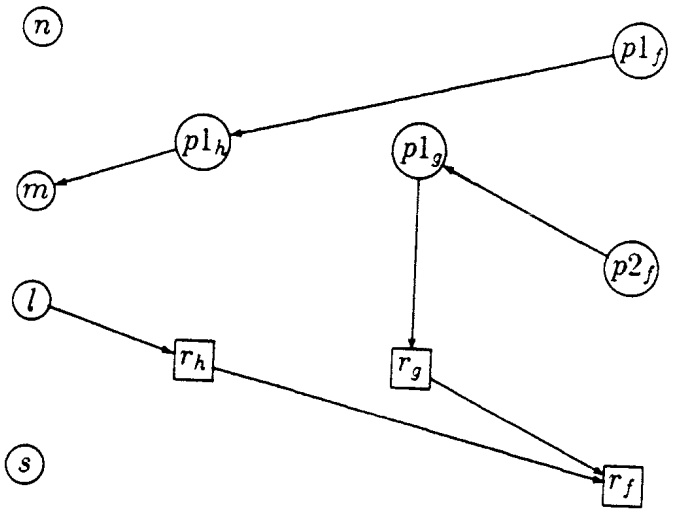
4.3.1 Flow Summary Graph

These dependencies are represented by a "flow summary graph". Paths in the graph express how parameters and new objects can be transmitted through function calls and reach results or be stored in memory. The set of graph nodes consists of nodes representing each of *nil* (node n), *static/stack* (node s), *mem* (node m), "new object" (node l ("l" for location)),[3] and for each n-ary function f, a result node, r_f, and parameter nodes, $p1_f, \ldots, pn_f$. The nodes n, s and l are sources of edges only; m is only a sink.

[3]In order to keep track of the size of heap objects, the flow summary graph should have an l node for each different size of object. Then if a function f may return a new object, its size could be included with the $result[f]$ summary information.

Edges in the flow summary graph fall into two general categories: *result edges*, whose destination is a result node, and *parameter edges* whose source is a parameter node. *Result edges* originate at n, s, l, a result node, or a parameter node corresponding to the same function as the result node. *Parameter edges* end at m or at a parameter node for a called function. Edges from a parameter node to a result node are arbitrarily considered as result edges. A result edge from l to the result node for f means that f may return an object it allocates *but does not store.* Similarly result edges from parameter and result nodes indicate that f may return the corresponding value, but does not store it to memory. Parameter edges indicate when a parameter may be stored to memory, or passed to another function.

Figure 4.1 gives a program and the corresponding flow summary graph. Since f may return the result of calling either g or h, there are two edges incident on r_f. The other edges should be self-explanatory.




```
function f[p1, p2]
    if — then
        r ← g[p2]
    else
        r ← h[p1]
    return r

function g[p1]
    return p1

function h[p1]
    mem ← p1
    r ← alloc
    return r
```

Figure 4.1 Example of a Flow Summary Graph

Interprocedural information is derived from the flow summary graph. The relevant questions translate to path problems on the graph. For example, in figure 4.1, the existence of a path from $p1_f$ to m indicates that f may store its first parameter. The path from $p2_f$ to r_f indicates that f may return its second argument. f may also return a new object. Since there is only one path to r_h, from l, h *must* return a newly allocated object.

4.3.2 Derivation of the Graph

The flow summary graph is created by analyzing each function definition in turn and adding the appropriate edges. The analysis basically uses the accessibility framework, $\mathcal{F}_A$, of section 3.4, with modifications in the set of objects and the treatment of some instructions. When analyzing function f, the set O of objects is augmented with "parameter objects" $o_1, \ldots, o_n$ for each of f's parameters, and with "result objects" c_i for the result of each call site c_i in f. These correspond to the "formal sources" and "unresolved sources" of [Rug87]. Parameters must be treated as sources of objects if their propagation is to be traced. Similarly for function results. The result object c_i acts as a place-holder for any objects which the function called at site c_i may return.

A function's parameters are registers, $p_1, \ldots, p_n$. Previously these registers were assumed to initially contain a *nil* value. But here the start node in the flow graph for f is assumed to initialize them to refer to the corresponding parameter object, with the sequence of instructions $p_1 \leftarrow o_1, \ \ldots, \ p_n \leftarrow o_n$. Calls in f are treated as allocations: $r \leftarrow alloc_{c_i}$. The passing of arguments is ignored during the data flow process.

As before, the solution which $\mathcal{F}_A$ computes at each point, associates with each register the set of objects to which that register may refer at that point. The sets

associated with arguments at call sites, and with *mem* at the end of the function, determine what parameter edges should be added to the flow summary graph on behalf of the analyzed function. Every parameter object appearing in one of these sets causes a parameter edge to be added to the graph. Result edges are determined by the set associated with the returned register, in conjunction with the set associated with *mem* at the end of the function. Allocated objects, parameter objects and result objects in the returned register's set add edges only if the object does not appear in the set associated with *mem*. More formally, given a solution a for a function f, the following process computes the edges which should be added to the flow summary graph: (*end* is the point in the flow graph for f at which the *return* instruction appears.)

- At each call site c_i, let g be the called function and $x_1, \ldots, x_n$ be the arguments. Then $a_{c_i}[x_j]$ is the set of objects which may be passed as the jth argument to g.

  For all parameter objects o_k (representing parameters of f) in $a_{c_i}[x_j]$, add a parameter edge from pk_f to pj_g.

- For all parameter objects o_l in $a_{end}[mem]$, add a parameter edge from pl_f to m.

- If x is the register returned at the end of f, $a_{end}[x]$ is the set of objects which may be returned. The edges added here are result edges.

  If $nil \in a_{end}[x]$, add an edge from n to r_f.

  If $stack/static \in a_{end}[x]$, add an edge from s to r_f.

  For all other objects $i \in a_{end}[x]$ such that $i \notin a_{end}[mem]$

  - if i is parameter object o_k, add an edge from pk_f to r_f

- if i is result object c_i resulting from a call to g at call site c_i, add an edge from r_g to r_f

- if i is an ordinary allocated object add an edge from l to r_f

A flow summary graph may be cyclic if the program it summarizes is recursive. Since there are no edges from a result node to a parameter node, cycles will consist exclusively of parameter nodes or exclusively of result nodes. Parameter node cycles are produced by a function which calls itself recursively (directly or indirectly) with some ith parameter in the ith argument position. (For example the exponentiation function: $f(m,n) = \ if\ (n = 0)\ then\ 1\ else\ m * f(m, n-1)$ produces a cycle from $p1_f$ to itself). Cycles of result nodes are generated by tail recursive functions. Note that not all recursive programs will have cyclic flow summary graphs. For example, consider a recursively defined factorial function (which returns a bignum, (i.e. a heap object)):

$$f(n) = \ if\ n = 0\ then\ 1\ else\ n * f(n-1)$$

Figure 4.2 is the flow summary graph which would be produced if $*$ and $-$ were

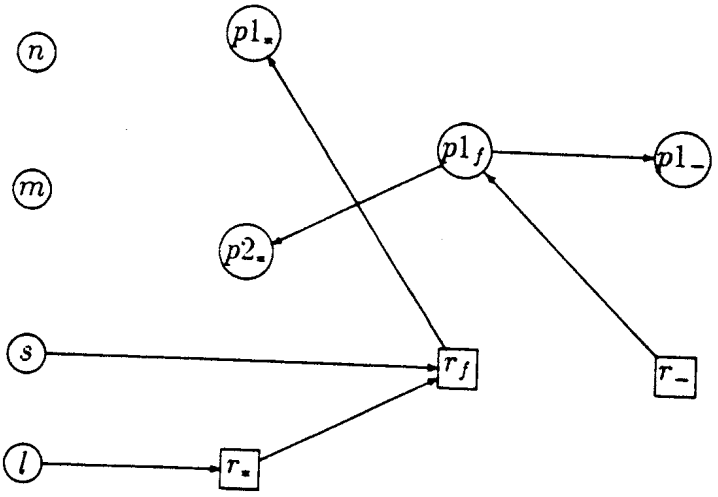


Figure 4.2 Flow Summary Graph for the Factorial Function.

not built-in functions.[4] f returns either a static object (1) or the object allocated by the bignum multiplication function. f's argument is passed to the multiplication and subtraction functions, but not to the recursive call. Thus no graph cycles are produced.

4.3.3 Summary Information from the Graph

A flow summary graph provides all the information needed to compute summary information for all defined functions. The translation from paths in the graph to summary information is as follows:

- Function f may store its ith parameter to memory ($stored_i[f]$ is true) if and only if m can be reached from pi_f.
- Function f may return its ith parameter ($returned_i[f]$ is true) only if r_f can be reached from pi_f.
- The subset of the nodes $\{l, n, s, p1_f, \ldots, pn_f\}$ from which there is a path to r_f corresponds to $result[f]$.

Some paths in the graph do not reflect possible flow through calls. This can produce overly conservative information.

The summary information is produced from the graph with two iterative processes. One propagates values back from m toward parameter nodes. The other propagates values forward toward result nodes from all other nodes. These processes are formulated as distributive data flow analysis frameworks.

[4] In fact, integer subtraction produces inline code. Also the front-end indicates that the call to bignum multiplication is an allocation site, and that the function does not save any references to the arguments.

4.4 Separate Compilation

Russell's separate compilation facility allows a function defined in one file to be called in another, but it forces the file containing the definition to be compiled before that containing the call. Optimization information about functions defined in a source file is appended to the object file, and can be used in compiling source code that contains calls to those functions. The interprocedural summary information described in this chapter becomes part of the appended optimization information. Before reference count analysis is performed on a function definition, the summary information for external functions it calls is read and used to decide how to treat the calls.

4.5 Discussion

Full interprocedural reference count analysis has three phases. The first is intraprocedural accessibility analysis of each function. Its purpose is to compute the flow of parameters to calls and to memory, and of objects to the returned register. The flow summary graph uses this information to encode the dependence of the summary information for a function on functions it calls.

The second, interprocedural phase computes summary information by solving path problems on the flow summary graph. Summary information is used to decide how to handle calls in the final intraprocedural phase, which is the full-blown reference-count analysis, as described in chapter 3.

By comparison Ruggieri's algorithm (c.f. chapter 2.2) has only two phases. They correspond to the first two phases of our algorithm. For her analysis, per-function "summary" information is the goal: "with which function should each allocator be

associated ?". Our goal is to identify instructions which create garbage. This involves propagating information intraprocedurally to every program point.

Chapter 5

Results

The algorithm described in the previous two chapters has been implemented as an optional pass of the Russell compiler. Table 5.1 presents timing measurements for a variety of programs. The first column gives execution times when the default optimizations are used. These include optimizations to put activation records and variable cells on the stack where possible [BH88], and some inline expansion of functions. The second column adds the compile time garbage collection optimization described in this dissertation. All programs were run in a 576k heap, which was not expanded during execution. Times are given in seconds on a Sun3/60. The number of garbage collections is given in parentheses. The code generator uses a rather primitive register allocator.

	without reference analysis	with reference analysis
fact1	14.1 (8)	13.8 (0)
fact2	20.0 (15)	13.8 (0)
rational	73.3 (34)	71 (20)
divide	37.5 (36)	32.4 (24)
reals	53.0 (41)	35.1 (13)
exact	37.4 (26)	32.7 (20)

Table 5.1 Timing Comparison. Times in seconds on a Sun3/60 with a 576k heap. Number of garbage collections is given in parentheses.

The first two programs compute the exact value (using bignums) of 100!, one thousand times, using a straightforward recursive program. The reference count analysis allows all bignum values to be explicitly deallocated, so that no garbage collections are required. In the first of these programs, "fact1", the results are not very impressive because each garbage collection takes negligible time : eight of them take about 0.3secs. This is because there are very few accessible heap objects to be marked by that phase of the collector. As pointed out earlier, insertion of deallocation code primarily shifts overhead of the *mark* phase to compile time. The work of freeing inaccessible storage, which the sweep phase normally does, still gets done during execution time, though it is spread over the execution. Preliminary timings for this example showed that garbage collection was faster than explicit deallocation. We therefore hand coded the (short) most frequently executed path through the deallocation routine, and succeeded in making it a little faster than garbage collection.

The second program preallocates 240k bytes of memory (half the heap), which remains accessible and must be traversed by the marking phase. The 15 traversals of the 240k account for the 5.8 secs difference between unoptimized times for "fact1" and "fact2". With explicit deallocation no garbage collection is needed, and costs of repeatedly marking the preallocated data are avoided.[1]

The third example, "rational", solves a randomly generated 35 by 35 system of linear equations with 0-1 coefficients using Gaussian elimination with exact rational arithmetic. The fourth example, "divide", performs 100 divisions of large bignums (1000! / 900!). As for the second program, 240k bytes of memory are preallocated. "reals", solves an 8 by 8 system of equations using Gaussian elimination with construc-

[1]This situation emphasizes the motivation for generation scavenging. Garbage collection of the preallocated data is a fruitless exercise, since it is always accessible. A generation scavenger would not bother to traverse it after the first few attempts.

tive real arithmetic. In these examples, explicit deallocation (primarily of bignums) results in a noticeable performance improvement.

We observed that in some cases certain important loop variables cannot be explicitly deallocated because on the first iteration they may refer to the same object as the variable used to initialize them (see figure 5.1). This difficulty could be overcome by unrolling the loop once. Alternatively the loop variable (x in the example) could be initialized with a copy of the initial value in a new object, thereby reducing reference counts.

Other problems arise if all registers are not explicitly initialized (assigned a static object) since it must be assumed that if a register is not initialized on some path it may reference something which is not deallocatable.

The final example performs 20 multiplications on constructive real numbers using an inefficient stream representation of the reals [BCOR86]. The benefit here is gained from including "single reference" tags in stream cons nodes (see the last paragraph of section 1.3).

For these experiments heap size was kept constant, and the effect of our optimization on execution time was measured. If, instead the heap size is allowed to grow on demand during execution, the unoptimized version of some of the (larger)

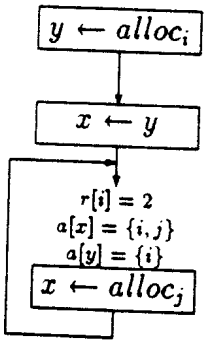


Figure 5.1 In the loop x's referent cannot be deallocated.

programs run as fast as the optimized ones, but at the expense of heap consumption. For example, the unoptimized "reals" takes 36.2 seconds when heap expansion is enabled.

Chapter 6

Conclusion

The objective of this thesis was to show that reference count analysis can improve the precision of previous storage analyses, and to demonstrate the feasibility of using this analysis to insert explicit deallocation code into Russell programs, and thereby reduce run time garbage collection costs. By ignoring the complexity of data structures and nested pointers we were able to investigate the specific advantages of static reference counting.

We are unaware of any other implementation of static reference counting. In fact, many of the storage optimization techniques which have been proposed have not been implemented, or have been in a haphazard way. Our algorithm was fairly easy to implement. While this work concentrated on Russell, the analysis is performed on three-address intermediate code and is largely language independent.

The research began as a class assignment to apply Hudak's semantic abstraction of reference counting [Hud86] to the problem of the many short-lived temporaries eating up heap space in (numerical) Russell programs. Since Russell's storage allocator maintains a free list of storage cells, it is feasible to return inaccessible objects to it.

6.1 Summary and Contributions

In chapter 3 we presented the reference count analysis algorithm and its use in inserting explicit deallocation code. We began with accessibility analysis and showed why

this is not sufficiently precise. Reference counting improves the accessibility analysis by allowing us to infer that, while an object may be referred to by any of n registers, at most m of them may actually refer to it. If m is less than n then reference count analysis is more precise.

Previous work on storage analysis has developed in other directions. In chapter 2 some of these were outlined. An overview of optimizations which reduce storage management costs was given. Two comprehensive storage analysis schemes were presented. It is hoped that as well as providing a basis for evaluation of our work, the presentation will be useful to anyone interested in understanding previous work in this area. The final section of chapter 2 presented an algorithm aimed at shifting garbage collection overhead to compile time. The resulting optimizations are related to my work.

In chapter 3 we noted an interesting property of "combining" distributive data flow analysis frameworks. The resulting framework is not necessarily distributive. This was a little surprising to us at first. Our initial algorithm used the combined framework and we had assumed it was distributive. While this is not a major discovery, it has not been mentioned before.

Chapter 4 presents an interprocedural algorithm which computes the summary information about functions that is relevant to reference count analysis. The algorithm uses a specially designed graph. Computation of summary information is reduced to solving path problems on this graph.

The resulting algorithm has been implemented as an optional pass of the Russell compiler. Some timing results are presented in chapter 5. The algorithm is quite effective at uncovering inaccessible objects, and results in noticeable speed-ups, particularly in some of the Russell "bignum" benchmark programs.

6.2 Further Work

Now that we have shown the advantages of static reference counting, it would be interesting to combine it with one of the earlier storage analysis schemes to come up with an analysis that incorporates the benefits of both. There appears to be no reason why they can not be combined. Specifically, I would work from Ruggieri's lifetime analysis, taking the results of her interprocedural phase, add another intraprocedural phase (corresponding to $\mathcal{F}_A$), and then put reference counts on top.

As for optimization of Russell storage management, the approach described here does not solve all the problems. It is not yet clear whether insertion of explicit deallocation code is the ideal way to handle the problem of floating point and bignum temporaries. For some of them, particularly floating point ones, stack allocation might be feasible. But reference count analysis provided an easy and effective way to deal with the problem for now.

An intermediately difficult extension has been suggested to me [Com88]. It is reminiscent of Barth's optimizations. If our reference count analysis establishes that an object can be deallocated in the same basic block in which it is allocated, then the object could be stack allocated. This is correct since in a basic block there is a single thread of control, and everything concerning such an object takes place on the path from the allocation to the deallocation point. Many expression temporaries can be expected to fall into this category. The optimization, like all transformations from heap to stack allocation, would improve storage usage, and cut out heap allocation and deallocation time.

Bibliography

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*, chapter 3. McGraw-Hill, 1985.

[ASU86] A.V. Aho, Ravi Sethi, and J.D. Ullman. *Compilers Principles, Techniques and Tools*, chapter 10. Addison-Wesley, 1986.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[BCOR86] Hans-J Boehm, Robert Cartwright, Michael J. O'Donnell, and Mark Riggle. Exact real arithmetic: A case study in higher order programming. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming*, pages 162–173, 1986.

[BD86] Hans-J Boehm and Alan Demers. Implementing russell. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 186–195, ACM, July 1986.

[BDD85] Hans-J Boehm, Alan Demers, and James Donahue. *A Programmer's Introduction to Russell.* Technical Report 85–16, Computer Science Department, Rice University, March 1985.

[BH88] Hans-J Boehm and Lucy Hederman. Storage allocation optimizations in a compiler for russell. July 1988. Abstract submitted for POPL 89.

[Boe87] Hans-J Boehm. Constructive real interpretation of numerical programs. In *Proceedings of the ACM SIGPLAN '87 Conference on Interpreters and Interpretive Techniques*, pages 214–221, July 1987.

[BW88] Hans-J Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9), September 88.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, ACM, June 1988.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[Cha87a] David Chase. *Garbage Collection and Other Optimizations.* PhD thesis. Rice University, Houston, Texas, November 1987.

[Cha87b] David R. Chase. Notes for a Colloquium at Rice University, November 1987.

[Coh81] J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[Com88] R. Scott Comer. Personal Communication, September 1988.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[Gop88] K. Gopinath. *Copy Elimination in Single Assignmnet Languages.* PhD thesis, Stanford University, Stanford, California, March 1988.

[HB85] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 300–314, January 1985.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 351–363, ACM, August 1986.

[Ken81] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnik and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Kil73] G.A. Kildall. Global expression optimization during compilation. In *Proceedings of the ACM Principles of Programming Languages*, pages 194–206, October 1973.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data-flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[LH83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, ACM, June 1988.

[MJ76] Steven S. Muchnik and Neil D. Jones. Binding time optimizations in programming languages: Some thoughts toward the design of an ideal language. In *Conference Record of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 77–91, 1976.

[MJ81] S. Muchnik and N. Jones. Flow analysis and optimization of LISP-like structures. In S. Muchnik and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Moo84] David A. Moon. Garbage collection in a large Lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, 1984.

[Mye81] Eugene Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eigth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[RM88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[Rov85] Paul Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Technical Report CSL-84-7, Xerox Palo Alto Research Center, July 1985.

[Rug87] Cristina Ruggieri. *Dynamic Memory Allocation Techniques Based on the Lifetime of Objects.* PhD thesis, Purdue University, West Lafayette, Indiana, August 1987.

[Sch75] Jacob T. Schwartz. Optimization of very high-level languages —I. Value transmission and its corollaries. *Journal of Computer Languages*, 1:161–194, 1975.

[SG77] S.W.Clark and C.C. Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2):78–87, February 1977.

[Ste78] Guy L. Steele Jr. *RABBIT: A Compiler for Scheme.* AI Memo 474, Massachussetts Institute of Technology, May 1978.

[Ung84] David Ungar. Generation scavenging : A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 19 (5)*, pages 157–167, May 1984.