# Garbage Collection and Other Optimizations

David R. Chase

August, 1987; reformatted August 2000

**Abstract**

Existing techniques for garbage collection and machine code optimizations can interfere with each other. The inability to fully optimize code in a garbage-collected system is a hidden cost of garbage collection. One solution to this problem is proposed; an inexpensive protocol that permits most optimizations and garbage collection to coexist.

A second approach to this problem and a separate problem in its own right is to reduce the need for garbage collection. This requires analysis of storage lifetime. Inferring storage lifetime is difficult in a language with nested and recursive data structures, but it is precisely these languages in which garbage collection is most useful. An improved analysis for "storage containment" is described.

Containment information can be represented in a directed graph. The derivation of this graph falls into a monotone data-flow analysis framework; in addition, the derivation has the Church-Rosser property. The graphs produced in the analysis of a value-assignment language have the property that they can be replaced with a single graph without losing any information. These properties also assist in the generation of graphs for side-effect languages.

A different approach to avoiding obtaining memory from the garbage collector is also proposed. Existing techniques are either not general or run the risk of consuming all of a bounded memory. A general, low overhead technique that does not consume excessive amounts of memory is described.

# Contents

# List of Figures

# Chapter 1

# Introduction

Garbage collection and optimizing compilers are both aids to programming productivity. Garbage collection removes some of the need for programmers to account for their use of memory resources. Even if the only measure is lines of code [Bro75], garbage collection helps because the lines of code to manually manage memory are no longer needed. An entire family of proofs is avoided; the programmer need no longer prove that each item freed is not useful to the rest of the program, and the programmer need not prove that every item not freed is useful; the garbage collector automatically ensures that this is true. Information-hiding [Par72] is improved, because a module's specification need not spell out the required allocation and deallocation of parameters and results. Optimizing compilers aid productivity by automating low-level code improvement. This allows programmers to use fewer lines of code and to prove fewer facts about their program. Optimizing compilers make high-level languages more attractive by making their implementations more efficient [Ken81]; use of a higher-level language reduces the lines of a code needed for a given program, thus improving productivity.

Unfortunately, garbage collection and optimizing compilers can interfere with each other. In the worst case, this interference causes code to run incorrectly; in less disastrous cases either many optimizations must be avoided or a less efficient garbage collector must be used. Generally, this interference forces a choice between garbage collection and optimization; a programmer cannot have the best of both worlds.

This dissertation is an investigation of compile-time optimizations applied in a garbage-collected system. I believe that better optimizing compilers can be written for languages in garbage-collected systems. To support

this, I demonstrate ways that current optimization techniques interfere with garbage collection, and describe ways to perform optimizations without disrupting garbage collection. I also describe improvements to existing optimizations that avoid allocation of garbage collected storage and improvements to the analysis needed to implement these optimizations in a compiler.

## 1.1 Intended application

This work is intended to help in the efficient implementation of languages running in a garbage-collected system on stock hardware. The model for compilation and optimization is based on compilers for procedural languages described in Hecht, Aho *et al*, and Kennedy [Hec77, ASU86, Ken81].

This model for compilation, analysis and optimization was chosen because of its widespread use and my familiarity with it. The collector is expected to be of the copying-compacting variety because those appear to have the potential for greatest efficiency. Stock hardware was chosen for several reasons: in the "real world" it is both cheaper and faster; it represents a least common denominator of available computing machines; and it removes the temptation to appeal to a "hardware assist" whenever an especially hard problem appears.

The languages for which this work seems most applicable are strict, statically-typed, and support garbage-collection for some or all objects. Programs written in such languages are more easily analyzed than those written in lazy, weakly-typed languages, and because there is no need for run-time type-checking or scheduling, garbage collection can consume a significant amount of time in compiled programs. Thus, optimizations to avoid the direct and indirect costs of garbage collection are more likely to be profitable.

Unfortunately (for the sake of examples), few if any popular languages fall into this class. Cedar Mesa [Rov85] and Modula-2+ [RLW85, Wir83] have these properties, but they are not especially well-known and I am not that familiar with them. Other features (concurrency and exception-handling) in these languages complicate flow analysis, but are irrelevant to this work. Given this, examples will be written in a mixture of FP [Bac78], Pascal without **free**, and impure Lisp.

## 1.2 Organization

The dissertation is divided into 7 chapters. They contain:

1. This introduction.

2. Background material on garbage collection, data flow analysis, and optimization.

3. A description of unfortunate interactions that might occur between optimized code and a garbage collector, and ways to avoid these interactions. This chapter motivates the phrase "the hidden cost of garbage collection". The hidden cost of garbage collection is the time and memory spent to maintain the run-time environment in a garbage-collectible state. This section also describes ways that some of these interactions can be avoided, though additional analysis in the compiler is required.

4. A review of optimizations designed to cut down on time spent garbage collecting, either by helping the collector or by avoiding the allocation of garbage-collected storage. These optimizations rely on value lifetime analysis which is also described here.

5. A description of a new storage containment analysis, and an investigation of its properties. The new analysis constructs a graph describing containment relationships holding between definitions and allocations within a program. Interesting and useful properties of these graphs appear for both value and side-effect languages.

6. A description of new optimizations that avoid the allocation of garbage-collected storage. These optimizations are based on the principle that optimization should not significantly increase the amount of storage that a program uses. The optimizations are general, do not squander memory, and have low run-time overhead.

7. Conclusions.

# Chapter 2

# Background

This chapter reviews algorithms, analyses, and terminology that will appear later in the dissertation. Review and tutorial information on data-flow analysis and optimizations are widely available [AC72, Ken81, Hec77, ASU86]. The literature on garbage collection has been summarized by Cohen [Coh81] and Nicolau [CN83], though there has been a great deal of recent work [Ung84, BW88, LH83, Rov85, Bro85, Hug85, SCN84, Moo84].

## 2.1 Garbage collection

Garbage collection is an implementation technique for automatically recovering unused memory and reusing it. Its principal advantage is that it is automatic; the programmer does not need to write code to re-use memory. In a local sense this simplifies programming because the code does not need to be written or debugged; in a global sense this simplifies programming because it is no longer necessary to devise and follow cross-module protocols and to account for memory use. The principal disadvantage of garbage collection is that it adds time and space overheads, though for a good garbage collector these can be smaller than the overhead of a bad manual implementation of storage reuse. It is also difficult or impossible to write a garbage collector for some languages because of their cavalier treatment of pointers.

### 2.1.1 Basic algorithms

All garbage collection algorithms attempt to discover objects that are not useful to the further execution of a program and reuse their storage. All algorithms classify an object as garbage when it is no longer *active,* though the algorithms may not reclaim all inactive objects[1]. A small set of objects called *root objects* is active by definition, and other objects are active if they can be reached via a path of pointers (*containment by reference*) or inclusions (*containment by value*). Given an object, a garbage collector must be able to find every pointer contained within it; given a pointer, a garbage collector algorithm must be able to find the object which it identifies.

Good garbage collectors exploit certain expected properties of programs. Typically, young objects are more likely to become inactive than old objects. It also appears that young objects refer to old objects much more often than old objects refer to young objects, and that most objects are referred to by only one object. These properties have been observed experimentally [Cla79, CG77] and inferred from the improved performance of collectors that exploit them [LH83, Moo84, SCN84, DB76, Rov85]. It has also been observed that the sizes of objects are not distributed uniformly [BB77, Nie77].

#### Reference counting

Reference counting garbage collection identifies reusable objects by counting the number of pointers to an object [Col60]. When this count reaches zero, the object is reusable. After an object has been identified as reusable the collector must undo the effects of pointers contained within the object; that is, for each pointer $p$ in the object, the collector must decrement the reference count of the object which $p$ identifies. Every time a pointer variable's value is changed, one object's count must be decreased and the other object's count must be increased.

The simple form of reference counting has several major disadvantages.

1. It cannot reclaim circular containment graphs. All objects in a cycle will have a non-zero reference count, preventing reclamation by the collector, but a cycle need not be reachable from a root object.

[1] I make a distinction between *active* and *useful*; if an object is active, it implements part of a program's state; if an object is useful it not only implements part of the state, but a change (or series of changes) in its contents will affect some later step in the program's execution.

2. The reference counts must be stored somewhere, thus using more memory.

3. Reference counts can grow larger than the maximum value that their field will hold. This requires code to check for overflowed reference counts, and can be a source of other uncollectible objects.

4. The maintenance of correct reference counts at pointer manipulations imposes a time overhead on the execution of ordinary code.

5. Many objects can become unreachable in a single operation; when this happens it can take some time to update all of the reference counts.

These problems can be mitigated in several ways. Circular graphs can be collected with the use of an additional reference count [Bro85] or restricting the structure of the graphs produced [FW79]. The use of hash links [Bob75] to store reference counts [DB76] makes possible the "trick" of not storing any count for the common (reference count equal to one) case and can also handle larger reference counts. References from activation records need not be counted; this avoids much of the expense associated with pointer manipulation, at the cost of less timely collection [DB76, Rov85]. It is possible to defer reference count adjustments; this avoids the need to spend time updating many reference counts when many objects suddenly become free.

Reference counting has its advantages:

1. It is not inherently necessary for a reference counting collector to preempt other computation for an unbounded amount of time; memory can be collected incrementally.

2. Memory is recycled quickly; soon after an object becomes unreachable it can be made available for reuse. This means that a reference counting collector can run well with a high percentage of memory active.

3. Deferred reference counting operations have small critical regions. This can be useful in situations where garbage collection should not interrupt other computation for long periods of time.

In spite of its problems, reference counting has been used to collect garbage in large systems [DB76, Rov85]. These systems employ some of the variations described above to get acceptable performance, though they rely upon some assistance from programmers to break cycles so that they may be collected.

### Mark-and-sweep

The mark-and-sweep garbage collection algorithm is very straightforward. It identifies unusable objects by applying the definition. The marking phase starts from the root objects tracing paths to mark all reachable objects as active. The sweep phase scans all objects, and all those not marked as active are eligible for reuse. Because this process can be time-consuming, garbage collection is typically run when no more memory is available.

This style of collection also has several disadvantages:

1. Collection preempts all other computation, and can take significant amounts of time.

2. Collection can perform very poorly in a virtual memory environment because the marking phase has poor locality of reference and low predictability. The sweep phase has poor locality of reference, but much higher predictability. Overall locality of reference can be low because objects are distributed throughout memory.

3. Mark bits are required to record whether or not an object is active. This takes a small amount of additional memory.

4. The sweep phase, though rapid, takes time proportional to the total amount of memory in the system. Increasing the amount of memory in the system reduces the frequency of garbage collection, but it also increases the cost of each individual collection.

5. Mark-and-sweep collection performs very poorly when the percentage of active memory is high.

The major variation on mark-and-sweep collection is the occasional use of a compacting phase to improve locality of reference. Because older objects are expected to live longer, a single compaction tends to improve both overall locality and marking locality through several garbage collections. Concurrent mark-and-sweep collectors that avoid long preemption have been designed [DLM$^+$78] but they require fine-grained cooperation with the rest of the program.

Mark-and-sweep collection has one major advantage; it can work without much cooperation from the rest of the program [BW88]. It also collects circular graphs without any other restrictions or modifications to the algorithm.

### Copy-and-compact

Copy-and-compact collection collects garbage by making a copy of active storage in a free area and reusing all of the old storage [FY69, Che70]. A collection starts by copying the root objects to the new area. When an object in the old area is copied to the new area, it is overwritten with its new address and a bit is set to indicate that it has been copied. Active objects are discovered by examining pointers included in objects copied to the new area; these pointers will be directed into the old area, identifying candidates for copying. A candidate object is examined to see if has already been copied; if it has, then the pointer to it is updated to reflect the new address. Otherwise, the object is copied and overwritten as described above.

The problems associated with copying-compacting collectors are:

1. Garbage collection preempts other computation.

2. A large portion of memory is always wasted because there must be sufficient memory to hold a copy of the active objects. There must also be enough additional memory that collections do not occur very frequently.

3. The collector must be able to locate exactly the set of active pointers because they must be updated when objects are moved. This makes compacting garbage collection difficult in an uncooperative environment.

4. The collector's locality of reference is not especially good.

There have been a number of variations on copying-compacting collection designed to avoid the need to preempt other computing. These variations are usually incremental, not concurrent, with the collector gaining control at certain events [Bak78, LH83]. This avoids the overhead of explicit synchronization. Other variations attempt to compact and collect small portions of memory at a time; this avoids the need to have large amounts of free memory for the entire set of active objects [Ung84].

Copying-compacting collection has these advantages:

1. Circular structures are collected.

2. Memory is frequently compacted, so the program's locality of reference can be quite good.

3. Because free memory is maintained as one large contiguous area, allocation of new memory can be very rapid.

4. The cost of a single collection is proportional to the number of active objects, not the total size amount of memory. Increasing the amount of memory available decreases collection frequency but does not increase the cost of a collection.

Copying-compacting collectors have been used or proposed for use in several recent systems [Moo84, BS83, Ung84]. All of these systems separate younger objects from older objects, and collect the younger objects much more frequently than the older objects.

### 2.1.2 Important characteristics

All garbage collection algorithms share a few important characteristics. These may seem basic, but they help determine how practical a particular algorithm will be in a given run time system.

#### Per-object information

In all garbage collection systems a certain amount of per-object information must be maintained. This includes information about the size and structure of the object as well as the reference counts and flag bits mentioned above.

This information can be stored in several ways. The most straightforward technique is to include with each object storage to contain this information. A second technique is the use of hash links [Bob75]; an object's address locates its information within a separate table. A third technique is the Big Bag of Pages. Using this technique, a set of pages (a *bag*) is reserved for each object size (or structure, or type). An object's address identifies the bag holding it, and thus identifies its size. The use of bags also permits the implementation of bag-specific tables to contain other information; because all objects in a bag are the same size it is possible to use indexing instead of hashing to locate information. If pointers are tagged, then the structure of an object can be discovered by scanning it for pointers contained within it.

#### Location of pointers

All garbage collection algorithms must be able to locate pointers.

In a reference counting system this is necessary (1) to decrement the reference counts of objects referenced by a reclaimed object and (2) to determine when a reference count adjustment is required. Non-pointer assignments do not cause any reference count adjustments, but pointer assignments do. In a mark-and-sweep system this is necessary for the location of active objects. In a copying-compacting collector this is necessary both for the location of active objects and for the updating of pointers to moved objects.

Note the importance of exact pointer location in reference counting and compacting garbage collectors. For a reference counting collector, treating a pointer as a non-pointer leads to an incorrect reference count; either an object will be recycled while it is still active, or it will never be recycled. If a non-pointer is treated as a pointer, then the garbage collector may treat as a reference count a piece of storage that has some other meaning. In a compacting collector, an ignored pointer is not updated and will continue to point into the old area. If a non-pointer is treated as a pointer, then it will be updated to point to the new location of the "object" which it "identifies".

Pointer location is not so important to mark-and-sweep collection. As long as at least one pointer to an active object is found, the object will not be incorrectly recycled. If a non-pointer is treated as a pointer, then the garbage collector may attempt to set a non-existent mark bit, but there are techniques to avoid this problem. Note especially that mark-and-sweep will not incorrectly modify or re-use memory if a non-pointer that "points" to an actual object is interpreted as a pointer; this may prevent reclamation of the object in one collection, but if the non-pointer's value is changed then the object may be reclaimed in a later collection. Reference counting and compacting collectors do not share this property.

Pointers can be located in several ways. If it is possible to determine at run-time the type of any given object, then maps describing the structure (location of included pointers) of each object type can be employed. In a statically-typed system, a type map can even provide the types of referenced objects, removing the need to store separate type information. In some Algol 68 systems the type map was implicit in code generated by the compiler to trace through a given type [Wod71, Mar71]. A second approach to this problem is the tagging of machine words to identify which ones are pointers. Here, pointers are located by identifying those words within an object that are pointers. The third approach is used in *conservative* collectors [BW88, DB76, Rov85]; that is, collectors that fail to reclaim some inactive objects, though they never reclaim an active object. This approach identifies words

that *might be* pointers by checking the referenced storage to see if it is in fact an object. This check is usually accomplished through the use of a big bag of pages; if the address is within a bag, and corresponds to the first address of an object within the bag, then it is a pointer.

## 2.2 Data-flow analysis

Data-flow analysis is one technique for discovering properties of programs. The information from this analysis is used when optimizing programs. The terms introduced here will be used later in the paper when describing existing optimizations and proposing new ones.

### 2.2.1 Control flow

A program comprises memory and instructions. *Control-flow analysis* determines what paths through the instructions are possible. Instructions are first divided into *basic blocks.* A basic block is a sequence of instructions with the property that if one instruction is executed, then all instructions are executed; that is, a basic block is single-entry straight-line code. Basic blocks are the nodes in the *program flow graph*. A directed edge $\langle b_1, b_2 \rangle$ in the program flow graph represents a possible transfer of control from the end of $b_1$ to the beginning of $b_2$. A program flow graph is one instance of a *flow graph.* A flow graph is a directed graph $(N, E)$ with a single *entry* or *start* node $n_0$ from which all other nodes are reachable.

Certain properties of flow graphs are useful in many flow analysis algorithms. A node $n$ is said to *dominate* another node $m$ if all paths from the root $n_0$ to $m$ pass through $n$. Within a graph a *strongly connected region* (abbreviated SCR) is a subgraph $S$ such that for every pair of nodes $m$ and $n$ in $S$, there is a path from $m$ to $n$. A *strongly connected component* (abbreviated SCC) is a strongly connected region that is not contained within any other strongly connected region. If, given an SCR $S$, there is a node $h$ such that $h$ is in $S$ and all paths from $n_0$ to nodes in $S$ pass through $h$ is called a *single entry strongly connected component* with *header node* $h$ (abbreviated SESCR). A SESCR roughly corresponds to a "loop" in the source language. Note that $h$ dominates every node in a SESCR.

Many flow graphs have a property called *reducibility.* A *reducible* graph $G$ has the property that repeated *interval reductions* of $G$ eventually yield

a single node. A single interval reduction partitions a graph into *intervals*. Each interval $I$ has a header node $h$ and the properties that (1) $h$ dominates every node in the interval and (2) all cycles within the interval contain $h$. Reduction into intervals was developed by Cocke and Allen [Coc70, All70]. More efficient algorithms for performing interval reduction and testing for reducibility were later developed by Hecht and Ullman [HU72], Graham and Wegman [GW76] and Tarjan [Tar74].

### 2.2.2 Local analysis

Local data-flow analysis determines the relationships holding between definitions (stores) and uses (fetches) occurring within a single basic block. Value-numbering [CS70, Ken81] can be used to eliminate redundant expressions and to reduce constant expressions within a block. Another approach constructs an expression DAG representing the transmission of information within the block [ASU86]. Both of these techniques also generate the set of expressions in the block that are still *available;* that is, expressions that will yield the same result if re-evaluated at the end of the block.

### 2.2.3 Global analysis

Global data-flow analysis determines the relationships between definitions and uses of variables and results and occurrences of expressions throughout a program.

#### Liveness and availability

An expression $e$ occurring at an instruction $i$ is said to be available if on every path from the entry node $n_0$ to $i$ (1) $e$ appears, and (2) no variable in $e$ is redefined between the last occurrence of $e$ on the path and $i$. Availability corresponds directly to redundancy; if $e$ occurs at $i$ and $e$ is available at $i$, then it is possible to avoid evaluating $e$ at $i$.

A variable $v$ is said to be *live* at an instruction $p$ if there are two instructions $o$ and $i$ such that $v$ is defined at $o$, used at $i$, and there is a path from $o$ to $i$ which passes through $p$ but not through any instruction redefining $v$. If $v$ is not live at $p$, then (at $p$) either $v$ is never defined or $v$'s value will not be used in any future evaluations. Liveness is also a property of definition sites; a definition $d$ (of a variable $v$) is live at $p$ if there is a path from $d$ to a

use $i$ passing through $p$ without passing through any instruction redefining $v$. The set of instructions where a definition $d$ is live is called the *live range* of $d$.

#### Use-definition chains

Use-definition chains relate definitions of a variable to possible uses of that variable and uses of a variable to possible definitions of that variable. Given a use $u$ of a variable $v$, the set of possible definitions for $v$ at $u$ is written $DEFS(u)$. Given a definition $d$ for a variable $v$, the set of possible uses of $v$ from $d$ is written $USES(d)$.

## 2.3 Optimizations

"Optimizing" compilers apply a number of transformations to programs in order to increase their expected efficiency. Described here are well-known transformations that will appear again later in the dissertation. All of these transformations will be shown to interfere with garbage collection in some way. It is especially notable that important opportunities for performing several of these optimizations arise in the compilation of addressing arithmetic [ASU86]; this is precisely where these optimizations can interfere with garbage collection.

### 2.3.1 Register allocation

The abstract operational semantics for languages often make use of addressable memory. Addressable memory is a component of conventional computers, but these computers also contain a small amount of much faster memory known as registers. The goal of register allocation is to associate objects with registers in a way that makes the compiled program run as fast as possible.

One good method for allocating registers uses graph coloring [CAC+81]. This method treats each live range of a variable as a vertex in a graph $G$, and creates an edge between two vertices $v_i$ and $v_j$ if the corresponding live ranges intersect. If $k$ is the number of registers available and $G$ can be colored with fewer $k$ colors, then all of the variables may be stored in registers. If $G$ cannot be colored with $k$ or fewer colors, then some variables must either not be placed in registers (removing vertices and edges from $G$) or *spilled.* A

variable is spilled if it is moved from one register to another or if it is moved from a register to memory. If a variable $i$ is spilled, then $v_i$ in $G$ may be split into several vertices that share the edges originally incident to $v_i$.

### 2.3.2 Redundant expression elimination

Redundant expression elimination seeks to avoid re-evaluating duplicated expressions. To do this it identifies duplicated expressions, introduces a temporary variable, saves the result of the first evaluation in the temporary and uses the value stored in the temporary instead of evaluating the duplicate. Availability is used to determine when an expression is redundant.

### 2.3.3 Loop reduction in strength

Loop reduction in strength converts multiplications involving a loop counter into additions. To see that this is possible, suppose that P(x) is a degree $n$ polynomial in $x$ with $n > 0$. Consider $P(x+1) - P(x)$. This is a degree $n-1$ polynomial in $x$, $P'(x)$. Reduction in strength converts the loop

$$\begin{array}{l} \textbf{for } x = 1 \textbf{ to } 100 \\ \quad \ldots P(x) \ldots \end{array} \quad to \quad \begin{array}{l} t \leftarrow P(1) \\ \textbf{for } x = 1 \textbf{ to } 100 \\ \quad \ldots t \ldots \\ \quad t \leftarrow t + P'(x) \end{array}$$

Since $P'(x)$ is also a polynomial in $x$, this process can be repeated until no multiplications remain, yielding this loop:

$$\begin{array}{l} t \leftarrow P(1) \\ t' \leftarrow P'(1) \\ \ldots \\ t^{(n)} \leftarrow P^n(1) \\ \textbf{for } x = 1 \textbf{ to } 100 \\ \quad \ldots t \ldots \\ \quad t \leftarrow t + t' \\ \quad t' \leftarrow t' + t'' \\ \quad \ldots \\ \quad t^{(n-1)} \leftarrow t^{(n-1)} + t^{(n)} \end{array}$$

Allen, Cocke and Kennedy give an algorithm for performing reduction in strength on a simple intermediate code that can reduce polynomials in this way [ACK81].

### 2.3.4 Loop invariant code motion

Loop invariant code motion attempts to find expressions or code whose results do not change during a given execution of the loop. This is an important optimization because any improvements are multiplied by the number of times the loop is executed.

The discovery of invariant expressions is fairly simple. A variable $v$ (a trivial expression) is loop-invariant if there is no definition for $v$ within the loop body. A non-trivial expression $e_1$ OP $e_2$ is invariant if $e_1$ and $e_2$ are loop-invariant. Provided that the code motion is safe [Ken72], a loop-invariant expression may be evaluated before entering the loop. More ambitious algorithms move invariant assignment statements and control structures out of loop bodies [CLZ86].

### 2.3.5 Dead code elimination

Dead code elimination removes code when the compiler can determine that it has no effect on the program's output. This optimization is usually intended to function as a cleanup phase after other transformations have been applied.

Dead code elimination uses use-definition chains to discover instructions that have no effect on a program's output. If an instruction $i$ defines a variable $v$, and $USES(v)$ at $i$ is empty, then executing $i$ has no effect on the program's output. Therefore, $i$ may safely be removed.

### 2.3.6 Procedure linkage conventions and tailoring

Language implementations define a standard procedure interface to allow linking of separately compiled procedures. Such an interface is general and often includes support for saving of registers, debugging, and exception handling. Without knowledge of the behavior of a called procedure, a compiler must also make worst-case assumptions about the effects of the procedure on its parameters and global variables. This generality can be costly.

*Linkage tailoring* [AC72] reduces the cost of a procedure linkage by using a special-purpose non-standard interface. Allen and Cocke describe four classes of procedure linkage.

**closed** This is a standard procedure linkage. Registers are saved at procedure entry and restored at procedure exit, and results must appear in a standard register(s). Parameters are usually passed in memory.

**open** This is no linkage at all; the procedure is entirely incorporated into the calling program. This can be very profitable because there is no overhead at all, and the procedure body is exposed to optimizations within the calling program.

**semi-open** The calling and called procedures are compiled at the same time, but a procedure "boundary" is maintained. Properties of the actual parameters (in the caller) can be used to optimize code in the called procedure. Register saving may be reduced or omitted, and parameters may be passed in special locations or registers, or omitted altogether.

**semi-closed** The called procedure is compiled first, and information from that compilation is used when compiling the caller. The linkage may pass parameters in registers or special locations, and information about the called routine may allow less conservative assumptions and more optimization when compiling the caller.

# Chapter 3

# Interference

The combination of garbage collection and optimization in the same system may not work well. Garbage collectors locate pointers and objects, and a good run-time environment for garbage collection makes these operations reliable and inexpensive. Several useful optimizations, however, change the run-time environment in ways that make pointer and object location unreliable or expensive.

Consider the following example (in which $a$ is dynamically allocated).

$$x \leftarrow a[i]$$
$$a[i] \leftarrow \textbf{new}()$$

Here, the address of the $i$th element of $a$ is calculated twice, once before allocating memory and once after. Ordinarily it is safe to calculate the address once and reuse it, but if the memory allocation triggers a (compacting) garbage collection then $a$ will be moved and the cached address will be incorrect.

One solution to this problem is to not optimize address expressions, or at least to not optimize address expressions across garbage collections. Modifications to the garbage collector can allow some optimizations, but an uncooperative or careless optimizer will still cause problems. I propose a better solution to this problem that allows garbage collection and optimization to coexist.

## 3.1 Setting

The language compiled will be "safe" [Owi81], in the sense that garbage collection always works correctly with unoptimized code; the interference is with the compiler, not the programmer. The implementation of the language will not use concurrency; this is a simplifying assumption. I also assume that the language can be typed-checked at compile time. Languages whose implementations might produce such an environment include typed functional languages, Russell, ML, and **free**-free Pascal.

The garbage collector will be based on a copying-compacting collector; it will be a "stop-the-world" collector, invoked only when an object is allocated. In general, the collector is invoked rarely; this assumption is used in deciding which compiler transformations are likely to be profitable. Ungar's generation scavenging collector [Ung84] is an example of an efficient collector meeting these requirements. I use a copy-and-compact collector for several reasons. It appears to be the best collector in a system with plenty of real memory; the cost of a single collection is proportional to the amount of memory in use, not to the amount of memory available, and object allocation is fast and simple. It also produces the most interesting interferences with optimizing compilers, because a compacting collector will change pointer values (as opposed to a mark-and-sweep collector, which will not).

I also assume that is little or no hardware for garbage collection, and I admit the possibility of machine registers.

## 3.2 Interference

### 3.2.1 Register allocation

Garbage collectors look for pointers in standard[1] locations. Introducing cross-statement register allocation introduces another set of locations where pointers can be stored. If the collector is not designed to examine those locations, then leaving values in registers will cause the garbage collector to fail. This is a simple interference, but it cannot be ignored in a real system.

The garbage collector must know exactly which registers contain pointers. Methods for locating pointers in memory may not work with registers;

[1] What do I mean by "standard"? I guess I mean that the collector is designed with one set of assumptions, and thus "standard" means "assumed".

pointers in memory can be located because they are part of a larger object whose structure is known, or pointers themselves may be tagged. Registers are not part of any larger structure, so that approach will not work, and maintaining tags without hardware assist is time-consuming.

At procedure calls values in registers are moved into a register save area. This is another location for pointers that might not have been anticipated by the author of the garbage collector. Introduction of special procedure linkages may also leave active pointers in unusual places.

### 3.2.2 Redundant expression elimination, loop invariant code motion, and reduction in strength

Garbage collection interferes with these three optimizations in similar ways. A storage allocation represents a potential call to the garbage collector, which in turn represents a redefinition of every active pointer value. Since the compiler can only perform safe transformations, it must treat each allocation as a potential redefinition of every active pointer value, and there will be fewer opportunities to apply these optimizations to address expressions. On the other hand, the compiler could proceed with the optimizations and leave the problem of discovering encoded pointers to the garbage collector.

Reduction in strength of address expressions is a productive optimization in languages that use arrays in their implementation. When this is not the case (for example, in classical implementations of simple Lisp), reduction in strength cannot be applied to address expressions because there is no guarantee that the components of aggregates will be linearly addressable.

The usefulness of redundant expression elimination and loop invariant code motion applied to addresses also depends upon a language's semantics and implementation. In many Lisp implementations, for example, all values are represented with pointers to storage even when the storage for the value is no larger than the pointer (containment by reference). This permits a uniform treatment of data in the implementation (this is necessary for efficiency in the absence of type-checking). If there are no destructive operations (e.g., **rplaca** or **rplacd**) then these optimizations cannot be profitably applied to addresses. Because all values are represented with pointers, taking the address of a value yields a pointer to a pointer; subsequent accesses take two indirections, not one. If, however, **rplaca** and **rplacd** are allowed, then there are situations in which redundant expression elimination can profitably be

```
(cdr x)
⋮
(rplacd x y)
```

Figure 3.1: Opportunity for REE in Lisp

applied to an address expression. For example, in figure 3.1 the operation **(cdr** $x$**)** is followed by **(rplacd** $x$ $y$**)**. In this case, the address of the **cdr** field of $x$'s storage may be re-used when $x$'s **cdr** field is replaced[2].

If a language's implementation uses containment by value, then other opportunities may arise. When a value contained in another is accessed several times, its address (offset from the start of the storage for the containing value) will be recomputed each time. If the contained value uses less storage than a pointer and it will not be modified, then it saves time to cache the value in a temporary and re-use it. If the value might be modified, or if it uses (a great deal) more storage than a pointer, then it will be more profitable to cache the pointer and re-use it[3]. A hindrance to the use of a compacting garbage collector in Algol-like languages is the use of "VAR" (reference) parameters. Record data structures in Pascal and Modula are implemented with containment by value; passing an element of a record as a VAR parameter to a subroutine creates a pointer into the middle of the record[4].

Without overwriting or containment by value, redundant address expressions do not appear in a functional language because there is no need to re-use the address of a value; there are no side-effects in a functional language and a value (actually a pointer to storage) takes no more memory than a pointer to a value. However, Hudak and Bloss describe overwriting optimizations for a functional language that, given a naive translation from Fortran to functional form, introduces the same overwriting behavior present in the original Fortran program [HB85]. In doing this it also introduces the redundant addressing expressions present in the original program. Thus, the use of overwriting optimizations with a functional language can introduce redundant address expressions.

[2]For the sake of example, pretend that the **rplacd** couldn't be done more quickly in a single machine instruction.

[3]If it might be modified, then it is more *correct* to cache the pointer.

[4]Thanks to John Ellis for pointing out this problem.

For an optimizing compiler, any storage allocation represents a potential redefinition of every pointer value in the program. If a strongly connected region of a flow graph contains a storage allocation, then there are no address-valued region constants, and thus no expression containing an address will be inductive or invariant. If there is a path between two otherwise equal address expressions that contains a call to the storage allocator, then the second expression is not redundant and cannot be eliminated.

### 3.2.3 Dead code elimination

Dead code elimination interacts with garbage collection in unusual ways. In certain situations, apparently "dead" code helps the garbage collector; if it is removed, then garbage collection is less efficient or effective. Code to initialize newly allocated memory may appear to be dead, but these initializations can be important to the operation of the garbage collector.

#### Aids to garbage collector

There are several situations in which a program's performance is improved by the addition of apparently useless assignment statements. This optimization is currently done "by hand", but there is no reason that an optimizing compiler for a garbage collected system should not also attempt to perform this optimization. In all cases, the optimization is the assignment of a null pointer value to one part of an object, and in all cases that object remains accessible, but only certain parts of it will be referenced in the future. In most cases this object is an activation record. The program's performance is improved in two ways: by removing pointers, the cost of tracing through the graph of active objects is reduced; by removing pointers, some active objects may be made inaccessible and will be reclaimed in a garbage collection. By increasing the effectiveness of each garbage collection, the frequency of collection is lowered and the total cost reduced. In certain situations, this optimization can break cycles in the storage graph, thus allowing collection by a reference-counting collector.

When this optimization is performed correctly, the pointer variable receiving the null value will not be used in any subsequent point in the program. This can often be detected by an optimizing compiler through the use of data-flow analysis, and it is usually inferred that such an assignment need not be performed because it has no effect on the program's execution (certainly it

has no effect on the *meaning* of the program). Unfortunately, removing this "dead" assignment undoes the assistance to the garbage collector.

In the first example (figure 3.2) three objects $a$, $b$ and $c$ are created before the call to $g$, but only $b$ and $c$ are used during or after the call to $g$. If it is likely that $g$ will cause a garbage collection and $a$'s value uses a large amount of memory, then it may speed up the program to assign **nil** to $a$ before the call to $g$. This assignment removes any reference to $a$'s value that originates in an activation record for this code, possibly making the storage for $a$'s value available for reuse.

$a \leftarrow \ldots$
$b \leftarrow \ldots$
$c \leftarrow f(a, b)$
$a \leftarrow$ **nil** remove the pointer to $a$'s value from this stack frame
$d \leftarrow g(c, b)$
$\ldots$

Figure 3.2: Kill a pointer before a call

In the second example (figure 3.3) the function $f$ returns a function $g$. Depending upon the semantics of the language and the details of the implementation, the representation of $g$'s value may include a pointer to an activation record for $f$. If this is so, then the activation record will be placed on the heap and all values reachable from that activation record will remain live as long as $g$ is live. However, $g$ may not use all the values accessible from $f$'s activation record; to allow better reuse of storage, references that are effectively dead are removed from the activation record. Note that the need for this optimization depends upon the language implementation; it is by no means universal.

### Storage initialization

Several languages that require garbage collection also support the creation of "array" or "tuple" objects; that is, objects containing many other objects, all accessible in constant time. These objects are usually implemented as arrays of storage; that is, as enough contiguous words of memory to contain the entire object. When these objects are allocated, all of their components must be given some initial value; they cannot contain "junk" values if a garbage

```
f(a) = {
  b = ...
  c = ...
  ...
  g(x) = {
    ...
    }    end of g's definition
  ...
  a ← nil
  b ← nil      a and b are not referenced by g
  return g
  }
```

Figure 3.3: Clear a frame before returning a closure

collection occurs before everything has its intended (by the programmer) value. This problem occurs even in languages that have no destructive operations, and no way for the programmer to avoid supplying an initial value for objects. In figure 3.4 is a simple FP [Bac78] function and a plausible compilation demonstrating this problem. The function is "apply-to-all iota". It takes as input a tuple of integers and produces a tuple of tuples of integers.

The first allocation obtains storage for *result*. If *result*'s storage is not initialized, then any garbage collection provoked by a subsequent allocation may attempt to treat uninitialized portions of *result*'s storage as if they were pointers. This is not necessarily a serious problem; if the collector checks pointers to be sure that they are valid before tracing them, then it will at worst result in some garbage being uncollected. However, if the collector trusts all pointers (a possibility in a strongly-typed language), then it may treat pieces of objects as if they were object headers (setting tag bits, for instance).

If an optimizing compiler were to ignore the requirements of the garbage collector, it might very well discover that the pre-initializations are dead assignments and remove them. This is clearly unacceptable.

$\alpha\iota$
$\iota : n = \langle 1, 2, \ldots, n \rangle$
$(\alpha f) : \langle x_1, \ldots, x_n \rangle = \langle f : x_1, \ldots, f : x_n \rangle$
$(\alpha\iota) : \langle 2, 3, 4 \rangle = \langle \langle 1, 2 \rangle \langle 1, 2, 3 \rangle \langle 1, 2, 3, 4 \rangle \rangle$

```
n ← length(input)
result ← allocate(n)
for i ← 1 to n {
   result[i] ← allocate(input[i])
   for j ← 1 to input[i] {
      result[i][j] ← j
      }
   }
```

Figure 3.4: Apply-to-all iota and its compilation

## 3.3 Coping with interference in the collector

### 3.3.1 Identifying pointers to objects

The compiler can perform register allocation across storage allocations if the garbage collector can determine which values in registers are in fact pointers.

One way to do this is to reserve (by convention, not by architecture) some bits in each machine word for tags. This has the disadvantage of wasting space and not taking advantage of machine arithmetic designed for the whole word, but it does work correctly when pointers are stored in registers.

Another way to locate pointers in registers is to reserve some registers exclusively for pointer values. This has the disadvantage of complicating register allocation in the compiler, but it will allow some register allocation in a garbage-collected system. Note that it is not safe for the author of the garbage collector to assume that "address registers" specified in the machine's architecture will always and only hold pointers; the architectural restriction is on the *use* of the registers, not their contents. A clever compiler writer might use the other set of registers for spilling when they are available.

If the garbage collector does not relocate objects, then it is permissible for it to treat non-pointer values as pointers. However, doing this requires that a value pointing into the object space but not pointing at a particular object

be identified; otherwise the collector might treat data found within the object as header information private to the collector and modify it. To identify non-object addresses the collector can either use a big bag of pages (BIBOP) or unforgeable object headers. With a BIBOP, each candidate pointer locates a corresponding bag of pages with associated object size and start address. The address of the beginning of the bag is subtracted from the pointer, and that result is divided by the bag's object size. If the remainder is zero, then the pointer in fact addresses the beginning of an object. Unforgeable object headers typically require a second table not accessible to the program; an object header contains an index into the table which in turn identifies the object. Doing this requires making an entry in the table at each allocation and a separate garbage collection for the header table. Neither of these solutions is entirely satisfactory.

The compiler can store information describing the register assignment at various points within the program. Given this information, the garbage collector can accurately locate pointers to objects stored in registers. This solution seems best, because its time overhead occurs only during garbage collection. There is some space overhead; there must be a map describing the register assignment at each "point" in the program where a garbage collection might occur; this includes storage allocations, and can include calls to subroutines. The need for a map at each call site depends upon the register saving conventions for subroutine calls. If the called routine saves registers, then it will store them contiguously in uninterpreted storage. This situation requires an assignment map for each call site, because in order to trace pointers stored in registers the collector must (1) restore the registers from uninterpreted storage according to the save mask and (2) interpret the registers according to the assignment map in effect at the call site. Restoring the registers for a given activation record also requires that all activation records below (called by) the given one be scanned because a called procedure need not save all registers.

If the calling routine saves registers, then it may either store them contiguously in uninterpreted storage or (assuming that the registers shadow cells in addressable memory) store them back to their locations in addressable memory. The second choice requires no assignment maps for call sites because all the values stored in registers have been copied out of the registers. There are several trade-offs that must be evaluated in a real (optimized) system.

1. The use of a caller-saves convention simplifies and improves the implementation of tail-call elimination, and also simplifies the implementation of continuations if the language supports these. If a language supports exception-handling, then caller-saves allows a simpler, more efficient processing of exceptions. In general, use of caller-saves makes it possible to examine an activation record without referring to other activation records. Steele and Sussman discuss this [Ste77a, SS76].

2. Storing registers into contiguous storage can be much faster than storing them into cells scattered in memory. Often there are special machine instructions to do this.

3. Hennessy and Chow found that their register allocator was not as effective with a caller-saves convention [CH84]. Their measure of effectiveness, however, is static, not dynamic.

4. Caller-saves increases code size because the code to save registers must be replicated at every call site. The code size is increased even more if registers are stored back into their shadowed cells instead of in contiguous storage, but this removes the need to have an assignment map for each call site.

### 3.3.2 Discovering targets of offset pointers

The techniques above allow a garbage collector to find pointer values. However, if redundant expression elimination, loop invariant code motion or reduction in strength has been applied to address expressions, then some pointers may address the interior of an object instead of the head of an object. To interpret the contents of an object the collector must find the head given one of these offset pointers.

It seems that techniques used above to separate pointer values from non-pointer values could be used to adapted to find the object containing a cell addressed by a pointer. With a BIBOP this can be done with a little arithmetic; the header can be found by subtracting from a pointer the remainder after subtraction of the beginning address of the BIBOP and division by the BIBOP's object size. With a header table the collector can either scan backwards from the addressed cell until a header is discovered, or a search can be performed in the header table itself.

Unfortunately, all of these methods depend upon the assumption that an offset pointer will address memory contained within the object upon which it is based. There is no reason that this should be true. Figure 3.5 shows a piece of code for which the compiler might generate a constant address used to access elements of the array $A$ (with elements indexed from 0 to 100), but not by itself addressing any part of $A$. One way to generate code for this loop might create the region constant $addr(A) + 200$; when combined with $i$, the result lies within $A$, but the constant itself does not address any part of $i$. It is possible to prevent the creation of pointers offset outside of objects by not allowing the compiler to treat any addressing intermediate results as inductive or redundant. When performing reduction in strength the compiler must be also be careful not to increment inductive address variables past the end of an object on the last loop iteration.

```
for i ← 100 to 200
    A[200 − i] ← i
    ...
```

Figure 3.5: Generating an offset pointer out of an object

## 3.4 Coping with interference in the compiler

As noted above, the compiler can produce maps for use by the collector and debuggers describing the assignment of registers to variables. A variation of this generalizes to handle the offset pointers produced by loop invariant code motion, reduction in strength, and reduction expression elimination.

The previous section implies that the compiler produces register assignment maps *interpreted by the garbage collector*. Suppose that a "map" is stored as code and data that moves register and save area contents into the variables' "true" locations (that is, it places pointers in locations where the collector will look for them). After the garbage collection, another piece of code is run to restore values into registers and save areas so that code executing after the collection will see the updated data. The collector must still interpret maps describing the contents of activation records and other objects, but it does not need to reconstruct an activation record before examining it. I will call these maps the *cleanup* and *dirtyup* maps. This implementation

of register assignment maps has the disadvantage of requiring more storage for the maps, but has several advantages which should outweigh this.

First, executed maps should be faster than interpreted maps. Second, this method helps define a *flexible* interface between the compiler and the garbage collector. The collector does not need any special-case code to deal with customized procedure linkages, and the compiler does not need to know about the details of the garbage collection algorithm. This interface can also be used to simplify the implementation of debuggers for optimized code; by using it, a debugger does not need to consider register assignment.

Third, and most important, this encoding of maps as code and data generalizes to allow treatment of offset pointers produced by redundant expression elimination, reduction in strength, and loop invariant code motion. For REE and LIVCM, the cleanup map does nothing, though the compiler is constrained to leave available the components of invariant and redundant address expressions. After a garbage collection, the dirtyup map calculates new values for the address expressions. Notice that this code in the dirtyup map is a *use* of the expression components; if it is treated in the same way as ordinary code during dead code elimination the constraint mentioned above is automatically satisfied.

Reduction in strength requires a slightly different approach. After addition reduction in strength of $A+i$ into $t_{A+i}$, where $A$ is an array address and $i$ is an induction variable, one would like to do away with $i$ and the code to increment it. The method used for REE and LIVCM will not allow this. By storing $A$ (a component of the expression) and $A'$ (a copy of $A$), however, it is possible to implement the maps without reference to $i$. The cleanup map does nothing; during a collection, $A'$ and the temporary variable containing $t_{A+i}$ are not traced or altered. After the collection, the new $t_{A+i}$ and $A'$ are updated by

$$\begin{aligned} t_{A+i} &\leftarrow t_{A+i} - A' + A \\ A' &\leftarrow A \end{aligned}$$

The compiler-generated maps also make it possible to avoid performing some initializations of newly allocated storage. In some cases the initialization code is truly (and correctly) dead and will be removed anyway. In other cases this is not so, but the initialization can still be omitted if the undefined portions of the array are initialized before a garbage collection. This is especially true in code where the iteration is generated by the compiler, as

it is in the "apply-to-all iota" example in figure 3.4. At the call to *allocate* within the loop, it is clear that elements 1 through $i-1$ have been initialized and that elements $i$ through $n$ have not. Notice that this situation demands exact information; the garbage collector depends upon complete definition of all objects, but it is (very) wrong to initialize an already defined element to a null value.

For languages with reference parameters and containment by value (e.g., Pascal and Modula-2), compiler-generated maps can make it possible to use a compacting garbage collector[5]. The cleanup maps do nothing. The dirtyup maps must be run in order from the root procedure in the call chain to the leaf procedure, passing information down the call chain. In a procedure $A$ at a site calling $B$, it is clear whether or not the reference parameters are contained (by value) within some larger object in $A$'s scope. If they are, then the compiler must pass the new location to $B$'s dirtyup map so it can move the parameters in $B$'s scope. In $B$, it is clear to the compiler which parameters are reference parameters, and thus it can generate code to handle movement of the parameters. It is very important that Pascal and Modula-2 do not allow generation of pointers except with calls to **new**. If it were possible to take the address of a reference parameter and assign it into a global, then the scheme described here would not be sufficient. However, because this is not allowed it is guaranteed that all offset pointers can only appear in activation records and temporaries generated by the optimizer.

This is similar in spirit to the work of Hennessy [Hen82] and Zellweger [Zel83] on debugging optimized code. Like a debugger, a garbage collector must recover information about a program's execution. Unlike a debugger, a garbage collector must always run and must obtain all of the information it needs, but it is only run at certain points within a program. A debugger might be used to examine a program's state at any point in its execution.

Handling interference in the compiler is preferable to handling interference in the collector because it hinders fewer optimizations and does not add run-time overhead in the garbage collector or allocator. It has the disadvantage of requiring more space.

[5]In the absence of the *ADR* system primitive.

## 3.5 Adapting existing algorithms

To cope with garbage collection in the compiler, existing optimization algorithms must be adapted to generate the maps used by the garbage collector and satisfy any other constraints imposed by the garbage collector. If the problem of code space is ignored, it appears that these modified algorithms will perform the same improvements as the original versions[6].

In general, the existing algorithms are modified to

1. ignore the effects of garbage collection when discovering expressions that are redundant, invariant, or reducible;

2. insert temporary variables as in the original algorithm, but for each temporary record the generating expression and the reason for the temporary (is the expression invariant, redundant, or reducible?);

3. compute the live ranges of the introduced temporaries;

4. for each garbage collection or call site, generate the before and after maps to restore the values of temporaries live at the site. This takes place before dead code elimination, so the values needed to (re)calculate the values stored in the temporaries are still available. The temporaries generated by reduction in strength present a special case because it is desirable to avoid introducing uses of induction variables. Note that this code is treated in the same way as the "ordinary" code for purposes of program analysis and optimization.

5. add to the maps code to finish initializing incompletely initialized objects, and leave hints to the dead code eliminator so that it will discover which allocation initializations may be removed.

6. perform dead code elimination.

It is difficult, in general, to tell which objects need not be initialized at allocation. The compiler must be able to gather exact information describing the definition of a value at every collection site reachable from the value's creation. The information at all sites need not be the same, but it must always be exact.

[6]There will be fewer dead variables, but this does not directly affect execution speed.

# Chapter 4

# Allocation optimization and analysis

Compilers for Lisp and other languages with garbage collection make use of some optimizations designed to avoid allocation of garbage-collected storage. These optimizations fall into three general categories: those that use other allocation methods (static and stack) to obtain memory, those that directly re-use previously allocated storage, and those that assist the garbage collector in its operations. Re-use of storage is especially attractive because it avoids the cost of initializing the object. The motivation for the Lisp optimizations is to reduce the time and space costs of frequent operations that "ought" to be cheap.

Though not a motivation for any of the work described here, the previous chapter provides two more reasons for avoiding allocation of garbage collected memory. Addressing expressions *can* be optimized if the address is known not to refer to garbage-collectible memory; thus, converting a heap allocation to a stack or static allocation may permit more optimization. Addressing expressions can also be optimized if the live range of the temporaries generated does not include a heap allocation site; converting a heap allocation to a stack allocation may remove a heap allocation site that hinders optimization.

## 4.1 Non-heap allocation

Conversion of heap allocations to stack or static allocation is usually used to improve the cost of procedure calls and speed up arithmetic. These are common operations that are expected to be cheap.

### 4.1.1 Activation records

In languages with "first class functions" and lexical scope, it is possible to write a function (call it *f*) which returns a lexically enclosed function (call it *g*). With a classical implementation of lexical scope [ASU86, pages 416–422], this creates a need for non-LIFO (non-stack) allocation of procedure activation records because each instance of $g$ returned by $f$ may contain references to an activation record for $f$, even when $f$ itself is not active. Allocating activation records from the heap, however, adds a large cost to each procedure call. Therefore, compilers for languages with this combination of features usually include special-purpose analysis to determine when a function's activation record will not survive the activation of the function, and use stack allocation when this is the case [KKR+86, BGS82, Ste78]. Methods to do this are ad hoc, cheap, and usually effective.

In the Rabbit compiler [Ste78], the results of evaluating lambda (function-producing) expressions are traced to determine their use. Uses are divided into three categories: function as data, in which the value produced by the lambda expression is assigned to a variable with a non-function (not applied to arguments) use; functions referenced (applied) from data, in which the value produced by the lambda expression is assigned to a variable with only function use, but some of those uses occur within functions treated as data; and functions as functions, in which the value of the lambda expression is only applied to arguments, and only within functions that are neither used as data nor referenced from data. In the first case, it is necessary to create a "full closure" containing both a pointer to the code and the environment in which the lambda was evaluated. In the second case, it is necessary to create a "partial closure"; all possible uses of the function have been discovered, but it is still necessary to retain the environment. Note that "referenced by data" includes indirect references through other functions referenced as a data. In the last case, no closure is necessary because the function is known and the environment can be recovered from the environment active at the call site. Examples of these three uses are shown in figure 4.1. Here, the enclosing

$f = \lambda x.\{$
$\quad g = \lambda y.\{\ldots\}$
$\quad h = \lambda z.\{\ldots g(z)\ldots\}$
$\quad i = \lambda w.\{\ldots\}(x)$
$\quad$ `return` $cons(h, i)$
$\quad \}$

Figure 4.1: Various function uses

function (with unspecified use) is $f$. Within $f$, the function $h$ is treated as a value and must be represented in the most general form. The function $g$ is referenced from $h$, and so needs a saved environment for possible execution after leaving the current scope. It is not necessary to save a pointer to the code for $g$ in the closure because the only reference is from $h$, where the address of $g$'s code will be known. The anonymous function applied to $x$ to produce $i$ does not need any special treatment because it can only be called when $f$ is active. Any references to names declared in $f$ can be made directly through $f$'s activation record.

It is worth noting that the access link and display implementations of lexical scope are "by reference"; in true functional languages lexical scope can be implemented "by value". With an access link implementation, function values use pointers to their lexical ancestors' activation records to obtain bindings for lexically inherited variables; with scope "by value" it is possible to copy the bindings into the function object when it is created. This slightly increases the time to generate a function-valued object, but simplifies the rest of the implementation. Notice also that scope-by-value preserves only those variables that are actually used by the returned function; all others are collectible. In a scope-by-reference implementation, values bound to variables in a heap-allocated activation record cannot be collected until the activation record is collected, even though only a few of the variables are actually referenced. Yet another implementation of lexical scope by reference treats a variable binding as a pointer to an "assignable cell", which contains the the actual value. In this case, inheritance of a binding from a lexical ancestor is performed by copying the binding (a pointer) into the inheriting frame. This technique requires an additional indirection to reference lexically scoped variables and a heap allocation to obtain the assignable cell, but avoids extending the lifetime of lexically enclosing activation records when a function-valued

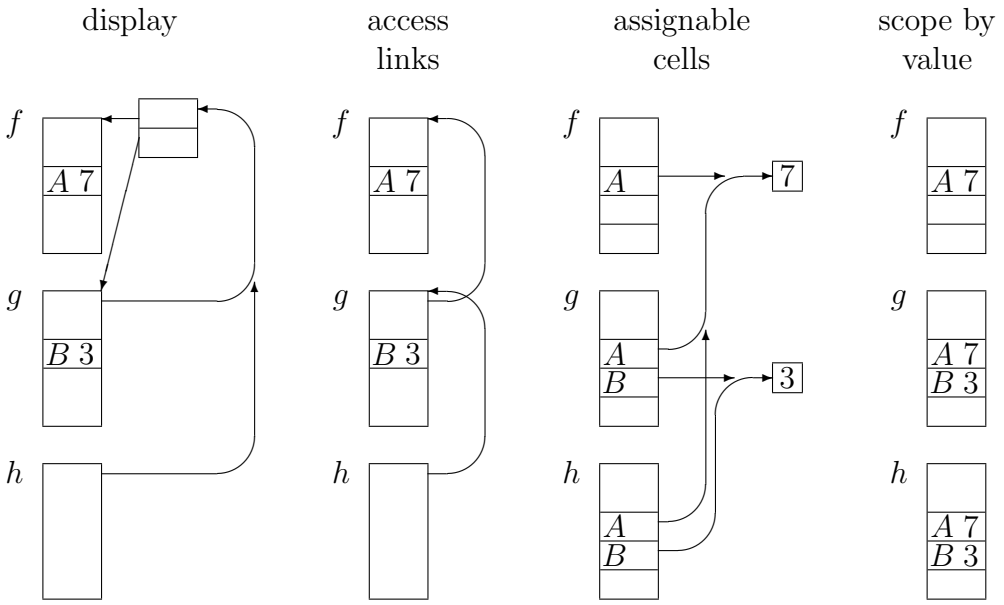



Figure 4.2: Different implementations of lexical scope

object is returned. In figure 4.2 shows the references resulting from these four different ways of implementing lexical scope. The function $h$ is nested within $g$ which is in turn nested within $f$. The variable $A$ is defined in the scope of $f$, and the variable $B$ is defined in the scope of $g$.

### 4.1.2 Numbers

In the MacLisp [Ste77b] and S-1 Common Lisp [BGS82] compilers, efficient execution of arithmetic operations is an important goal. This goal is hindered by Lisp's uniform treatment of objects and the resulting need for a consistent data representation; all values are represented as pointers to storage, which is in turn (generally) obtained from the garbage-collected heap. To speed up arithmetic operations, the Common Lisp compiler[1] searches for opportunities to use a direct (non-pointer) representation for numbers, or failing that, to allocate the numbers on a data stack rather than on the heap.

Several representation constraints are maintained by the compiler. The

[1]The analysis in the Common Lisp compiler is cleaner, more general and more correct than the analysis in the MacLisp compiler, though they are similar.

**cons** operation always creates a heap-allocated object; all objects reachable from heap-allocated objects must themselves be heap allocated; all arguments to (used-defined) procedures must be represented as pointers, though not necessarily to heap-allocated objects; and values returned by functions or stored in global variables must be represented as pointers to heap-allocated storage. Operations that require a heap-allocated object are called "unsafe".

The first part of "representation analysis" discovers opportunities to avoid representing numeric values with pointers. In a top-down pass it propagates the most desirable representation (WANTREP) from value consumers to value producers. For example, in "*(***if** *p x y)*" the WANTREP for the entire expression is propagated inward to the expressions $x$ and $y$, and the WANTREP for $p$ is JUMP. In a bottom-up pass the actual representation used (ISREP) is propagated from value producers to value consumers and conversions are introduced where necessary. Conversions are necessary whenever ISREP and WANTREP differ. Returning to the **if** example, suppose that the WANTREP for the entire expression is NUM and that the ISREPs for $x$ and $y$ are NUM and PTR. In this case, a conversion is inserted to convert the value produced by $y$ from pointer representation to number representation. Introduction of variables clutters the tree-walking nature of this algorithm, and requires the solution of simultaneous equations to obtain the best answer. In practice, heuristic solutions are good enough.

When a number must appear in pointer format (if it is used as an argument to a subroutine) allocating storage on a stack (or "push down list", or "PDL") is faster than allocating storage from the heap. Opportunities to perform stack allocation are detected with tree walks from top to bottom and bottom to top. From the top down, expressions' "PDLOKP" flags are set if their consumers will accept a PDL number. If a PDL number is allowed, then the value of PDLOKP is the expression "authorizing" the use of a PDL number. For example, in *(***if** *p x y)* the value of PDLOKP passed to the entire expression is passed on to $x$ and $y$, and the **if** authorizes the use of a PDL number in $p$. The bottom-up phase identifies those expressions that are able to produce a PDL number by setting the flag PDLNUMP. When PDLOKP and PDLNUMP are both true, and the desired representation (WANTREP) is a pointer, and the generated representation (ISREP) is a number, then the allocation is performed on the stack. Note that a procedure must ensure (at run time) that an argument is heap-allocated before it performs an unsafe operation on the argument.

### 4.1.3 Variables

Jones and Muchnick [MJ76] discuss the use of flow analysis for automatic choice of storage discipline for variables in Tempo. For each variable, their analysis determines those points in the program where the variable requires storage; that is, it determines the live range of the variable. Using this information it is possible to determine where storage for a variable must be allocated and where it may be freed.

Optimization of variable allocations is also performed in compilers for Scheme, ML, and Russell. Some implementations for these languages provide lexical scope by copying bindings into function objects, but introduce another level of indirection to permit side-effects to variables. That is, the activation record contains a pointer to an "assignable cell". The assignable cell, in turn, contains a pointer to the storage actually allocated for the object (for example, a **cons** cell). Assignment to a variable changes the contents of the assignable cell, not the stack frame. When a function object using a lexically bound name is created, the pointer to the assignable cell is copied into the function object. Assignments to the lexical variable within the new function thus affect the value of the variable in the enclosing function. This extra level of indirection requires another object allocated on the heap for the assignable cell.

## 4.2 Overwriting allocation

Hudak and Bloss [HB85] describe methods for detecting opportunities to overwrite array storage in a functional language implementation. Their approach assumes the existence of an operation $upd(a, i, x)$ that takes as input an array $a$, an index $i$ and some value $x$ and produces a "new" array whose $i$th element has value $x$ and whose other elements are identical to the corresponding elements of $a$. The language is functional; *upd* does not alter $a$'s value. Implementing *upd* in a straightforward requires the allocation and initialization of another array; this is often inefficient.

Their analysis is performed on a functional flow graph. Values "flow" through edges of the graph from nodes (which produce values) to other nodes (which also consume values). To describe a node $u$ performing an *upd* on the results from nodes $a$, $i$ and $x$, they write $u : upd(a, i, x)$. For each node $a$, $producers(a)$ is the set of nodes which might generate the *object* appearing at

$a$ and *consumers*$(a)$ is the set of nodes which might use the object produced at $a$. Note that the definitions of these two functions depends upon the implementation of the language; if the identity function *id* is implemented in such a way that it copies (creates a new object) its input, then *producers*$(a : id(x))$ will always contain $a$ and only $a$. If the output of *id* is the input object, then *producers*$(a : id(x))$ is *producers*$(x)$. Hudak and Bloss give recursive definitions for *producers* and *consumers* in their paper.

The sets *after*$(u)$, *needed-by*$(u)$, and *leading-to*$(u)$ model the strictness properties of the language. *After*$(u)$ contains the nodes that might be evaluated after $u$'s evaluation. *Needed-by*$(u)$ contains those nodes whose values are (definitely) needed to evaluate $u$. After $u$ has been evaluated, every node in *needed-by*$(u)$ has also been evaluated. *Leading-to*$(u)$ contains those nodes that must have been evaluated before "arriving at" $u$'s evaluation. Notice again that these functions depend upon the language's implementation, not just its semantics. For example, for the node $a : if(p, f, g)$, *needed-by*$(a)$ includes $p$ because $p$'s value is definitely required before the *if* can be evaluated. If evaluation is not aggressive, *leading-to*$(f)$ contains $p$ because the result of evaluating $p$ is needed to decide whether $f$ or $g$ should be evaluated; under an aggressive evaluation policy, $f$ and $g$ might be evaluated before or concurrently with $p$'s evaluation, and thus it is not possible to say that $p$'s evaluation must precede $f$'s. *After*$(p)$ contains $f$ and $g$ because one of those might be evaluated after evaluating $p$.

Given these functions describing evaluation and data flow in the program, define the set $C$ for an update node $u : upd(a, i, x)$ by

$$C = \bigcup \{consumers(s) | s \in producers(a)\}.$$

$C$ contains other potential uses of the value $a$. For $u$, the conflict set *conflicts*$(u)$ is

$$(C \cap after(u)) - leading\text{-}to(u) - needed\text{-}by(u).$$

That is, *conflicts*$(u)$ is {other potential uses of $u$} intersected with {those nodes which might be evaluated after $u$}, minus {all the nodes that must be evaluated before evaluating $u$}.

Given $u : upd(a, i, x)$, *conflicts*$(u)$, and arguments $f_{in_i}$ to the function $f$ containing $u$, if

$$conflicts(u) = \emptyset \wedge \nexists f_{in_i} \in producers(a)$$

then $u$ may be computed by modifying $a$ in place. Even when this condition is not satisfied, it may be possible to evaluate $u$ in-place. Fixing a particular evaluation order (consistent with the language's semantics) may alter *after*$(u)$ and *leading-to*$(u)$ in such a way that *conflicts*$(u)$ becomes empty. Interprocedural analysis may provide better information about arguments to the enclosing function $f$ (in the absence of other information, one must assume that they are shared). Reference counts may be attached to the values produced by nodes in *producers*$(u) \cap$ *producers*$(z)$, where $z \in$ *conflicts*$(u)$. This is called "optimized reference counting" because it is only used where some benefit is possible; the overhead of reference-counting all objects is avoided.

## 4.3 Assisting the garbage collector

Barth [Bar77] describes an optimization for use in reference-counting systems. Here, the aim is to assist (speed up) the garbage collector by removing canceling pairs of reference count adjustments.

The setting for this optimization is Deutsch and Bobrow's garbage collector for the Interlisp system [DB76], a hybrid of reference counting and marking garbage collection. References from heap objects to heap objects are counted, but references from program variables (originating in activation records) are not. The counts themselves are not stored in the objects; instead, two tables are maintained. The Zero Count Table (ZCT) identifies objects with a count of zero, and the Multiple Reference Table (MRT) identifies objects with a count larger than one. If an object is only referenced once, then it appears in no table. Updates to the tables are modeled as transactions; these are ALLOC, REF and DEREF. ALLOC enters a newly created object into the ZCT. REF and DEREF perform the obvious adjustments to ZCT and MRT to reflect changes in the reference counts. Collection of objects is not immediate; periodically a collector sweeps the stack and collects all objects that are in the ZCT but not referenced from the stack.

Pairs of transactions that cancel are ALLOC-REF, REF-DEREF, and DEREF-REF. Barth describes the removal of ALLOC-REF transaction pairs. To do this, he traces the flow of newly allocated objects through variables in activation records.

For each allocation site, he computes a modified depth-first numbering of the flow graph such that no node (except the first) is numbered until all of its

parents (predecessor nodes) have been numbered [Tar72]. (It is unclear how Barth intends to treat irreducible graphs, or those in which the allocation site does not dominate all of its descendants.) For each edge E in this graph (of numbered nodes) he computes the bit vectors E.DS (Definitely Set) and E.MS (Maybe Set) identifying those variables that may reference the new object. For each node, the incoming DS sets are intersected to yield the outgoing DS set and the incoming MS sets are unioned to yield the outgoing MS set. If the node is an assignment to a variable, then the outgoing sets are adjusted to reflect the change.

After computing DS and MS for each node in the graph, the algorithm visits the nodes again. In this pass, it propagates "allocation transaction owed" markers through the graph. Initially, edges leaving the allocation are marked. For each node visited, do nothing if any incoming edge is unmarked. Otherwise (all edges are marked), check for cancellation and changes to the marking. Cancellation can occur in two situations. If the node is an assignment, the left-hand side is a heap reference, and the right-hand side is a member of all the incoming DS sets, then an ALLOC-REF cancellation is possible. All the incoming markers are erased and the outgoing edges are unmarked. If the node is an assignment and the outgoing MS and DS sets are empty, then the cell is no longer accessible. All the incoming markers are erased, the outgoing edges are unmarked, and in-line code to free the cell is generated (as opposed to entering the cell into the ZCT). If no cancellation occurs, the markers can be propagated provided two conditions are satisfied. The DS sets on the outgoing edges must be non-empty (there must exist some reference to the object), and the assignment cannot be an assignment whose left-hand side is a heap address and whose right-hand side variable is in an MS set but not in a DS set (that is, there can be no possibility that the assignment will create a heap reference). When this is the case, the incoming markers are erased and the outgoing edges are marked.

After the second visit is complete, insert an ALLOC transaction on each marked edge. This will settle any "unpaid debts" remaining, and take care of edges out of the dominated subgraph (they will be marked). Barth describes another possible optimization in which transactions are batched, but gives no algorithm. Here, a series of REF or DEREF transactions is replaced with one REFn or DEREFn transaction.

## 4.4 SETL

Storage optimizations of all three types listed above were proposed for the SETL compiler [Sch75]. SETL [KS75] does not provide the ability to return lexically scoped functions, so the compiler may safely allocate activation records on the stack, but aggregate objects in general (tuples and sets) are treated as values and reclaimed with a garbage collector, providing ample opportunity for overwriting and stack allocation optimizations. In addition, Schwartz proposes analysis to discover groups of objects that are linked together, and thus must be freed together. Such groups are allocated in one batch to reduce the overhead to the allocator and the garbage collector.

SETL is superficially an Algol-like language. It has Algol-like control structure, variables, assignment, functions and subroutines. SETL, however, has a much richer variety of data types and operations on those types than is usual for Algol-like languages. Non-primitive types and operations on those types are based on sets and operations on sets. In addition, SETL is a value language, not a pointer language, and uses a garbage collector to reclaim unused storage.

In the language implementation non-atomic values are represented with pointers to storage and values are assigned by copying pointers, thus sharing the storage that is addressed by the pointer. Shared storage may not be altered because that would violate SETL's value semantics. One goal of the compiler, then, is to detect situations in which an updating operation (set insertion and deletion, altering tuple members) is applied to a value whose storage is not shared, and to introduce an in-place update operation.

In the analysis of a SETL program, each occurrence of a variable $x$ in which its value is used is called an *ivariable*. Similarly, each occurrence of a variable $x$ in which it is (re)defined is called an *ovariable*. The program itself is partitioned into *basic blocks*, which are the nodes in the program's *control flow graph* $G$. A (directed) edge from one block $b_1$ to another block $b_2$ in $G$ represents a flow of control from the end of block $b_1$ to the beginning of block $b_2$. An ovariable $o$ and ivariable $i$ are said to be *chained together* if both $i$ and $o$ are occurrences of the same variable $x$[2] and if there exists a path through $G$ from $o$ to $i$ containing no assignments to $x$. If $i$ and $o$ are chained together, then $i \in du(o)$ and $o \in ud(i)$. *Ud* and du are the use-to-definition and definition-to-use maps of traditional data-flow analysis. An ovariable $o$

[2]Note that $i$ and $o$ are variable occurrences, not variables themselves.

is said to be *dead* when $du(o) = \emptyset$.

### 4.4.1 Overwriting in SETL

Given an ivariable $i$ with value $p$ used in an updating instruction *op*, it is desirable to discover that $p$ is not shared and that *op* may be performed in-place. To do this, the compiler constructs $l(i)$, the set of ovariables with values possibly incorporating $p$[3]. If every ovariable $o$ in $l(i)$ is dead[4] at *op*, then destructive use of $p$ is possible.

An approximation (upper bound) for $l(i)$ is computed in two steps. In the first step, the functions $crthis(i)$ and $crpart(o)$ are calculated. The set $crthis(i)$ is the set of all ovariables whose evaluation[5] can produce an object that becomes the value of $i$. This is similar to the set $producers(x)$ used by Hudak and Bloss in the paper described above. The set $crpart(o)$ is the set of all ovariables whose evaluation can produce an object that becomes a part of the current value $v$ of $o$; that is, the object becomes $v$, or becomes a part of a member of $v$.

$$crthis(i) = \bigcup_{o \in ud(i)} crthis(o) \tag{4.1}$$

$$crmemb(i) = \bigcup_{o \in ud(i)} crmemb(o) \tag{4.2}$$

$$crpart(i) = \bigcup_{o \in ud(i)} crpart(o) \tag{4.3}$$

$$crthis(o) = \begin{cases} crthis(i_1) & \text{if } o \leftarrow i_1 \\ \bigcup_{j \in crmemb(i_1)} crthis(j) & \text{if } o \leftarrow\ni i_1 \\ \{o\} & \text{otherwise} \end{cases} \tag{4.4}$$

[3]Schwartz uses the phrase "current value". I believe this means "value, given that the program is about to execute *op*".

[4]By "dead at a point", I assume that Schwartz means that if the instruction $x_o \leftarrow x_i$ were inserted (where $x$ is the variable occurring as an ovariable elsewhere in the program), then either $du(x_o) = \emptyset$ or $ud(x_i) = \emptyset$; that is, no ovariable occurrence of $x$ can be chained to this point, or any occurrence that does reach this point is unable to reach any ivariable occurrence from this point.

[5]"Evaluation of an ovariable" means calculation of the value assigned to it.

$$crmemb(o) = \begin{cases} crmemb(i_1) & \text{if } o \leftarrow i_1 \\ \bigcup_{j \in crmemb(i_1)} crmemb(j) & \text{if } o \leftarrow \ni i_1 \\ \{i_1\} & \text{if } o \leftarrow \{i_1\} \\ crmemb(i_1) \cup crmemb(i_2) & \text{if } o \leftarrow i_1 \cup i_2 \\ crmemb(i_1) & \text{if } o \leftarrow i_1 - i_2 \\ \emptyset & \text{if } o \leftarrow \text{data} \\ \emptyset & \text{otherwise} \end{cases} \tag{4.5}$$

$$crpart(o) = \begin{cases} crpart(i_1) & \text{if } o \leftarrow i_1 \\ \bigcup_{j \in crmemb(i_1)} crpart(j) & \text{if } o \leftarrow \ni i_1 \\ \{o\} \cup crpart(i_1) & \text{if } o \leftarrow \{i_1\} \\ \{o\} \cup crpart(i_1) \cup crpart(i_2) & \text{if } o \leftarrow i_1 \cup i_2 \\ \{o\} \cup crpart(i_1) & \text{if } o \leftarrow i_1 - i_2 \\ \{o\} & \text{otherwise} \end{cases} \tag{4.6}$$

In the second step, the set $exsinthis(i)$ is calculated. This is the set of all instructions that might have been executed between the creation of the pointer $p$ and the time it becomes the value of $i$. The calculations of *crthis*, *crpart* and *exsinthis* are accomplished through the solution of a great number of recursive equations. The equations are monotonic in their right-hand sides, so they may be solved iteratively.

Given these functions, $l(i)$ can be calculated. The operations that might have been executed since $p$ was created form the set $P_i = exsinthis(i)$. The operations that might have created $p$ form the set $crthis(i)$. Given a subpart $\overline{P}$ of the program $P$, define a relativized *crpart* function $crpart_{\overline{P}}(o)$ by solving equations 4.1 through 4.6 with $ud(i)$ replaced by $ud(i) \cap \overline{P}$. If

$$crpart_{P_i}(o) \cap crthis(i) \neq \emptyset$$

then $o$ is in $l(i)$. The above formula is equivalent to

$$o \in {crpart_{P_i}}^{-1}(crthis(i))$$

giving the condition for destructive use (quoting Schwartz [Sch75, p.178]):

> Let $i$ be an ivariable of a SETL program $P$ and let
>
> $$P_i = exsinthis(i).$$
>
> Then if every $o$ belonging to the set
>
> $${crpart_{P_i}}^{-1}(crthis(i))$$

is dead immediately before $i$ is used (where in describing an ovariable as dead we ignore its immediately following use in the operation containing $i$) then $i$ may be used destructively.

### 4.4.2 Stack allocation in SETL

Schwartz describes an analysis that will detect (some) opportunities to allocate objects from the stack instead of from the heap. He proposes stack allocation at an interval granularity; that is, any object whose lifetime is contained within the interval should be stack-allocated and freed at edges leaving the interval. Freeing such objects is very inexpensive, because the stack pointer is just restored to its value on interval entry.

Given an interval $I$, build the set $\Delta$ of ovariables of $I$ having no uses outside $I$. From $\Delta$, form $\Delta'$ by removing from $\Delta$ variables belonging to $crpart(i)$ for some $i$ not in $I$. That is, remove from $\Delta$ ovariables that are incorporated into values whose lifetimes are not contained within $I$. Finally, notice that the introduction of destructive updates may extend the lifetimes of objects (not values) outside of $I$. To account for this check the ovariable $o$ of each destructive operation in $I$. If $o$ does not belong to $\Delta'$, then let $i$ be the ivariable modified in the destructive operation and remove from $\Delta'$ all ovariables $o$ in $crpart(i)$. Repeat this process until $\Delta'$ stops shrinking. The variables remaining in $\Delta'$ can be allocated on a stack that is popped at exits from the interval $I$.

### 4.4.3 Area allocation in SETL

Another proposed transformation assists the garbage collector by grouping several objects into a single storage object, or area. This helps the garbage collector in several ways. Storage for all the grouped objects is obtained in a single large request instead of many smaller requests. When searching for reachable objects in a collection, references between objects within the area are not traversed, and all of the objects are marked live when the area is marked live. Finally, cyclic reference graphs among the objects in the area can prevent collection by some techniques (reference counting) if the objects are allocated individually, but will not if all members of the cycle are contained in the area. This compiler optimization is similar to a hand-optimization proposed by Bobrow [Bob80].

### 4.4.4 Later work

In later work on the SETL compiler, the stack and area allocation algorithms are not mentioned. The copy (overwriting update) optimization, however, reappears in several forms.

Schwartz [Sch76] describes a "shadow variable technique" that appears to be very powerful. In this analysis, the "share bit" used in the implementation to detect overwriting opportunities at run-time is made explicit with a "shadow variable". For a variable $y$, the shadow variable *yshare* undergoes the following assignments:

1. For a simple assignment $x \leftarrow y$, set $yshare = 1$ unless $y$ is dead, and set $xshare = 1$ (unless $y$ is dead and $yshare = 0$?)

2. For an assignment of the form $x \leftarrow \langle expr \rangle$, set $xshare = 1$ if $\langle expr \rangle$ is a value-retrieving expression, and set $xshare = 0$ if $\langle expr \rangle$ is a value-creating expression.

3. When $y$ is incorporated into a compound object set $yshare = 1$ unless $y$ is dead.

For the moment, we will ignore the possibility of procedure calls, although Schwartz treats that case in his newsletter.

At compile time, possible destructive updates to $y$ are treated as uses of *yshare*, since this is in fact the case. Constant propagation algorithms can detect cases in which overwriting is always possible, and places where it is never possible (and thus the value of the share bit may be ignored). Live/dead analysis on the share bits can also detect places where the share bits are no longer needed, and thus the bits need not be updated.

Schwartz further proposes an algorithm that helps guide the movement of testing and copy operations out of loops. In figure 4.3 the first assignment to $s$ establishes $s$ as a shared value. In the following loop $s$ is updated once on each iteration, requiring code within the loop to test the share bit and perform the copy[6].

To remove these operations from the loop Schwartz uses an interval-based technique that classifies the use of each share-bit variable into one of four categories; definitely zero, definitely one, inherently indefinite, and indefinite

[6]Note that this copy operation will only be performed once.

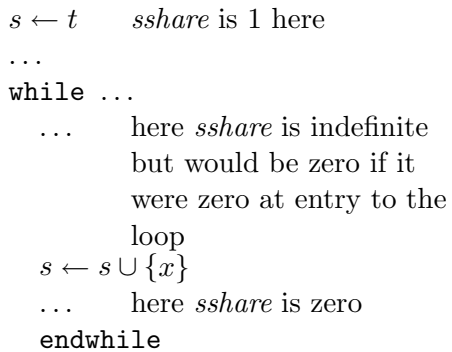



Figure 4.3: Removing copying from within loops

but would be zero if it were zero at entry to the interval containing the occurrence. For each occurrence in the fourth category find the largest interval in the full-program graph derivation sequence such that that occurrence remains in the fourth category. At the entrance to that interval make a copy of the value corresponding to the occurrence, ensuring that the share bit is zero and permitting the removal of share bit test and copy operations from within the loop.

The SETL optimizer [Cou85] uses this technique in conjunction with an improved value-flow analysis method [FSS83] to avoid copying objects. The value-flow analysis phase performs copy-elimination analysis for each destructive use $DU$. For each $DU$, the optimizer calculates the set of all preceding variable occurrences $VO$ whose variable might contain a pointer to the value modified at $DU$.

The first step of this analysis is a backward pass to find all variable occurrences $VO$ from which the value $VAL$ at $DU$ might have been obtained. These occurrences include those whose values are the same as $VAL$, and those that might contain $VAL$. The information propagated backwards consists of pairs $[VO, R]$ where $R$ is a value containment relationship that holds at $VO$. A value containment relationship is any composition of elementary containment operations, such as "is a set member", "is a tuple member", "is the $n$th tuple member". In the SETL compiler, $n$ is less than 10, and any containment relationships that require more than 10 elementary operations are converted to the worst case "is any member".

After finding all possible sources $VO$ for the value $VAL$, the algorithm

propagates value-containment relationships forward from the *VO* to all other occurrences that might contain *VAL*. After this step, all occurrences that might contain *VAL* have been located. Next, the algorithm forms the set of all variable definitions that can reach *DU*. If none of these definitions is live at *DU*, then the update may be performed without copying *VAL*.

## 4.5 Discussion

Of the work described above, that done in the SETL optimizer is the most extensive. Hudak and Bloss, Barth, and Muchnick and Jones do not address the problems posed by nested objects. Hudak and Bloss analyze a non-strict language without nested aggregates to detect opportunities for in-place updates. Non-strictness makes the analysis harder; they get their best results when that feature is removed. Muchnick and Jones only consider the lifetimes of variables, and do not consider the lifetimes of values. Barth's analysis traces the lifetimes of values as long as they are only referenced from variables. The S-1 Common Lisp compiler traces the lifetimes of numbers and functions, and handles the case of "containment" within functions (i.e., the code for one function calls another function), but does not treat any other form of containment.

As described by Schwartz [Sch75], the SETL optimizer was intended to improve four aspects of storage allocation. The transformations proposed were copy avoiding, avoiding garbage collection bookkeeping operations, stack allocation and area allocation. Of these, only the first appears in the SETL optimizer [Cou85] as of 1985. However, the storage for many temporary values is allocated on the stack, not the heap, and the analysis to avoid share-bit bookkeeping is in place, though the transformation is not yet performed[7].

Several things limit the analysis performed by the SETL compiler. The containment relationships used in value flow analysis are more general than a bit-vector approach, but they are still limited by their ability to express only one relationship at a time and by the bounds imposed to avoid infinite containment relationships. These restrictions can be relaxed.

The interval-based stack allocation algorithm is also limited in that it can only be applied to the first interval reduction of the graph $G_0$ to $G_1$. The interval method relies on the fact that intervals are single-entry, and

[7] I think that the SETL project had bigger fish to fry, so to speak.

frees all stack storage allocated within the interval by restoring the stack pointer to its value on entry to the interval. The difficulty in performing stack allocation within the intervals of $G_1$ is that no stack object in a $G_1$ interval $I_1$ may be allocated within a $G_0$ interval $I_0$; if it is, then it will be deallocated on exit from $I_0$. Such an object must be allocated before entering $I_0$. This is not always possible, because the object's size may not be known at that point. Stack allocation within intervals can also cause problems if the interval contains a loop. This problem will be addressed in a later chapter.

In all of the work described above, a result allocated by a function (as opposed to extracted from an object existing before the function call) is always obtained from garbage-collected memory. This approach is clearly necessary without interprocedural analysis; if the result is always placed in static memory or on the stack, then the calling routine must copy it to another location if it is not used immediately, and the called routine must always copy its result to the standard location, even it if it is a part of an existing value. Ruggieri addresses this problem in her thesis [Rug87].

# Chapter 5

# Improved containment analysis

The analysis described in the previous chapter can be improved. Here are some improvements and elaborations to what has gone before. The previous chapter described methods used in SETL and Lisp compilers to determine when one value is contained within another. The SETL compiler uses this analysis to help find opportunities for performing updates in place. This information can also be used to avoid allocating memory from the garbage collected heap. Unfortunately, this analysis is limited in two ways. For each definition reaching a use, the SETL compiler can only maintain a single value containment relationship. If more than one relationship is possible, then a less specific relationship must be used. The containment relationships are also finite; if a relationship is composed of more than 10 elementary containment relations, then it is replaced with an "any member" relation. This hides any information about the internal structure of the contained object. Lisp compilers only apply containment analysis to closures and floating point numbers.

## 5.1 Storage containment relationships

My improvement to this is a *storage containment graph*, or SCG. The storage containment graph is designed to approximate containment relationships holding between storage allocated at various definition points. All storage arising from a particular definition is treated in the same way; this places a bound on the graph's size. The nodes in an SCG correspond either to variable definitions and modifications in the program or to the storage as-

sociated with those definitions. The two sets of nodes are called *def* nodes and *store* nodes. The edges in an SCG correspond to selector operations and to the relation between definitions and storage. These two sets of edges are called *selector* edges and *store* edges. Selector edges are labeled with selector operations.

In an SCG, two nodes $d$ and $e$ are joined by a selector edge $s : \langle d, e \rangle$ when an object created at $e$ can be extracted from an object created at $d$ by applying an $s$ selection operation to the object created at $d$. If there is no such edge, then no object created at $e$ can be extracted from a object created at $d$ by applying $s$ to that object.

Store nodes have significance only in relation to other store nodes; they are the same, or they are not. Each store node represents a distinguishable unit of storage. Each store node is associated with an *allocation node*, which is a def node corresponding to a definition (site) that places its result into allocated storage.

Because allocation of new storage always generates an unshared unit of storage, no two allocation nodes in the SCG have edges to the same store node. If two def nodes $d$ and $e$ have edges to the same store node $\sigma$, then objects resulting from $d$ and $e$ might both "use" the same storage. If there is no store node $\sigma$ such that $d$ and $e$ both have edges to $\sigma$, then objects created at $d$ and $e$ will not use the same storage. The scheme presented here does not distinguish between storage produced from different executions of the same allocation node.

The distinction between store nodes and definition nodes reflects an aspect of implementation on a conventional machine; there is no one-to-one correspondence between storage and values. For example, after an overwriting optimization the overwritten storage lives on, but (parts of) the old value do not. This distinction also makes clear the differences between copy-by-value and copy-by-reference.

## 5.2 Using the SCG

The SCG is designed to answer questions concerning storage containment and sharing. Some of these questions are:

- Given a definition $d$, what storage might be contained within objects resulting from $d$?

- Given a storage node $\sigma$ (representing storage allocated at a particular definition), what definitions might produce an object containing $\sigma$?
- Given a definition $d$, what store nodes might be overwritten by a destructive update to an object resulting from $d$? (This circumlocution means: "If values are pointers, to what storage might values resulting from $d$ point?")

"Contain" means "be implemented with", or "access via a chain of pointer references"; the essential notion is that if the abstract value associated with an object $a$ is changed when another object $b$ is changed, then $a$ contains $b$[1]. Containment is determined by a particular implementation, not by language semantics.

These questions translate into path questions on the SCG. The question "what storage can the results of $d$ contain?" translates into "what store nodes are on paths from $d$?" By assumption, if there is a store edge from $d$ to some store node $\sigma$ then $d$ can contain $\sigma$. If there is an $s_1$-edge from $d_1$ to $d_2$, then there might be a pointer from an object $o_1$ resulting from $d_1$ to an object $o_2$ resulting from $d_2$. Store nodes adjacent to $d_2$ represent additional storage that might be contained in $o_1$, as do other store nodes contained in $o_2$ (reachable via paths through $d_3$, $d_4$, ..., $d_n$, connected with edges $s_2$, $s_3$, ...,$s_{n-1}$). To see that the abstract value associated with an object can change if the storage adjacent to one of the $d_i$ on the path changes, consider the possible results of applying the selectors $s_1$ through $s_{n-1}$ to $o_1$. Application of $s_1$ may (not must; we strive for the best approximation we can afford) yield $o_2$, and application of $s_2$ to $o_2$ may in turn yield $o_3$, and so on until $o_n$ is produced. A change to the storage for $o_n$ changes the associated value, thus changing the (possible) result of $s_{n-1} \circ s_2 \circ \ldots \circ s_1$ applied to $o_1$. Thus, the value of $o_1$ is changed and it contains the storage (represented by the store node $\sigma$) reachable through the path $(d_1, \ldots, d_n, \sigma)$.

The second question "what definitions produce results that can contain $\sigma$?" is just the converse of the first, and can be answered by searching for nodes on paths to a given store node $\sigma$.

The last question "what storage might be affected by a destructive update to an object resulting from a definition $d$?" translates into "what store nodes

[1]I am digressing, but I am also bothered that the notion of "containment" is so vague. I am also bothered by the continuing confusion of "abstract value" and "implemented value" (or "object", as I try to say).

are adjacent to $d$ in the SCG?" (I wish to defer the question of "what is a destructive update?" for now.)

These questions are asked when analyzing storage lifetimes to convert heap allocations to stack allocations and to introduce destructive update operations. The first optimization is useful in both value-assignment and side-effect languages; the second is useful in value-assignment languages[2].

To convert a heap allocation into a stack allocation (and be able to reclaim the storage at some future time) it is necessary to place a bound on the lifetime of the storage resulting from the allocation. It is generally useful to find the smallest set of nodes in the program flow graph at which the storage might be accessed. The following three steps compute an approximation to the lifetime of storage allocated at a definition node $d$:

1. There will be a single store node $\sigma_d$ adjacent to $d$ through a store edge. This is so because $d$ is an allocation node.

2. Find the set $\Delta$ of all def nodes containing $\sigma_d$, i.e., all def nodes on paths to $\sigma_d$.

3. Using simple use-definition chains, find all uses of these definitions. The nodes and edges on paths from definitions in $\Delta$ to their uses form the live range of the storage allocated at $d$.

Replacing a copying update with a destructive update requires a grubbier analysis. An updating operation $d : x.s \leftarrow y$ can be converted to reuse storage associated with $x$ if certain conditions are met. The compiler must guarantee that any storage associated with $x$ is not used past this operation. This can be determined by the following four steps:

1. Using simple use-definition chains, find all definitions for $x$.

2. Using the SCG, find all store nodes adjacent to these definitions.

3. Using the SCG, determine the lifetimes of all of these store nodes. This is described above.

[2]I use *value-assignment* to mean those languages in which $x \leftarrow y$ has the effect of assigning $y$'s *value* to $x$ as opposed to assigning a pointer to $y$'s value to $x$. A *side-effect* language is one in which the pointer is assigned; thus, a change to $y$'s value has the (side-) effect of changing $x$'s value. I use "value-assignment" instead of "functional" because the implementation of lazy functional languages often involves storage side-effects.

4. If any of these definitions is live along an edge leaving $d$, then the update cannot be performed destructively.

The introduction of an overwriting operation changes the SCG; this will be explained below with the construction of the SCG.

It appears that paths through the SCG will be traced frequently in lifetime analysis. If the purpose of the lifetime analysis is to find places to overwrite storage, then the form of the SCG cannot be changed because the SCG may change. An overwriting optimization reuses storage, so the lifetime of a containment of a store node can change. For stack allocation optimizations, however, the SCG will not change and it is possible to perform path compression as paths are traced in the SCG.

## 5.3 Constructing an SCG for a language with value-assignment semantics

This construction of an SCG assumes the existence of simple use-definition chains, so that given a variable and a program point, it is possible to find the possible definitions of that variable. A modification to an aggregate value is treated as a definition in the construction of these use-def chains.

There are five cases (ignoring procedure calls) to consider in the construction of a storage containment graph. These are:

$d : x \leftarrow new[\,]$
: Here (definition $d$) $x$ is assigned a newly allocated piece of storage that is not initialized, or that is initialized without containing other storage.

$d : x \leftarrow new[y_1, \ldots, y_n]$
: Here $x$ is assigned a newly allocated piece of storage that is initialized using the variables $y_1$ through $y_n$.

$d : x \leftarrow y$
: Here $y$'s value is assigned to $x$.

$d : x \leftarrow y.s$
: Here the value obtained by applying the $s$ selector to $y$ is assigned to $x$. No storage is allocated.

$d : x.s \leftarrow y$

Here the value accessible by applying the $s$ selector to $x$ is replaced with $y$'s value. If the update is "in-place", then no new storage is allocated here; if the update is "copying", then new storage is allocated. In-place updates are allowed in a value language when it can be shown that side-effects will not occur.

### 5.3.1 Detailed description

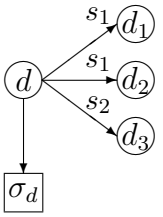
The operations described below must be applied to the SCG until no more edges or nodes can be added. All of the operations on the SCG never decrease the size of the SCG and are monotonic; if $A \subseteq B$, then $op(A) \subseteq op(B)$, and $A \subseteq op(A)$. Since, for any given program, there is an upper bound on the number of edges which can be added the construction will always terminate. The effects of procedure calls will not be discussed in depth here.

At a definition $d$ where a piece of storage is allocated, a new store node $\sigma_d$ is created and a store edge $\langle d, \sigma_d \rangle$ is added from $d$'s def node to the new store node $\sigma_d$. Note that $\sigma_d$ is distinct from any other store node in the SCG.

$$d : x \leftarrow new[\,] \Longrightarrow \textcircled{d} \longrightarrow \boxed{\sigma_d}$$

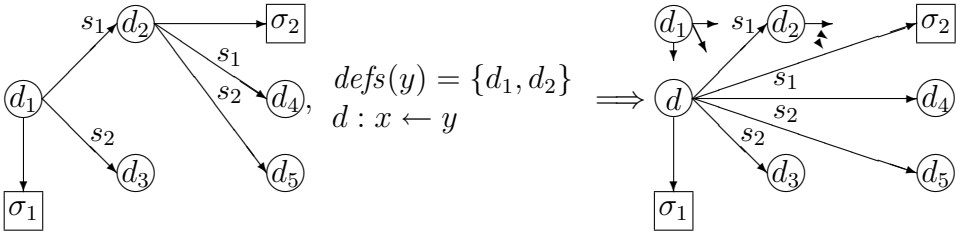
At a definition $d$ where a piece of storage is allocated and initialized to contain other storage, a new store node $\sigma_d$ is created and a store edge $\langle d, \sigma_d \rangle$ is added from $d$'s def node to the new store node $\sigma_d$. Selector edges leaving $d$ are also added. Each initializer $y_i$ corresponds to a selector $s_i$, and each initializer (variable) has a set of possible definitions that reach $d$. For each initializer $y_i$, add selector edges labeled $s_i$ to the def nodes whose corresponding definitions reach $y_i$.

$$\begin{array}{l} defs(y_1) = \{d_1, d_2\} \\ defs(y_2) = \{d_3\} \\ d : x \leftarrow new[y_1, y_2] \end{array} \Longrightarrow$$

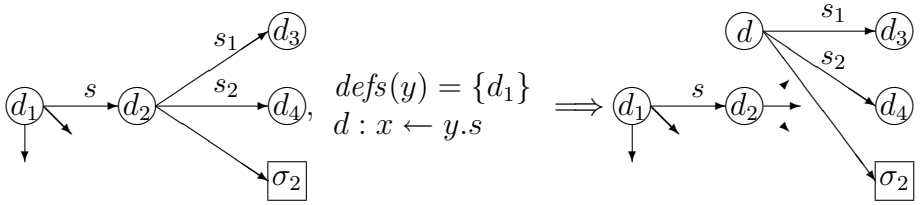



At an assignment, all edges leaving definition nodes for the right-hand side are copied to the assignment's definition node. The original edges are not shown in the result graph below for the sake of clarity; they have not

been removed from the SCG.




At a selection assignment statement ($x \leftarrow y.s$), we wish to make copies of all edges from nodes adjacent to def nodes for $y$ through $s$-labeled selector edges. That is, there is a set of nodes corresponding to definitions for $y$. There are (ought to be) selector edges labeled with $s$ leaving those nodes. From the nodes at the ends of those edges, copy all leaving edges to the def node for the selection assignment statement.

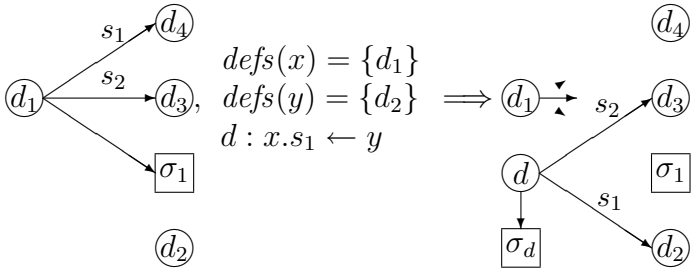



The case of value modification is more interesting. In a value language a "modification" usually allocates new storage for a copy of the original value, but this is not always necessary. Some selectors are "imprecise" (for example, array indices are imprecise selectors because their exact values are usually unknown at compile time); the effects of imprecise selection are introduced here.

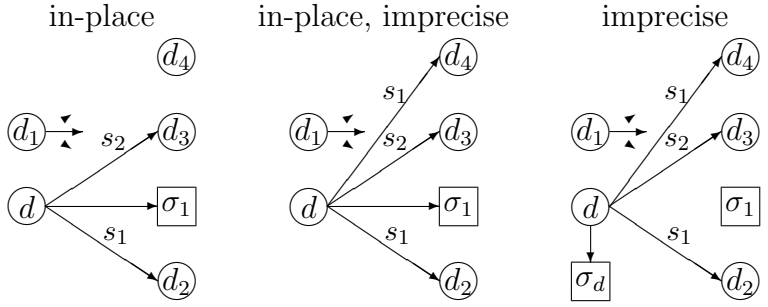
Given an assignment $d : x.s \leftarrow y$, copies of all edges leaving def nodes for $x$ are copied to the def node $d$, except for those labeled with the selector $s$. If $s$ is an imprecise selector, then the edges labeled $s$ *are* copied, because assignment through an imprecise selector is not guaranteed to "kill" values accessible through that selector. In either case, selector edges labeled $s$ are added from $d$ to all def nodes for $y$.

If the modification is in-place, then any storage that might have been used by definitions for $x$ might also be used by $d$. That is, edges are added from $d$ to all store nodes adjacent to definitions for $x$. If the modification is copying, then a new store node $\sigma_d$ is introduced. This is the only store node adjacent to $d$ if the update is non-overwriting. Because the language

has value-assignment semantics, in-place modifications are guaranteed not to have side-effects, and thus the SCG for a value language does not need to account for them. Correct modeling of side-effects will be discussed in a later section.




The diagram above shows the case of a copying update with a precise selector. The local results of the other three combinations are shown below.




### 5.3.2 Uniqueness of resulting SCG

Because the operations are monotonic and never decrease the graph[3], there is a unique storage containment graph regardless of the order in which the transformations are applied to the graph.

To see that this is so, note that only a finite number of transformations can be applied. These transformations are also locally confluent, and thus it follows by Newman's Lemma that they have the Church-Rosser property [PB85, Hue80, New42]. The transformations are locally confluent because they are monotonic, increasing, and finite.

**Proof** Suppose that there are two transformations $p_1$ and $p_2$ and a graph $G$ such that $p_1(G) \neq p_2(G)$. Let $G_1$ and $G_2$ initially be $p_1(G)$ and $p_2(G)$, and

[3]In fact, a transformation is applicable only if the result will be larger than the input.

let $l = 0$; the goal is to show that there exists a sequence of $l$ transformations on $G_1$ and $G_2$ such that are eventually equal. Suppose that there is some part $x$ of $G_2$ that is not in $G_1$; then update $G_1$, $G_2$ and $l$ by

$$\begin{array}{ll} l \text{ even} & l \text{ odd} \\ {G_1}' = p_2(G_1) & {G_1}' = p_1(G_1) \\ {G_2}' = p_1(G_2) & {G_2}' = p_2(G_2) \\ l' = l + 1 & l' = l + 1. \end{array}$$

The idea is that $G_1$ and $G_2$ are updated by alternating applications of $p_1$ and $p_2$; thus, it will always be true that there are $f$ and $g$ such that

$$\begin{array}{l} {G_1}' = f(p_1(G)) \text{ and } G_2 = f(G) \\ {G_2}' = g(p_2(G)) \text{ and } G_1 = g(G). \end{array}$$

Because the functions are monotonic and increasing, it is clear that ${G_1}'$ contains $G_2$ and that ${G_2}'$ contains $G_1$. Thus, whenever $G_1$ and $G_2$ differ we have an effective procedure to add the differing edges. Since we can only apply a finite number of transformations, $G_1$ and $G_2$ must eventually be equal. ∎

### 5.3.3 Correcting for overwriting update

If a copying update is converted to an overwriting update, this changes the lifetime of a piece of storage. Since this optimization uses the SCG, it is useful if the SCG can be modified to reflect the change, rather than re-calculated from scratch. The change in lifetime must be reflected in the SCG so that non-heap allocation optimizations and subsequent overwriting optimizations will be performed correctly. For a def node $d : x.s \leftarrow y$ where an overwriting update is introduced, perform the following operations:

1. Form the set $\Sigma$ of all store nodes adjacent to definitions for $x$ that reach $d$. None of these store nodes are live past $d$ because otherwise $d$ would not be a candidate for an overwriting update.

2. For all definitions $d'$ adjacent to the storage allocated at $d$, remove the store edge $\langle d', \sigma_d \rangle$ and add a store edge from $d'$ to each store node in $\Sigma$.

3. Remove $\sigma_d$.

After doing this, all definitions that previously were adjacent to the storage allocated at $d$ are now adjacent to the storage overwritten at $d$. Another way to do this is to form edges from $d$ to all store nodes in $\Sigma$, remove $\sigma_d$ and all edges to it, and iterate over the program definitions to repair the SCG. The results are the same.

### 5.3.4 Coping with incomplete information

Inevitably, some value definitions will be incomplete. Incomplete information is easily expressed in the SCG framework. In the worst case, nothing is known about a variable, not even its type (i.e., not even what selectors may be applied to it). This is expressed in the SCG as a single def node $\perp$ and a single store node $\sigma_\perp$. For each selector $s$ used in the program there is a selector edge from $\perp$ to $\perp$ labeled with $s$. Thus, any selection applied to $\perp$ yields $\perp$, and any assignment of $\perp$ will reference the store $\sigma_\perp$. Less incomplete information can also be expressed. If, for example, the program can be type-checked at compile time, then there can be a different $\perp_t$ for each type $t$. If a selector $s$ applied to an object $t$ always yields an object of type $u$, then the only $s$ edge leaving $\perp_t$ will end at $\perp_u$. Note that the bottom store nodes are distinct from store nodes allocated in the program because any allocation in the program is guaranteed to produce a piece of storage distinct from all existing storage. Differently typed bottom nodes get different store because it is assumed that two variables of different types will never simultaneously use the same piece of storage.

Figure 5.2 contains a sample pseudo-Lisp program to insert an object into a reversed list. Figure 5.1 shows the storage containment graph resulting from this program. The definition sites in the program are numbered, and the definitions reaching the input(s) for each definition site are also shown. The selectors are A and D[4]. The structure of the list $l$ is unknown, so its A and D edges are directed to $\perp$, and its store is $\sigma_\perp$.

## 5.4 What's really happening

So far I have only described the construction of SCGs for value languages. Also, so far I have implied the existence of a single SCG for the whole program. It happens that for a value language the information provided by a

[4]For *Address* and *Decrement*.

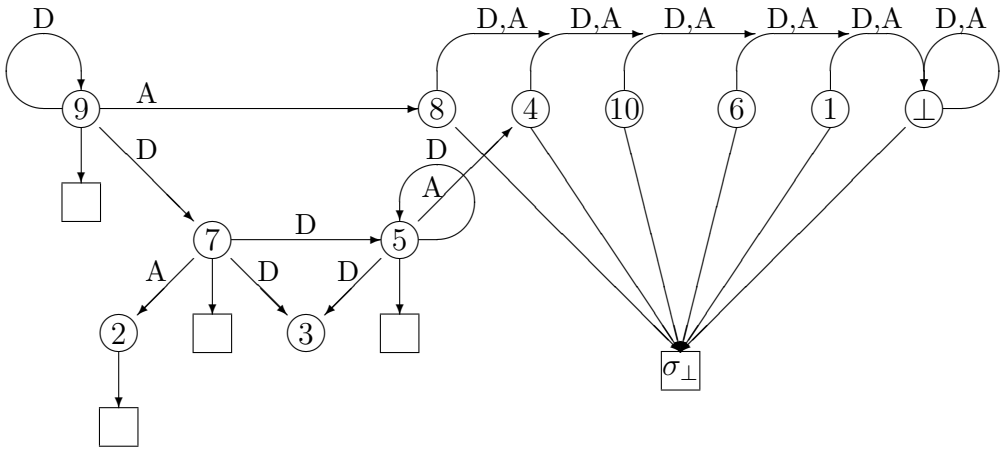



Figure 5.1: Value containment graph

| Def | Code | Reaching definitions |
|---|---|---|
| 1 | $l \leftarrow \ldots$ | |
| 2 | $a \leftarrow \ldots$ | |
| | $\ldots$ | |
| 3 | $l' \leftarrow$ nil | |
| | **while** $l \neq$ nil **and** $a < car[l]$ { | |
| 4 | $t_1 \leftarrow car[l]$ | $\{1, 6\}$ |
| 5 | $l' \leftarrow cons[t_1, l']$ | $\{4\}, \{3, 5\}$ |
| 6 | $l \leftarrow cdr[l]$ | $\{1, 6\}$ |
| | } | |
| 7 | $l' \leftarrow cons[a, l']$ | $\{2\}, \{3, 5\}$ |
| | **while** $l \neq$ nil { | |
| 8 | $t_2 \leftarrow car[l]$ | $\{1, 6, 10\}$ |
| 9 | $l' \leftarrow cons[t_2, l']$ | $\{8\}, \{7, 9\}$ |
| 10 | $l \leftarrow cdr[l]$ | $\{1, 6, 10\}$ |
| | } | |
| | $\ldots$ | |

Figure 5.2: Insertion in reversed list

single SCG is as good as the information provided by having one SCG for each definition site in the program. This is proved below.

Consider producing an SCG for each definition or use in the program. A program provides a set of definitions $\Delta$, store nodes $\Sigma$, and selectors $S$. Each store node in $\Sigma$ is associated with definition node $d$ in $\Delta$; this correspondence is indicated by labeling the store node as $\sigma_d$. For a store node $\sigma_d$, $d$ is the definition allocating $\sigma_d$. In the event that more than one store node is allocated at $d$, the nodes will be further subscripted to distinguish them; for example, $\sigma_{d1}$. This naming of store nodes is done to simplify union and intersection operations on pairs of SCGs.

The program also contains a number of instructions of the form

$$\begin{aligned} x &\leftarrow new[] \\ x &\leftarrow new[y_1, \ldots, y_n] \\ x &\leftarrow y \\ x &\leftarrow y.s \\ x.s &\leftarrow y \end{aligned}$$

along with "conventional" instructions for manipulating "atomic" values. From these statements basic blocks and use-definition chains can be formed in the conventional way. I will write "$p_i$" to identify "program point $i$".

Now consider construction of $\mathrm{SCG}_i$ for each program point $i$. The SCG "at a point" describes storage containment immediately after executing the instruction at $p_i$. The SCG at $p_i$ depends upon the instruction at $p_i$ and the predecessors of $p_i$. Let

$$p_i(\mathrm{SCG}_j)$$

stand for the SCG produced by applying the transformations associated with the instruction at $p_i$ to $\mathrm{SCG}_j$. Given a program point $p_i$ with predecessors in $preds(p_i)$, $\mathrm{SCG}_i$ is given by

$$\mathrm{SCG}_i = \bigsqcup_{p_j \in preds(p_i)} p_i(\mathrm{SCG}_j).$$

(I will use $\sqcup$ and $\sqcap$ to indicate union and intersection of graphs.) These transformations are similar to the ones described for the construction of a single SCG:

$p_i : x \leftarrow new[\,]$

The new SCG has an additional definition node $d_i$ and additional store node $\sigma_{p_i}$. There is a store edge directed from $d_i$ to $\sigma_{d_i}$.

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

The new SCG has an additional definition node $d_i$ and additional store node $\sigma_{d_i}$. There is a store edge directed from $d_i$ to $\sigma_{d_i}$. For each $s_k, 1 \leq k \leq n$, there are edges labeled $s_k$ directed from $d_i$ to nodes defining $y_k$.

$p_i : x \leftarrow y$

The new SCG has an additional definition node $d_i$. The additional edges are the edges in the sets

$$\{s : \langle d_i, d' \rangle | \exists d_y \text{ defining } y \text{ and } \exists s : \langle d_y, d' \rangle\}$$

and

$$\{\langle d_i, \sigma' \rangle | \exists d_y \text{ defining } y \text{ and } \exists \langle d_y, \sigma' \rangle\}.$$

$p_i : x \leftarrow y.s$

The new SCG has an additional definition node $d_i$ and additional edges from the sets

$$\{t : \langle d_i, d' \rangle | \exists d_y \text{ defining } y \text{ and } \exists s : \langle d_y, d_{y.s} \rangle \text{ and } \exists t : \langle d_{y.s}, d' \rangle\}$$

and

$$\{\langle d_i, \sigma' \rangle | \exists d_y \text{ defining } y \text{ and } \exists s : \langle d_y, d_{y.s} \rangle \text{ and } \exists \langle d_{y.s}, \sigma' \rangle\}.$$

$p_i : x.s \leftarrow y$

The new SCG has an additional definition node $d_i$. It will also have an additional store node $\sigma_{d_i}$ because the assignment is assumed to be non-overwriting. The additional edges are the edges in the sets

$$\{t : \langle d_i, d' \rangle | \exists d_x \text{ defining } x \text{ and } \exists t : \langle d_x, d' \rangle \text{ such that either } s \neq t \text{ or } s \text{ is not a precise selector}\}$$

and

$$\{s : \langle d_i, d' \rangle | d' \text{ defines } y\}.$$

**otherwise**

The result SCG is identical to $\text{SCG}_i$.

Each $\text{SCG}_i$ is the smallest set consistent with (1) the rules above relating storage containment graphs and (2) the initial SCG. The initial SCG is an approximation to the containment relationships holding between objects not defined in the program. The vertices in the initial SCG are disjoint from the vertices added by any $p_i$. The SCGs can be obtained by iteratively performing the transformations until all the SCGs stop growing. If an SCG depends upon itself, then its initial value is the empty graph. Note that each SCG is equal to a finite composition of $p_i$'s and unions based on the initial SCG and the empty set.

### 5.4.1 Containment-preserving unions

In an SCG, "information" is the containment relationships holding between nodes. For each pair of nodes $(i, j)$ it is important to know if $i$ contains $j$, and if so, via what paths. Note that lack of a path is also significant. I wish to show that SCGs for all the points in a program can be collected into a single SCG without changing the containment relationships of a definition at any program point.

Here is a more precise definition of containment:

> Given a graph $G$ and vertices $i$ and $j$ in $G$, *$i$'s containment of $j$ in $G$* is the set of paths from $i$ to $j$ in $G$. This will be written $C(i, j, G)$.

The union of two directed graphs $A$ and $B$ preserves containment if and only if

$$\begin{array}{l} \forall i, j \in V_B, C(i, j, B) = C(i, j, A \sqcup B) \\ \forall i, j \in V_A, C(i, j, A) = C(i, j, A \sqcup B) \\ \forall i \in V_A, \forall j \in V_B - V_A, C(i, j, A \sqcup B) = \emptyset \\ \forall i \in V_B, \forall j \in V_A - V_B, C(i, j, A \sqcup B) = \emptyset \end{array} \tag{5.1}$$

The first two constraints prevent the change of any existing containment in $A$ and in $B$. The next two constraints guarantee that every node in $A$ or in $B$ contains the same nodes that it contained before the union; if the containment was previously undefined, then it is now empty. This may seem slightly asymmetrical; given a node $i$, the set of nodes containing $i$ can increase, but the set which $i$ contains cannot increase. This reflects a bias in the construction of the graph. The operations constructing the graph follow existing paths to some set of nodes and add edges to those nodes from

a (possibly) new node. By guaranteeing that no node contains more nodes after the union than before, we guarantee that the union is also "stable" with respect to the construction of the SCG(s). That is, I wish to guarantee that

$$\forall i, p_i(\text{SCG}) = \text{SCG}.$$

This is clearly true if the contents of nodes are not changed by the union.

The union of $A$ and $B$ will preserve containment if and only if the following conditions are satisfied:

$$\begin{array}{l} (E_A - E_B) \cap (V_A \cap V_B \times V_A \cap V_B) = \emptyset, \\ (E_B - E_A) \cap (V_A \cap V_B \times V_A \cap V_B) = \emptyset, \\ \text{if } \langle i, j \rangle \text{ an edge in } A \sqcup B \text{ and } i \in V_A \cap V_B, \text{ then } j \in V_A \cap V_B. \end{array} \tag{5.2}$$

**Lemma 1** *Conditions (5.1) and (5.2) are equivalent.*

**Proof** First, we prove that (5.2) implies (5.1). Suppose not; suppose that

$$\exists i, j \in V_A \text{ such that } C(i, j, A) \neq C(i, j, A \sqcup B).$$

Clearly, $C(i, j, A \sqcup B)$ is no smaller than $C(i, j, A)$, so there must be a path from $i$ to $j$ in $A \sqcup B$ that is not in $A$.

Let $p = (e_1, \ldots, e_n)$ be a shortest path in $C(i, j, A \sqcup B) - C(i, j, A)$. At least one edge in $p$ must be in $E_B - E_A$ since $p$ is not a path in $A$. Let $e_k = \langle v_k, v_{k+1} \rangle$ be the first such edge.

Clearly, $v_k$ and $v_{k+1}$ are both vertices in $V_B$ since $e_k$ is an edge in $E_B$. Because $A$ and $B$ have the same edges between common vertices and $e_k \notin E_A$, at least one of $v_k$ and $v_{k+1}$ is not in $V_A$. Since $e_k$ is the first edge on the path not in $E_A$, $v_k$ must be in $V_A$ and thus $v_{k+1}$ must not be in $V_A$. This leads to a contradiction, because if $v_k \in V_A \cap V_B$, then $v_{k+1}$ must be in $V_A \cap V_B$.

Suppose that

$$i \in V_A, j \in V_B - V_A, \text{ and } C(i, j, A \sqcup B) \neq \emptyset.$$

Then there is some path $p = (e_1, \ldots, e_n)$ from $i$ to $j$ in $A \sqcup B$. Let $p$ be a shortest such path and let $e_k = \langle v_k, v_{k+1} \rangle$ be the first edge such that $v_{k+1} \in V_B - V_A$. There can be no edges leaving $A \sqcap B$, so $v_k$ must be in $V_A - V_B$. But this implies the existence of an edge from a vertex in $V_A - V_B$ to a vertex in $V_B - V_A$. This is clearly not possible.

Next, we prove that (5.1) implies (5.2). Suppose that

$$\exists \langle i, j \rangle \in E_A - E_B, i, j \in V_A \cap V_B.$$

We have $\langle i, j \rangle \in C(i, j, A \sqcup B)$ and $\langle i, j \rangle \notin C(i, j, B)$. This contradicts the assumptions in (5.1).

Suppose that

$$\exists \langle i, j \rangle \in A \sqcup B, i \in V_A \cap V_B, j \notin V_A \cap V_B.$$

Suppose (without loss of generality) that $j \in V_A$. Then $j \in V_A - V_B$. But $\langle i, j \rangle \in C(i, j, A \sqcup B)$. This is a contradiction, because $i$ is in $B$ and thus $C(i, j, A \sqcup B) = \emptyset$. ∎

I will use the second description to prove additional properties about sets whose union preserves containment. I will write "$A \diamond B$" as an abbreviation for "containment is preserved in the union of $A$ and $B$".

**Lemma 2** *If $A \diamond B, B \diamond C,$ and $A \diamond C$, then $A \sqcup B \diamond C$.*

**Proof** First, show that

$$(E_{A \sqcup B} - E_C) \cap (V_{A \sqcup B} \cap V_C \times V_{A \sqcup B} \cap V_C) = \emptyset.$$

Suppose not; then there is some $e = \langle i, j \rangle$ in the set. Since $e$ is not in $E_C$, $e$ must be in $E_A$ or in $E_B$ and thus $i$ and $j$ are both in $A$ or both in $B$. Suppose that $e \in E_A$. Then we have $\langle i, j \rangle \in E_A - E_C$ and $i, j \in V_A \cap V_C$. But $A \diamond C$, so this is a contradiction.

Next, show that

$$(E_C - E_{A \cup B}) \cap (V_{A \sqcup B} \cap V_C \times V_{A \sqcup B} \cap V_C) = \emptyset.$$

Suppose not; then there is some $e = \langle i, j \rangle$ in the set. It is not possible for $i$ and $j$ to both be in $V_A$ because $i$ and $j$ are already in $V_C$; if they are in $V_A$ and in $V_C$, we have

$$(E_C - E_A) \cap (V_A \cap V_C \times V_A \cap V_C) \neq \emptyset,$$

which cannot happen because $A \diamond C$. Similarly, they cannot both be in $V_B$. Therefore, suppose $i \in V_A \cap V_C$. Because $A \diamond C$, this implies that $j$ is in $V_A \cap V_C$, a contradiction.

Next, show that

if $\langle i, j\rangle$ is an edge in $(A \sqcup B) \sqcup C$ and $i \in V_{A\sqcup B} \cap V_C$, then $j \in V_{A\sqcup B} \cap V_C$.

Suppose that $i \in V_{A\sqcup B} \cap V_C$. If $\langle i, j\rangle \in A$, then $i \in V_A \cap V_C$ and thus $j \in V_A \cap V_C$. If $\langle i, j\rangle \in B$, then $i \in V_B \cap V_C$ and thus $j \in V_B \cap V_C$. If $\langle i, j\rangle \in C$, then either $i \in V_A \cap V_C$ or $i \in V_B \cap V_C$. In either case, $\langle i, j\rangle$ is in both $A \sqcup C$ and $B \sqcup C$ so $j \in V_{A\sqcup B} \cap V_C$. ∎

**Lemma 3** *If $A \diamond B, B \diamond C$, and $C \subseteq B$, then $A \diamond C$.*

**Proof** First, show that

$$(E_A - E_C) \cap (V_A \cap V_C \times V_A \cap V_C) = \emptyset.$$

Suppose there is an edge $\langle i, j\rangle$ in $A$ that is not $C$, with $i$ and $j$ in $V_A \cap V_C$. Since $i$ and $j$ are in $C$ and $C \subseteq B$, $i$ and $j$ are also in $B$. Since $B \diamond C$, and $i$ and $j$ are in $V_B \cap V_C$, and $\langle i, j\rangle \notin E_C$, it must be true that $\langle i, j\rangle \notin E_B$. But $A \diamond B$, and $i$ and $j$ are in $V_A \cap V_B$, so $\langle i, j\rangle$ cannot be in $E_A$, a contradiction.

Next, show that

$$(E_C - E_A) \cap (V_A \cap V_C \times V_A \cap V_C) = \emptyset.$$

This is clearly true, since $(E_C - E_A) \subseteq (E_B - E_A)$ and $(V_C \cap V_A) \subseteq (V_B \cap V_A)$ and

$$(E_B - E_A) \cap (V_A \cap V_B \times V_A \cap V_B) = \emptyset.$$

Next, show that

$$\text{if } \langle i, j\rangle \text{ an edge in } A \sqcup C \text{ and } i \in V_A \cap V_C, \text{ then } j \in V_A \cap V_C.$$

Suppose $\langle i, j\rangle \in A \sqcup C$ and $i \in V_A \cap V_C$. Clearly, $j \in V_A \cap V_B$ because $B$ contains $C$. Since $i$ and $j$ are both in $V_A \cap V_B$, if $\langle i, j\rangle \in E_A$ then $\langle i, j\rangle \in E_B$. If $\langle i, j\rangle \notin E_A$, then $\langle i, j\rangle \in E_C$. However, $B$ contains $C$, so in either case $\langle i, j\rangle \in E_B$. So, we have $\langle i, j\rangle \in E_B \cup E_C$, $i \in V_B \cap V_C$, and $B \diamond C$, so $j \in V_C$. We also have $j \in V_A$, so $j \in V_A \cap V_C$. ∎

**Lemma 4** *If $A \sqcap C = B$, $A \diamond B$ and $C \diamond B$, then $A \diamond C$.*

**Proof** First, note that $V_A \cap V_C = V_B$ and $E_A - E_C = E_A - (E_C \cap E_A) = E_A - E_B$. Therefore,

$$(E_A - E_C) \cap (V_A \cap V_C \times V_A \cap V_C) = (E_A - E_B) \cap (V_A \cap V_B \times V_A \cap V_B) = \emptyset.$$

If $\langle i, j \rangle \in A \sqcup C$ and $i \in V_A \cap V_C$, then either $\langle i, j \rangle \in A$ or $\langle i, j \rangle \in C$. In either case, $i \in V_B$, so we know that $j$ is either in $V_A \cap V_B$ (because $\langle i, j \rangle \in A \sqcup B$ and $i \in V_A \cap V_B$) or in $V_C \cap V_B$. Both of these sets are equal to $V_B$, which is in turn equal to $V_A \cap V_C$, so we conclude that $A \diamond C$. ∎

**Lemma 5** *If $A \diamond B$, then $A \diamond A \sqcap B$.*

**Proof** Since $A \diamond B$, we know that

$$(E_A - E_B) \cap (V_A \cap V_B \times V_A \cap V_B) = \emptyset.$$

Since $E_A - E_B = E_A - (E_A \cap E_B)$ and $V_A \cap V_B = V_A \cap (V_A \cap V_B)$ we know that

$$(E_A - E_{A \sqcap B}) \cap (V_A \cap V_{A \sqcap B} \times V_A \cap V_B) = \emptyset$$

holds. Since $E_{A \sqcap B} - E_A = \emptyset$ the second condition of equation (5.2) holds trivially. If $\langle i, j \rangle$ is an edge in $A \sqcup (A \sqcap B)$, then $\langle i, j \rangle$ is certainly an edge in $A \sqcup B$. Since $A \diamond B$, we know that if $i \in V_A \cap V_B$ then $j \in V_A \cap V_B$. Since $V_A \cap V_B = V_A \cap V_{A \sqcap B}$, the third condition is satisfied and $A \diamond A \sqcap B$. ∎

To summarize, the following properties hold for the $\diamond$ relation:

$$\begin{array}{ll} A \diamond B \wedge B \diamond C \wedge A \diamond C \Longrightarrow A \sqcup B \diamond C & \text{(lemma 2)} \\ A \diamond B \wedge B \diamond C \wedge C \subseteq B \Longrightarrow A \diamond C & \text{(lemma 3)} \\ A \diamond B \wedge B \diamond C \wedge A \sqcap C = B \Longrightarrow A \diamond C & \text{(lemma 4)} \\ A \diamond B \Longrightarrow A \diamond A \sqcap B & \text{(lemma 5)} \end{array}$$

### 5.4.2 Properties of storage containment graphs

The goal is to show that for every pair of SCGs $(\mathrm{SCG}_i, \mathrm{SCG}_j)$ in a program, $\mathrm{SCG}_i \diamond \mathrm{SCG}_j$. Using this and lemma 2, it follows that

$$\forall p_i, \forall v_1, v_2 \in \mathrm{SCG}_i, C(v_1, v_2, \mathrm{SCG}_i) = C\left(v_1, v_2, \bigsqcup_{p_j} \mathrm{SCG}_j\right).$$

To do this, we need a few properties of the $\mathrm{SCG}_i$ and the $p_i$. Note that the general description of the transformations associated with the program points is "follow paths from fixed vertices into a storage containment graph to find some existing nodes, add some nodes, and add some edges from the added nodes to the existing nodes." The properties presented below are true for any transformations of that general form.

**Lemma 6** *There is a definition node $d_j$ in $SCG_i$ if and only if there is a path from a definition $p_j$ to $p_i$.*

**Proof** The only transformation that adds $d_j$ to an SCG is the one associated with program point $p_j$. Because all SCGs are as small as possible and because the initial SCG does not contain $d_j$, there must be a path from $p_j$ to $p_i$ if $d_j \in \mathrm{SCG}_i$. If there is a path from a definition $p_j$ to $p_i$, then the transformation associated with $p_j$ will produce a graph containing $d_j$, and no transformation applied on the path will remove $d_j$. ∎

**Lemma 7** $SCG_j \subseteq p_i(SCG_j)$.

**Proof** The transformation associated with $p_i$ only adds nodes and edges to a graph, so this is true. Note that it is possible that $\mathrm{SCG}_j = p_i(\mathrm{SCG}_j)$. Combining this lemma with the previous one, we see that if there is a path from $p_j$ to $p_i$, then $\mathrm{SCG}_j \subseteq \mathrm{SCG}_i$. ∎

**Lemma 8** *If $d_i \notin SCG$, then $SCG \diamond p_i(SCG)$.*

**Proof** Since $d_i$ is not in $\mathrm{SCG}_j$, $d_i$ is not in $\mathrm{SCG}_j \sqcap p_i(\mathrm{SCG}_j)$. Furthermore, $E_{\mathrm{SCG}_j} - E_{p_i(\mathrm{SCG}_j)}$ is empty, and all edges in $E_{p_i(\mathrm{SCG}_j)} - E_{\mathrm{SCG}_j}$ are of the form $\langle d_i, x\rangle$. Since $d_i$ is not in $\mathrm{SCG}_j \sqcap p_i(\mathrm{SCG}_j)$, the conditions of (5.2) are satisfied and $\mathrm{SCG}_j \diamond p_i(\mathrm{SCG}_j)$. ∎

**Lemma 9** *If $SCG_j \diamond SCG_k$ for all $j, k \in preds(p_i)$, then*

$$\bigsqcup_{p_j \in preds(p_i)} p_i(SCG_j) = p_i\left(\bigsqcup_{p_j \in preds(p_i)} SCG_j\right).$$

**Proof** Both sides of the equation contain the same vertices because $p_i$ always adds the same vertices. Therefore, if there are any differences in the two graphs, the right hand side must contain edges not found in the left hand side. Suppose that there is an edge $\langle d_i, d_j \rangle$ in the rhs that is not in the lhs. This means that there is a path from some node $d_k$ in $\bigsqcup_{p_j \in preds(p_i)} \mathrm{SCG}_j$ that is not in any of the $\mathrm{SCG}_j$. However, for all $l$, $\bigsqcup_{p_j \in preds(p_i)} \mathrm{SCG}_j \diamond \mathrm{SCG}_l$, so this is not possible. ∎

**Theorem 1** *For a value-assignment language, the storage containment graphs specific to program points may be replaced with a single storage graph without losing any information.*

**Proof** First, we must prove that the storage containment graphs for all of the program points have containment preserving unions; that is,

$$\forall \mathrm{SCG}_i, \mathrm{SCG}_j, \mathrm{SCG}_i \diamond \mathrm{SCG}_j.$$

We can prove this for acyclic program flow graphs by induction on execution path length. Let $\mathcal{P}_n$ be the set of program points to which the longest path from the entry node contains no more than $n$ vertices. Let $\mathcal{S}_n$ be the corresponding set of storage containment graphs. Let the notation $\diamond\mathcal{S}_n$ indicate that for all $\mathrm{SCG}_i$ and $\mathrm{SCG}_j$ in $\mathcal{S}_n$, $\mathrm{SCG}_i \diamond \mathrm{SCG}_j$.

For the base case, we must demonstrate the truth of $\diamond\mathcal{S}_0$. The only SCG in $\mathcal{S}_0$ is the initial SCG, and it clearly preserves containment in unions with itself.

For the inductive case, suppose that $\diamond\mathcal{S}_n$ is true. Let $\mathcal{S}_n{}' = \mathcal{S}_{n+1} - \mathcal{S}_n$; that is, $\mathcal{S}_n{}'$ contains the SCGs corresponding to those points to which the longest path has length exactly $n+1$. To establish $\diamond\mathcal{S}_{n+1}$, I must prove both that

$$\forall \mathrm{SCG}_i \in \mathcal{S}_n{}', \forall \mathrm{SCG}_j \in \mathcal{S}_n, \mathrm{SCG}_i \diamond \mathrm{SCG}_j$$

and that

$$\forall \mathrm{SCG}_i \in \mathcal{S}_n{}', \forall \mathrm{SCG}_j \in \mathcal{S}_n{}', \mathrm{SCG}_i \diamond \mathrm{SCG}_j.$$

If $\mathrm{SCG}_i \in \mathcal{S}_n{}'$, then we know that

$$\mathrm{SCG}_i = \bigsqcup_{p_j \in preds(p_i)} p_i(\mathrm{SCG}_j)$$

and that every $p_j$ in $preds(p_i)$ is in $\mathcal{S}_n$. By lemma 9 we know that $\mathrm{SCG}_i$ is equal to

$$p_i\left(\bigsqcup_{p_j \in preds(p_i)} \mathrm{SCG}_j\right).$$

I will abbreviate $\bigsqcup_{p_j \in preds(p_i)} \mathrm{SCG}_j$ as $U_i$; that is, $\mathrm{SCG}_i = p_i(U_i)$. Because we know that $U_i$ is the result of a finite union of members of $\mathcal{S}_n$ and that $\diamond\mathcal{S}_n$ holds, we may conclude that $U_i \diamond \mathrm{SCG}$ for every SCG in $\mathcal{S}_n$.

Let $G_n = \bigsqcup_{\mathrm{SCG} \in \mathcal{S}_n} \mathrm{SCG}$; because this is a finite union we know that $G_n \diamond \mathrm{SCG}$ for every SCG in $\mathcal{S}_n$ and that $G_n \diamond U_i$.

Because the program is acyclic, we know that vertex $d_i$ is not in $U_i$, and thus by lemma 8 we know that $\mathrm{SCG}_i \diamond U_i$.

To use lemma 4 to show that $\mathrm{SCG}_i \diamond G_n$, we must show that $\mathrm{SCG}_i \sqcap G_n = U_i$. Because the program is acyclic we know that vertex $d_i$ is not in $G_n$, and we know by the properties of $p_i$ that $\mathrm{SCG}_i$ is equal to $U_i$ plus the vertex $d_i$ and some edges of the form $\langle d_i, x\rangle$. Since $d_i$ is not in $G_n$, we conclude that $\mathrm{SCG}_i \sqcap G_n = U_i$. By the lemma, we also conclude that $\mathrm{SCG}_i \diamond G_n$.

Now, every $\mathrm{SCG}_j \in \mathcal{S}_n$ satisfies $\mathrm{SCG}_j \subseteq G_n$ and $\mathrm{SCG}_j \diamond G_n$. Since $\mathrm{SCG}_i \diamond G_n$, we can use lemma 3 to show that $\mathrm{SCG}_i \diamond \mathrm{SCG}_j$.

To see that $\mathrm{SCG}_i \diamond \mathrm{SCG}_j$ for every $\mathrm{SCG}_i$ and $\mathrm{SCG}_j$ in $\mathcal{S}_n{}'$, notice that $\mathrm{SCG}_i \sqcap \mathrm{SCG}_j = U_i \sqcap U_j$. This is so because the graph is acyclic and because the vertices $d_i$ and $d_j$ added to $U_i$ and $U_j$ are distinct. $U_i$ and $U_j$ are both finite unions of members of $\mathcal{S}_n$, so we know that $U_i \diamond U_j$. By lemma 5 we know that $U_i \diamond U_i \sqcap U_j$. Because

$$\mathrm{SCG}_i \sqcap \mathrm{SCG}_j = U_i \sqcap U_j \subseteq U_i$$

and $\mathrm{SCG}_i \diamond U_i$ and $U_i \diamond \mathrm{SCG}_i \sqcap \mathrm{SCG}_j$ we can use lemma 3 to show that $\mathrm{SCG}_i \diamond \mathrm{SCG}_i \sqcap \mathrm{SCG}_j$ and similarly that $\mathrm{SCG}_j \diamond \mathrm{SCG}_i \sqcap \mathrm{SCG}_j$. By lemma 4 we see that $\mathrm{SCG}_i \diamond \mathrm{SCG}_j$; thus $\diamond\mathcal{S}_{n+1}$ holds true.

To prove that the storage containment graphs for a program whose flow graph contains cycles have containment preserving unions we partition the graph up into its strongly connected components. If there is a path from $p_j$ to $p_i$, then $\mathrm{SCG}_j \subseteq \mathrm{SCG}_i$. All points in a strongly connected component $C$ thus must have the same storage containment graph $\mathrm{SCG}_C$, and $\mathrm{SCG}_C$ must contain the SCGs associated with points on paths into the SCG. There are additional vertices and edges in $\mathrm{SCG}_C$; the extra vertices are associated with definitions within $C$, and the extra edges originate at the extra vertices.

Thus, we may treat the collection on transformations within $C$ as one large transformation $p_C$ that adds several vertices and edges from those several vertices. This transformation shares with the individial $p_j$ properties used above in the proof; therefore the proof holds for this generalized transformation. ∎

For an example demonstrating how this works, consider the piece of code

$a \leftarrow \textbf{new}()$
**if . . .**
    **then** $b.s \leftarrow a$
    **else** $c.s \leftarrow a$
. . .

In the **then** clause a definition node (call it $d_{then}$) is created that contains $a$. Likewise the **else** clause defines $d_{else}$ containing $a$. Merging the storage containment graphs from those two points produces a graph yielding the information that $a$ is contained in both $d_{then}$ and $d_{else}$ in both clauses; that is, it appears that merging the two graphs produces less accurate information. The two definitions $d_{then}$ and $d_{else}$ are only live in their respective clauses however, so this does not matter. In other words, the information that $a$ is contained in $b$ is only interesting if $b$ is live.

## 5.5 Accounting for side-effects

Propagating side-effects to definition nodes is more complicated. Given an assignment $d : x.s \leftarrow y$, any def nodes that might be aliased to $x$ might be changed by this update. The nodes aliased to $x$ are those with edges to any store node to which $x$ has an edge.

### 5.5.1 First approach

One approach is to perform the following additional steps when building an SCG for a language with side-effects. For a given update $d : x.s \leftarrow y$:

1. Form the set of store nodes adjacent to $d$.

2. Find all defs adjacent to those store nodes.

3. Attach to those defs edges labeled $s$ to definitions for $y$. It is *not* safe to remove edges labeled $s$ from those nodes, since our information is not precise, and this assignment might not actually change those edges. This is so even if there is exactly one store node adjacent to $x$ and its alias, because that store node could represent two different allocations from one site.

The SCG resulting from this method is not as precise as the set of SCGs maintained for program points. This is because the SCG transformations for a language with storage side-effects do not generate SCGs with containment preserving unions.

### 5.5.2 Second approach

Notice that the major difference between updates with and without side-effects is the edges added to the graph to reflect possible side-effects. For a definition $d_i$, these side-effect edges need not have the form $\langle d_i, x\rangle$; all of the other edges introduced by an update without side-effects are still added. That is, lemmas 6 and 7 still hold, and thus the storage containment graphs computed at points in a strongly connected component will all be the same. Furthermore, at every point $p_i$ the storage containment graph reflecting side-effects will contain within it the storage containment graph for that point that does not reflect side-effects[5].

At worst we need only construct one SCG per strongly connected component of the program flow graph. This provides a convenient order for constructing these SCGs; if the strongly connected components have been identified, then we can process them in topological order. This also allows us to share graphs, saving space and time. Given a point $p_i$, it is only necessary to consider the SCGs for $p_i$'s predecessors to construct $p_i$'s SCG. The SCG so constructed will contain within it the predecessors' SCGs, so they can be shared to save space.

Though the point SCGs in a side-effect model always contain the corresponding point SCGs in a side-effect-free model, information is lost if the (single) side-effect-free SCG is shared by the SCGs in the side-effect model. Consider the conditions under which an assignment $d_1 : x.s \leftarrow y$ can affect

[5]Except for the store nodes representing storage allocated at updates; these, however, may be left out of a side-effect-free storage containment graph without affecting the rest of it.

(add $s$-edges to definitions for $y$ to) a definition node $d_2$. First, $d_2$ is only affected if $d_1$ and $d_2$ both have edges to the same store node $\sigma$. If $d_2$ precedes $d_1$ in the flow graph, then the edge from $d_2$ to $\sigma$ is in $d_1$'s SCG, so we have not lost any information. If $d_1$ precedes $d_2$ and $d_2$ is $x.s \leftarrow z$, then $d_2$ should not include any $s$-edges to definitions for $y$; however, using the side-effect-free SCG as a base for side-effect SCGs will cause those edges to appear as a side-effect. If the side-effect-free SCG is not used in this way, then $d_2$ and $d_1$ are not aliased at $d_1$ and the side-effect will not occur.

I believe it is difficult to do better than this. We are creating SCGs as precise as point-specific SCGs for a language with storage side-effects. One way to improve this is to model the side-effect of a killed reference; that is, *remove* edges in a point-to-point SCG transformation. This is not safe in the current model, because an SCG only expresses *may*-contain relationships. To remove an edge, it is necessary to have *must*-contain information. This is especially hard in the SCG model because all instances of storage from the same allocation are treated as one; no distinction is made between an allocation that is repeated many times, though only one instance is live, and an allocation that is repeated many times to build a large linked structure.

## 5.6 Comparison with SETL and Lisp analyses

The SCG yields better information than the value-flow analysis used in SETL [FSS83, Cou85]. An example demonstrating this is shown in figure 5.3. There, a new piece of storage is assigned to $a$ (line 1). The value contained in $a$ is subsequently incorporated into $b$ as either the first or second component (lines 2 and 3). At this point, the SETL value-flow analysis and the SCG provide the same information; both tell that $a$ may be contained in $b$ as either the first (definition at line 1) or second (definition at line 2) component. Assigning $b$ into $c$ (line 4) forces the SETL analysis to combine these two relationships into a single one; thus the SETL analysis can only infer that $a$ is a "CMP" (any component) of $c$. The SCG, on the other hand, copies edges from the nodes for the definitions at lines 2 and 3, preserving the information that $a$ is either the first or second component of $c$. The SETL analysis is also unable to infer that $a$ cannot be the first member of $c$ after line 5, because combining "not CMP1" with "CMP" still yields "CMP". The

SCG, however, can infer this fact and can also determine that after line 6 $a$ is not contained within $c$. This fact is not discovered by the SETL value-flow analysis.

| Line | Code |
|---|---|
| 1 | $a \leftarrow new[\,]$ |
| | **if** ... |
| 2 | **then** $b \leftarrow new[a, x, y]$ |
| 3 | **else** $b \leftarrow new[x, a, y]$ |
| 4 | $c \leftarrow b$ |
| 5 | $c.1 \leftarrow z$ |
| 6 | $c.2 \leftarrow w$ |

Figure 5.3: Comparison of SETL value-flow analysis and SCG

The analysis in Lisp compilers to discover what closures may be stack-allocated is a special case of this analysis. A closure may be stack-allocated if all uses of the closure are applications occurring within functions which may be stack-allocated. If a closure is assigned to a variable that is assigned to a global, referenced from the heap, returned as a function result, or passed to another procedure, then the closure cannot be stack-allocated.

A careful definition of "containment" for closures will induce a version of the SCG analysis subsuming the Lisp analysis. For this purpose, a "use" of a closure is any reference to it, and one closure "contains" another if the text of the first contains an application of the second. In the SCG analysis, the base case is at least as good as the one in the existing Lisp analysis; if a closure is returned from a procedure or passed as an argument to a procedure, then it cannot be stack-allocated. The SCG analysis may discover that an assignment to a heap-allocated cell or global variable does not preclude stack allocation, so here it may yield improved results. Given the same base case, the new analysis will yield exactly the same results as the Lisp analysis.

The S-1 Common Lisp compiler yields better results for allocation of floating point numbers than the SCG because (1) floating point parameters to a procedure need not be heap-allocated, even with no knowledge of the procedure's behavior and (2) the Lisp compiler attempts to discover situations in which no storage at all is needed.

Given that parameters need not be heap-allocated, the SCG analysis can stack-allocate numbers whenever the Lisp analysis can, and perhaps in a

few situations where the Lisp compiler does not. This is because the Lisp compiler heap-allocates any number that is contained within another object; with containment analysis this conservative approach is not necessary. Use of the SCG alone does not allow a compiler to conclude that no storage at all is needed; doing this requires either analysis of pointer expressions or special treatment to deal with the Lisp run-time data structures.

## 5.7 Complexity of constructing SCG for a value language

The size of an SCG and the time required for its construction depend upon several things. There will be one def node for each definition in the program, and one store node for each allocation site in the program. There may also be one $\perp_t$ node for each "type" $t$ in the program and a corresponding store node $\sigma_{\perp_t}$. In the worst case, all of the def nodes can be linked to each others' storage, producing $O(|defs|^2)$ edges. If all of the def nodes are linked to each other by every possible selector, then there will be $|defs|^2 \times |selectors|$ edges. Note that such a graph gives "bad news" answers; everything is contained by everything else, and thus cannot be overwritten or stack allocated.

Some amount of pre-processing based upon the use-definition chains will simplify the analysis of certain pathological graphs. One source of worst-case behavior in an SCG is cyclic patterns of simple assignment. The program in figure 5.4 gives rise to such a pathological graph. Here, the first 26 definitions

$a \leftarrow new[\,]$
...
$z \leftarrow new[\,]$
`while ...` {
  $a \leftarrow b$
  $b \leftarrow c$
  ...
  $z \leftarrow a$
  }

Figure 5.4: Program yielding pathological SCG

(allocations) create 26 def nodes and 26 store nodes in the SCG. The next 26

definitions are assignment statements, adding 26 more def nodes to the SCG. Each of the second 26 def nodes has copied to it edges leaving definitions for the right hand side of its corresponding assignment. Ultimately, each of the 26 assignment nodes in the loop will have an edge to each of the 26 store nodes.

The assignment graph $(V_A, E_A)$ is formed by using definitions in the program as the nodes $V_A$ and placing an edge from a node $d_i$ to another node $d_j$ if $d_i$ is a definition for the right hand side of the assignment statement corresponding to $d_j$. In the program above, the assignment nodes clearly form a strongly connected component. Since all of the edges leaving these nodes will be identical, and because the questions we are asking depend only upon the store nodes reached or not reached, it makes sense to collapse this cycle into a single node. This eliminates the pathological behavior associated with constructions similar to the one shown in figure 5.4.

Another transformation to the assignment graph that will not change reachability properties of the SCG is the $T_1$ transformation of Hecht and Ullman [HU72]. Clearly, if there is only one definition for the right hand side of an assignment statement, then the edges leaving that definition will be identical to the edges leaving the definition node for the right hand side. Note that the assignment graph for the example in figure 5.4 is not reducible, so the $T_1$ and $T_2$ transformations alone are not sufficient to collapse the assignment cycle into a single node.

### 5.7.1 Update Graph

The assignment graph can be generalized to include suitably labeled edges from def nodes to all other definitions which they use.[6] I will call this generalized graph the "update graph" (UG) because it can also be used to propagate updates through the SCG as it is built. The update graph is also important because transformations on the update graph can reduce the size of the final SCG. In a language without storage side effects an update graph completely specifies a storage containment graph. If side-effects are possible, then the strongly connected components of the flow graph must also be known in order to construct an SCG.

The update graph contains the same nodes as the SCG, but additional

[6]The direction of the edges is reversed, but this does not change the situation with regard to strongly connected regions in the assignment graph.

edges and edge labels are added to describe the copying of edges required to construct the SCG. An edge from one definition node $x$ to another definition node $y$ indicates that $x$ uses the value defined at $y$ in some way. Where an edge label is parameterized by some $s$, $s$ is a selector. The UG edge annotations (for edges directed from $x$ to $y$) are:

*copy*$(s, o)$ This edge is generated by an updating assignment statement or by a simple assignment statement. In the updating case, $x : a.s \leftarrow b$, $x$ is the definition node for this statement, $y$ is some definition node for $a$, and the edge is labeled *copy*$(s)$. In the simple case, $x : a \leftarrow b$, $x$ is the definition node for this statement and $y$ is some definition node for $b$. The edge is labeled *copy*$(\emptyset)$, meaning "no selector". The additional parameter $o$ indicates whether this is a *overwriting* assignment or not; $o$ has the values *reference* (overwriting) or *value* to describe what sort of copying takes place. This parameter is often implicit; in many value language implementations simple assignment is by reference, but update is not.

*select*$(s)$ This edge is generated by a selection from a variable that $y$ can define; that is, $a \leftarrow b.s$. Here, $x$ is the definition node corresponding to this statement, and $y$ is a definition node for some statement defining $b$.

*contain*$(s)$ This edge is either generated by an allocation/initialization or by an updating assignment statement; for example, $a.s \leftarrow b$ or $a \leftarrow new[\ldots, b, \ldots]$.

*store* This edge is generated by any definition that allocates storage whether it is a copying update or an allocation.

Note that the update graph has number of nodes equal to the number of definitions in the program plus the number of storage allocations (for store nodes). The number of edges in the update graph is equal to the number of edges in the use-definition chains plus the number of storage allocations (for store edges). Thus, the update graph has size comparable to the use-definition chains computed for a program.

The update graph completely specifies a storage containment graph. They share the same nodes, and by applying a series of transformations to the update graph the edges of the storage containment graph are obtained. Given an edge $\langle x, y \rangle$ from a def node $x$ to a def node $y$, apply the following transformations depending upon the edge label.

*copy*$(s, o)$ For each edge *contain*$(t) : \langle y, z\rangle$ such that $t \neq s$, create an edge $t : \langle x, z\rangle$; that is, copy edges from $y$ to $x$. If $o$ is *reference*, then also copy store edges from $y$, but otherwise do not. (If $o$ is *value* then a store node already exists.) If $s = \emptyset$ or if $s$ is not a precise selector, then copy all *contain* edges from $y$ to $x$.

*select*$(s)$ For each edge *contain*$(t) : \langle y, z\rangle$ such that $t = s$, create a new edge *copy*$(\emptyset,$ *reference*$) : \langle x, z\rangle$. In a later iteration this edge will cause edges to be copied from $z$ to $x$.

*contain*$(s)$ These edges do not cause any copying when the SCG is built, but they are copied and are used to find nodes from which edges are copied. After all transformations to the UG are complete, these edges form the "selector edges" of the SCG.

To adapt the update graph for use with a side-effect language, some *contain* edges must be further tagged to indicate that the containment represents an in-place modification. When such an edge, say *contain*$'(s)$, is copied, the tag is not copied.

*store* These edges do not cause copying when the SCG is built, but are themselves copied. After all transformations are complete, these edges form the "store" edges of the SCG.

### 5.7.2 Shrinking the update graph

Transformations on the UG can reduce the number of nodes in the UG and resulting SCG without affecting the information in the SCG. These transformations are very similar to changes to the assignment graph described above. When the only edge leaving a node $x$ is a *copy*$(\emptyset)$ edge to some node $y$, then $x$ may be merged with $y$ and all edges $\langle w, x\rangle$ changed to $\langle w, y\rangle$. This is the reversed form of the $T_1$ transformation applied to the assignment graph[7].

If there is a subgraph of the UG strongly connected by *copy*$(\emptyset,$ *reference*$)$ edges (hereafter referred to as *copy* edges), then all of the nodes in that subgraph may be merged into a single node and edges to any of them redirected to the merged node. This is the same as removal of strongly connected regions from the assignment graph. In the update graph, however, it is possible to introduce additional *copy* edges to create strongly connected regions that do

[7]The sense of information flow is reversed, so the $T_1$ transformation is also reversed.

not exist in the assignment graph. For example, consider the transformations implied by the graph shown in figure 5.5. An edge labeled $copy(\emptyset, reference)$

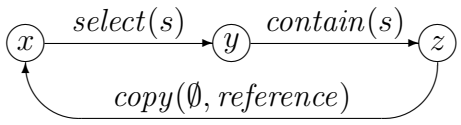



Figure 5.5: Introduction of copy edges

from $x$ to $z$ will be added. This forms a cycle of *copy* edges containing $x$ and $z$. It is important to discover strongly connected regions within the update graph because every node in the region will have the same edges in the SCG; there will be no additional information, but many more nodes and edges. Since some strongly connected regions will appear as the update graph is processed and it will reduce the size of the final answer, it may be advantageous to incrementally discover and collapse strongly connected components as edges are added.

There is an additional way to create $copy(\emptyset, reference)$ edges. Suppose that there are two edges from $x$ to $y$ labeled $copy(s, o)$ and $copy(t, p)$. The effect of the first edge is to copy all edges except for $contain(s)$ edges; the effect of the second edge is to copy all edges except for $contain(t)$ edges. Together the two nodes will copy all edges from $y$ to $x$, and therefore should be replaced with a single edge labeled $copy(\emptyset, q)$ (where $q$ is *reference* if either $o$ or $p$ is *reference*). Thus, there will never be more than one *copy* edge from one node to another.

### 5.7.3 Processing the update graph

The descriptions above provide enough information for a chaotic reduction of the update graph; if a reduction can be applied, then apply it. Other methods may be more efficient, however, and it may be profitable to watch for the creation of strongly connected components. Chaotic iteration does not do this.

When a $copy(s)$ edge from $x$ to $y$ is processed, the edges leaving $x$ may change. If this is so, then all nodes with *copy* or *select* edges directed to $x$ must be checked to see if they change. Notice that any algorithm that reduces the update graph must perform a transitive closure along $copy(\emptyset, reference)$

edges. Given this observation, I conclude that there is little point in separately searching for strongly connected components because the reduction itself must perform a transitive closure; any search for cycles can better be made part of the edge-propagation process.

Though the UG completely specifies an SCG in the value model, additional information from the control flow graph will aid in the efficient construction of an SCG. If the containment information has been computed for all of the (control-flow) predecessors of a node, then that node will only need to be visited once. This holds in both the value and side-effect models; side-effects only affect containment properties *at* successor nodes, though the containment properties *of* predecessor nodes may be changed *at* the successors. Thus, if we partition the control flow graph into its strongly connected components and topologically sort the resulting DAG, we can ensure that each component is "visited" only once. This holds for both the side-effect and value models, though in the side-effect model we create a new SCG for each component. We have, however, reduced the problem for both models to that of computing a storage containment graph on a strongly connected component; the only difference now is that the algorithm for a side-effect model must deal with edges modeling side-effects.

In the SCG for a value language each definition node $d$ in a strongly connected component $C$ (in the control flow graph) can be the source of an edge (containment or storage) to any of its predecessors. That is, if there are $m$ definition nodes in the component, $n$ definition nodes in the rest of the graph, and a maximum of $S$ selectors applied to any definition node, then there can be $O(n \times m \times S)$ edges leaving nodes in $C$. There can also be $O(n \times m \times S)$ *select* edges leaving nodes in $C$, but these are fixed for a given program; i.e., no new selection edges will appear.

We will use a worklist algorithm to model the addition of edges leaving nodes in $C$. When a *copy* or *contain* edge $\langle d, e \rangle$ is created it is placed on the worklist; to remove it from the worklist adjacent nodes are examined to see if the addition of this edge creates new edges. When all the adjacent nodes have been examined and the new edges created (and placed on the worklist), this edge is removed from the worklist.

When a *copy* edge $copy(s) : \langle d, e \rangle$ is added, new edges are found by examining *contain* edges leaving $e$. There can be $O(n \times S)$ of these edges, and therefore as many as $O(n \times S)$ edges could be added to the worklist before

$\langle d, e\rangle$ is removed. Since there can be only $O(n \times m)$ *copy* edges leaving $C$[8], this gives us a worst-case complexity of $O(n^2 \times m \times S)$.

When a *contain* edge $contain(s) : \langle d, e\rangle$ is added, new edges are found by examining edges in $C$ directed to $d$. There can be $O(m)$ edges labeled $select(s)$, and these can cause the creation of $O(m)$ copy edges. There can be $O(m)$ copy edges directed to $d$ and these can create edges by copying this edge. Since there can be $O(m \times n \times S)$ contain edges leaving $C$, we obtain a complexity of $O(m^2 \times n \times S)$.

Together, this gives us a worst-case complexity of $O(n^2 \times m \times S)$ to process a strongly connected component $C$ of the control flow graph. If a strongly connected component of copy edges is created in the SCG, all of the edges must be between definition nodes in $C$. If the component is not detected on the fly (and I have made no attempt to do this above) it can still be detected with an ordinary transitive closure algorithm restricted to the *copy* edges between nodes in $C$. This will have cost $O(m^3)$. Therefore, processing all components of the graph for a language with value-assignment will have worst-case time complexity $O(S \times n^3)$.

To handle side-effects, note that an in-place *contain′* edge becomes a *contain* edge when it is copied. Side effects occur when a store edge $\langle d, \sigma\rangle$ is created and an in-place edge $contain'(s) : \langle d, e\rangle$ exists. The result is to add an edge $contain(s) : \langle d', e\rangle$ for every $d'$ adjacent to $\sigma$. In the worst case, there can be $O(n)$ in-place contain edges leaving $d$, and $O(n)$ store edges adjacent to $\sigma$. Therefore, the time complexity to process one new store edge leaving an in-place update is (worst-case) $O(n^2)$. Furthermore, since there can be as many as $O(n^2)$ of these store edges, the total worst-case complexity is $O(n^4)$.

If a store edge $\langle d, \sigma\rangle$ is added and $d$ is not an in-place update, it is still necessary to consider side-effects to $\sigma$ from some other in-place update. Here, there can be $n$ possible updates affecting $\sigma$ and each of these can add as many as $n$ contain edges. Again, we obtain $O(n^4)$ worst-case complexity.

## 5.8 Dealing with procedure calls

The effects of calling procedures can also be modeled in the SCG. In the simplest treatment, the results returned by a procedure call contain everything that is both reachable within the procedure (via parameters or global

[8]Note the "trick" above for limiting the number of copy edges between two nodes to one.

variables) and consistent with the type system. In a language with storage side-effects a procedure call generates every possible consistent side-effect. In language with private variables, the results of one procedure call can contain objects reachable at another procedure call. This behavior can be mimicked by treating private variables as global variables that are not accessed by other procedures or modules.

Such an approximation is safe and requires no interprocedural analysis, but it is very pessimistic. Less pessimistic estimates will require some interprocedural information. The most precise estimate possible using the storage containment and update graphs is obtained by substituting a procedure's UG into the caller's UG at a procedure call site when the SCG for the calling procedure is built. This method suffers from two drawbacks:

- Recursive procedures cause infinite substitution.
- Reduction of the procedure's substituted UG yields containment information irrelevant to the caller—for example, containment relationships holding in the called procedure's temporary storage.

Both of these problems are solved by using a *partially reduced update graph*. A PRUG contains information about storage accessible when the procedure returns. This is typically storage reachable from returned values, global variables, and reference parameters. The graph is "partially reduced" because edges leaving the definition nodes corresponding to the procedure's parameters are not known, and thus cannot be copied or traversed. Thus, any instance of $copy(s) : \langle x, y \rangle$ or $select(s) : \langle x, y \rangle$, where $y$ is the definition node corresponding to a formal parameter, cannot be reduced. Instead, node $x$ is marked "unreduced" and all other nodes copying or selecting from an unreduced node will be marked "unreduced". Any unreduced node might contain storage from the caller that was not directly passed as a parameter. Note that if the results of a procedure call contain (applying the relation transitively) only reduced nodes then none of the caller's storage can be incorporated into those results.

This doesn't yet explain how to deal with recursive procedures or use a PRUG. A PRUG is substituted into a caller's UG at the procedure call site. By "substitution" I mean that a copy of the called procedure's PRUG is inserted into the caller's graph, and $copy(\emptyset)$ edges are added from the formal parameter definition nodes to definition nodes for the actual parameters in the calling procedure and $copy(\emptyset)$ are added from uses of the results in the

caller to the result definitions in the called procedure. Because a PRUG contains only nodes reachable at procedure exit, no nodes will be irrelevant. It is correct to substitute a new copy of the PRUG for each call site because storage allocated in different invocations of a procedure will be distinct, while the definitions for global variables and parameters will be obtained from the caller's context.

For recursive procedures the PRUG is generated by successive approximation. $\text{PRUG}_0$ is empty—nothing contains anything. Given $\text{PRUG}_i$, $\text{PRUG}_{i+1}$ is generated by substituting the edges of $\text{PRUG}_i$ into the UG at recursive call sites. The nodes of $\text{PRUG}_{i+1}$ are used instead of generating new copies. When $\text{PRUG}_{i+1} = \text{PRUG}_i$, the PRUG for the recursive procedure has been found and is equal to $\text{PRUG}_i$. This process must certainly terminate because the size of the graph is bounded.

A disadvantage of substituting a new PRUG at each call site is that this can lead to exponential growth of the update and storage containment graphs. Suppose a procedure $P_0$ contains two calls to $P_1$, and for $1 \leq i < n$, $P_i$ contains two calls to $P_{i+1}$. This will produce $2^i$ copies of the PRUG for procedure $P_i$. One heuristic for avoiding this exponential growth is to perform a single substitution for mutually unreachable (within one calling procedure) calls to a procedure. This technique fails, however, to reduce the expansion for the case where $P_i$ contains calls to $P_{i+1}$ and $P_{i+2}$. It is not yet clear how important it is to separate call sites, and it is not clear how large PRUG's will grow in practice. To achieve the exponential growth described above, it is necessary for each $P_i$ to actually make some structural contribution to the graph; simple transmission of formal parameter (of $P_i$) uses to actual parameter (of $P_{i+1}$) definitions should not increase the size of the PRUG for $P_i$ (this is so because of the reversed $T_1$ transformation mentioned in section 5.7.2).

## 5.9 Shortcomings

The storage containment graph identifies and separates storage resulting from different program points, but does not make any distinction between different instances of storage resulting from a single point. One application where this information would be very useful is in the parallelization of loops; if the storage allocated and referenced in different iterations of the loop were guaranteed to be disjoint, then it would be advantageous to run the loop

in parallel on multiple processors. Here, knowing that all instances of the storage are disjoint helps in inferring that the loop can actually be run in parallel (that there are no *loop carried dependences* [All83]) and ensures that the memory allocations can be performed locally, thus reducing use of global memory.

The worst-case complexity for construction of an SCG is also unappealing. One hopes that sparseness or some other property of "typical programs" will make the analysis affordable. On the other hand, the bounds described are not tight; it is possible that a closer examination of the algorithm (or a better algorithm) could reduce the time bound.

## 5.10 Related work

Though the SCG was derived from the value-flow analysis used in the SETL compiler, it bears a strong resemblance to analysis proposed by Jones and Muchnick [JM82]. There they use abstract interpretation [CC77, Myc81] to construct approximations to values appearing at definition sites. These approximations can be regarded as regular tree languages describing the values that can appear at the definition sites. The grammar for the regular tree language is similar to the SCG but it describes only value approximations, not use of storage. That is, the SCGs produced by value and side-effect semantics are always different, but the interpretations produced by Muchnick and Jones can be the same. By making the relationships between values and storage explicit, my analysis deals naturally with aliasing, side-effects, and the effects of overwriting optimizations on storage lifetime. I also feel that the kinds of questions answered by this analysis are more naturally expressed in a graphical framework; "can $x$ contain $y$'s storage?" translates to "is there a path from nodes defining $x$ to storage used by $y$?". Looking at regular tree grammars and asking the same questions, I find myself treating the grammar itself as the description of a graph.

Ruggieri has recently completed a dissertation [Rug87] in which a very similar analysis is proposed. She establishes a lattice theoretic framework and shows that the analysis is monotonic, but not distributive. She also describes an interprocedural version of the analysis. The work presented here differs in several ways: Ruggieri's analysis associates containment information with variables, while this work associates containment information with definition sites. It turns out that this distinction is important; in the presence of

recursive data types, Ruggieri is forced to assume a heuristic choice of "order" derived from the type-system. Past a certain point increasing the order does not improve the information computed. This happens when detailed information is combined with coarse information. For example, in

$$\begin{array}{ll} & a \leftarrow b \\ \textbf{l:} & a \leftarrow \textbf{cons}(x, a) \\ & \cdots \\ & \textbf{go to } l \end{array}$$

it is possible to establish that $a$ could contain $b$ nested 10 **cdr**'s deep; however, $a$ can also contain either $a$ or $b$ nested only one **cdr** deep. Increasing the order to 10 does not yield any useful information about $a$. By associating information with definition sites, however, this method automatically gets the most precise information that can be obtained by tuning the order.

This work also treats the important special case of languages with value-assignment semantics. For this case it is possible to compute a single instance of the containment graph to accurately describe the entire program rather than one per point or strongly connected component.

Ruggieri has devoted more attention to interprocedural containment and lifetime analysis; my approach, though correct, is potentially very expensive.

# Chapter 6

# Improved allocation optimizations

The heap-to-stack allocation conversion described here is better than the one proposed for the SETL compiler; it is safer and more general. The analysis in Lisp compilers is less general because it only attempts to optimize allocations for closures and numbers; it may also be unsafe, but this is hard to verify. Barth's analysis seems safe, but it is not designed to use the stack and depends on a reference counting model; when storage is nested, its allocation is not eligible for optimization in his system.

## 6.1 Problems with interval-based stack allocation

Schwartz describes a general method for converting heap allocations to stack allocations. Each interval in the program flow graph is examined to discover values whose lifetimes are contained within the interval. These are allocated on the stack. Because the interval has one entry node, it is easy to record the stack height at interval entry. At interval exit the stack is restored to its height on entry, thus freeing all objects allocated on the stack within the interval.

There are three problems with this approach. It is difficult to generalize to later interval reductions of the graph, it may interfere with tail-call elimination, and in certain situations the stack can grow without bound.

Interval-based stack allocation does not easily generalize to later reductions because stack deallocations are introduced after the first reduction. If a value $x$'s lifetime is contained within an interval $I_3$ of the second reduction (a $G_1$ interval) but not within an interval of the first reduction (a $G_0$ interval), that means that it is allocated inside one $G_0$ interval $I_1$ but used within another $G_0$ interval $I_2$. If the stack allocation for $x$ is performed *within* $I_1$, then it will be undone at exit from $I_1$; that is, before its use in $I_2$. Because $x$'s lifetime is not contained in any $G_0$ interval, if $x$ is to be stack allocated it must be stack allocated "underneath" those values whose lifetimes are contained within $G_0$ intervals. This problem is illustrated in figure 6.1. There

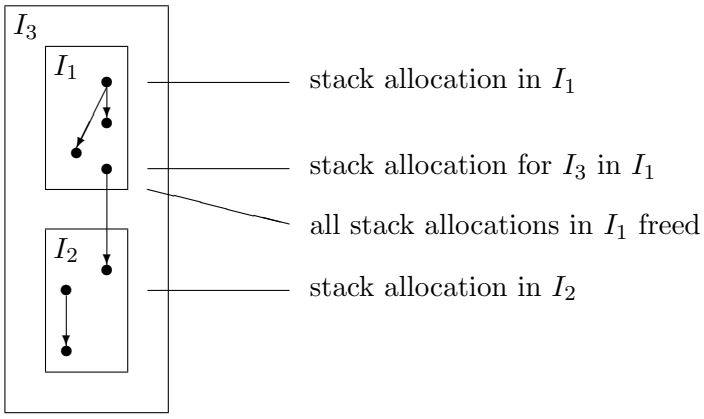



Figure 6.1: Problems with interval-based stack allocation

are two ways to deal with this problem. The $G_0$ interval can be "split" (as shown in figure 6.2) and objects allocated within the first part are freed at the split. At the true exit from the interval values on the stack are not freed. The other approach is to preallocate storage before entering $I_1$ (as shown in figure 6.3). This is possible whenever the size of the object can be determined before entering $I_1$.

Interval-based stack allocation also interferes with tail recursion elimination, though the interference is somewhat trivial. Procedure calls occurring at interval exits must be treated specially; if the interval can be split (as described above), then this should be done so that all storage on the stack may be freed.

The worst problem with interval-based stack allocation is that simple, likely programs cause it to waste an unbounded amount of stack storage. This

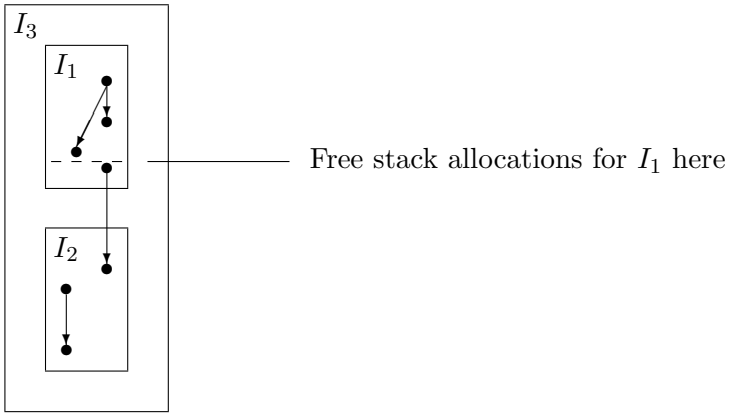



Figure 6.2: Splitting Interval

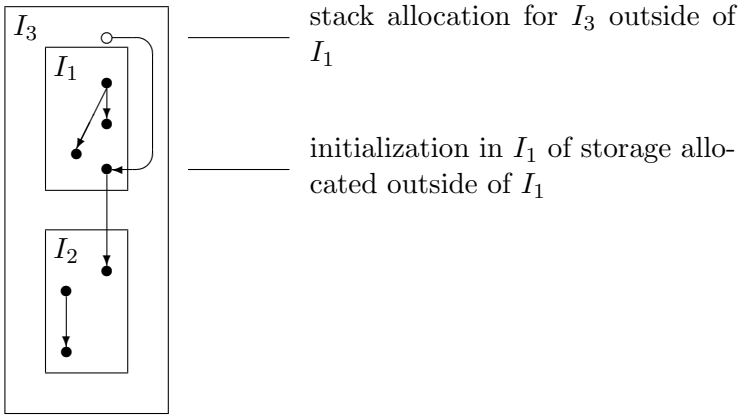



Figure 6.3: Hoisting allocation

is so because an interval may contain a loop (executed an unbounded number of times) containing the allocation of a value whose lifetime is contained within the body (one iteration) of the loop. Since this is all contained within the interval containing the loop, that allocation may be performed on the stack. Since the loop is entirely contained within the interval, nothing is removed from the stack until after leaving the loop. Each iteration of the loop executes another allocation, so each iteration of the loop consumes some amount of the stack. The value, however, is not live from one iteration to the next, so at any time all but one of the stack allocations is useless. Note that if that storage had been heap-allocated it would be recoverable by a garbage collection, but since it is allocated on the stack it cannot be recovered until exit from the loop. This is a serious problem; optimization should not convert a program with bounded resource consumption into one with unbounded resource consumption.

This problem can be treated by being more careful of allocations within loops, either by not moving these allocations to the stack or by popping allocations off of the stack after each iteration. I claim, however, that interval-based stack allocation is the wrong approach. It uses a clean run-time model and a clean compile-time analysis, but the model and the analysis are somewhat less elegant after all of the patches have been applied. The interval-based model also pays too much attention to keeping the stack height low at places where stack height is not a problem.

For example, the interval method works poorly for the following code fragment:

$a \leftarrow \mathbf{new}(\ldots)$
`loop` {
  $\ldots$
  $a \leftarrow \mathbf{new}(\ldots)$
  $\ldots$
  }
$\ldots a \ldots$

Here, the loop forms an interval. The lifetime of the definition within the loop, however, extends outside of the loop and thus outside of the interval, so it cannot be freed within the loop. It is also unwise to perform those allocations on the stack without freeing them within the loop, because there is no bound on the amount (percentage) of stack that can be wasted. Clearly, the interval method is not useful here.

## 6.2 An improved method

An algorithm to convert heap allocations to stack allocations should certainly obey one constraint: it should not convert a program that runs within available memory resources to one that does not (ignoring for the moment marginal cases). Rather than focusing on *doing* the optimization in intervals and patching it up to keep it safe, I feel that one should focus on *not doing* the optimization in situations where it will cause unbounded stack waste.

My notion of waste has been vague so far, because measuring waste in terms of storage used makes it hard to guarantee that waste will not be caused by single objects of large size. There is also the problem that in marginal cases a garbage-collected system will continue to run, though very slowly; in such situations, even a small amount of wasted storage will cause failure. I will assume that objects' sizes are much smaller than the total amount of memory in use, and I do not guarantee that these optimizations will not cause marginal programs to fail. I also want to assume that the total amount of space available to the stack is much larger than the amount used in any procedure activation.

Copying collectors and mark-and-sweep collectors typically require that one-half to one-third of the available memory be unused; thus, to achieve comparable resource consumption the storage-allocation optimization must ensure that no more than (say) one-third of the storage on the stack is unused. Together with the assumptions above, this means that a compiler should ensure that:

1. There is a bound on the total number of unused stack objects with unknown (to the compiler) size. This bound is necessary because a pathological program might "waste" (for example) one object of size 9 for each useful object of size 1. If the number of these objects is not bounded, then placing many of them on the stack could waste 90% of the available memory.

2. The bounds on wasted stack space are not exceeded by objects with size known to the compiler. The compiler may not be able to place a bound on the number of unused stack objects, but it can bound the fraction of stack space wasted by these objects.

I assume that all stack allocations within a procedure are undone upon exit from the procedure. This is reasonable because this is a traditional way

to implement procedure linkages, and because a compiler usually processes at least a whole procedure (as opposed to a part of a procedure). Given this, to introduce stack allocations the compiler must consider three cases:

1. allocations occurring on paths to recursive[1] calls—The compiler cannot easily place a bound on the number of times one of these allocations might be executed during a procedure activation.

2. allocations occurring within loops—The compiler cannot easily place a bound on the number of times one of these allocations might be executed during a procedure activation.

3. allocations occurring anywhere else—Each of these allocations is performed no more than once per procedure activation.

To perform an allocation in the third class on the stack, the compiler must guarantee that no storage resulting from that allocation is in use upon exit from the procedure. Using the SCG and liveness information, the compiler must ensure that no definition node on an SCG path to the allocation node is live at the exit. A definition node is live at the exit if it can define the procedure's result, a global variable, or a reference formal parameter. Given that such an allocation is not in use at procedure exit, it is always safe to stack-allocate it because the statement performing the allocation can be executed at most once before the memory is freed and the procedure itself is not recursive. Thus, at most one instance of storage from the allocation can be in use at any point during the execution of the program.

Converting allocations of the second class to use the stack requires more care to ensure that the stack waste stays small. I will first treat the case of allocations within "simple loops"—that is, single entry strongly connected regions containing no embedded cycles that do not include the header (entry) node.

### 6.2.1 Short allocations

An allocation $d$ within a loop $L$ is called *short* if its lifetime, restricted to nodes and edges in $L$, contains no cycles. Less formally, this means that there is a set of edges within $L$ across which the allocation is known to be

[1] As determined by the call graph; without a call graph, one must assume that all calls can be recursive.

dead, while permitting the value to "escape" through a loop exit and even re-enter through the loop header node (via an edge outside of $L$). To place these allocations on the stack requires the following transformations:

1. Record the stack height at loop entry. Call this the *loop stack base*.

2. Choose a set of edges which cuts the loop, and across which none of the short allocations is live. This set of nodes may be found by depth-first search from the loop header node forward along live edges in $L$. Insert code on each edge to restore the stack height to the loop stack base.

Several important details are omitted in the above description. The complete set of short allocations may have lifetimes whose union contains a cycle, making it impossible to find a loop-cutting set of edges. Short allocations with loop-invariant size should be allocated only once, in the loop header. This may not be any more efficient than allocating them in the portion of the stack that shrinks and grows with each loop iteration because it will still be necessary to adjust the stack on each iteration (unless all short allocations can be performed in the header), but it will make it easier to find a loop-cutting set of edges (if such a set exists). The definitions for "nodes" and "edges" in the graph are also important; it may be necessary to split a node into two parts (if this is possible) to cut the loop.

Detecting opportunities to introduce stack allocation within loops requires information computed "relative" to the loop. For example, I want to ignore live ranges that escape the loop and re-enter through the top of the loop. Such ranges are not short in the enclosing loop, but they are short in the inner loop and should be stack allocated there. For the best information from the SCG, this requires a SCG computed relative to the loop. The problem is simplified somewhat because we are only interested in lifetimes of values allocated within the loop; it is not necessary to construct a graph of the type system or to use any of the information derived from the surrounding graph. In fact, the type system used can be completely trivial—only a $\perp$ node with *no* leaving edges because anything reachable through the $\perp$ node came from outside the loop. For languages with side-effects, updating an object allocated outside the loop to contain an object allocated within the loop generates a containment edge from the $\perp$ node to the second object, giving it a non-short lifetime.

Several features of the heap-to-stack optimization for short allocations should be noted:

1. Variables receiving stack-allocated values within the loop may also be allocated outside of the loop. To see this, consider what happens on the first iteration. At the loop entry, the loop stack base is set to the current stack height and the body of the loop is entered. Somewhere in the execution of the loop a cutting edge is traversed and the stack height is reset to loop stack base. Any variable initialized outside the loop points to storage below loop stack base, so it is affected neither by the loop entry nor by the stack reset.

2. The live range of stack-allocated storage within the loop can span two iterations. This is so because the stack height is reset at a cutting edge, but not (necessarily) at a cycle edge or an exit edge.

3. The live range of stack-allocated storage within the loop can escape the loop. This is so because the stack height is not reset upon loop exit. Though the body of the loop can be executed many times, the loop is entered only once, and thus it is exited only once. Since the stack is reset during each iteration of the loop, there can be no more additional objects on the stack than there are allocation sites within the loop body.

4. The live range of stack-allocated objects within the loop can even extend beyond the lifetime of the procedure activation in certain cases. When this is the case, the object must be copied from the stack to heap-allocated storage upon exit from the loop, and pointers to the object's storage must be updated to reflect this. This is desirable because it is expected that it will take less time do $n$ (where $n$ is the number of loop iterations) stack allocations, one heap allocation and one copy than it will to take to do $n$ heap allocations. Within the loop, the object's address will not change during a garbage collection, so it is possible to perform optimizations that use the object's address within the loop.

   Note that objects with loop-invariant size, short lifetimes, but lifetimes extending beyond the procedure activation should be heap-allocated in the loop header unless it appears that optimizations involving the object's address will be exceptionally profitable.

If the union of the lifetimes of the short allocations contains a cycle, then there will be no set of cutting edges. In this case, not all of the allocations

can be moved onto the stack. Ideally, the compiler will choose the "best" allocations to place on the stack and heap-allocate the others. Determining the minimum cost set of DAGS to remove to break all cycles, however, is an NP-complete problem, even with unit cost. This follows by transformation of vertex cover [GJ79] to cycle-breaking DAG removal (CBDR).

**Proof** First, show that an instance of vertex cover can be transformed to CBDR in polynomial time.

Given an undirected graph $G = (V, E)$, create a rooted directed graph $G'$ with root vertex $R$ and vertices $a_i$ and $b_i$ for each vertex $v_i$ in $G$. For each $v_i$ in $G$ there are edges $\langle R, a_i \rangle$ and $\langle b_i, R \rangle$. For each $v_i$ in $G$, create a DAG $D_i$ in $G'$ equal to

$$(\{b_i, R, a_i\}, \{\langle b_i, R \rangle, \langle R, a_i \rangle\})$$

At this point there are no cycles in $G'$, so there can be no cycles in the union of the DAGs. The graph $G'$ is shown in figure 6.4. If $n$ is the number of vertices in $G$, then the number of vertices and edges in $G'$ are $2n+1$ and $3n$.

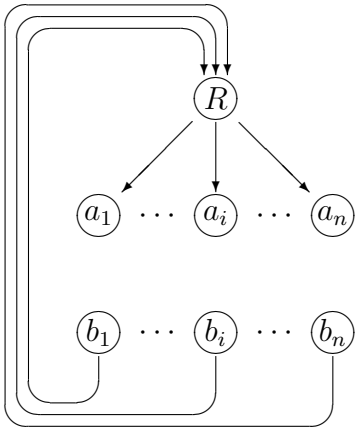



Figure 6.4: Initial $G'$

Now consider the edges in $G$. For each edge $\langle v_i, v_j \rangle$, add four vertices

$$c_i^j, d_i^j, c_j^i, d_j^i$$

and eight edges

$$\langle a_i, c_i^j \rangle, \langle c_i^j, d_i^j \rangle, \langle a_j, c_j^i \rangle, \langle c_j^i, d_j^i \rangle, \langle a_i, c_j^i \rangle, \langle d_i^j, b_j \rangle, \langle a_j, c_i^j \rangle, \langle d_j^i, b_i \rangle$$

These are shown in figure 6.5. The vertices

$$c_i^j, c_j^i, d_i^j, d_j^i$$

and edges

$$\langle a_i, c_i^j \rangle, \langle c_i^j, d_i^j \rangle, \langle a_i, c_j^i \rangle, \langle d_j^i, b_i \rangle$$

are added to $D_i$ (shown in figure 6.6) and similar vertices and edges are added to $D_j$. Note that $D_i$ and $D_j$ remain acyclic. If the number of edges in $G$ is $m$, then $G'$ now has $1 + 2n + 4m$ vertices and $3n + 8m$ edges. Clearly, $G'$ can be constructed in time polynomial in the size (number of vertices) in $G$.

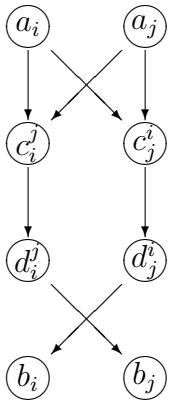



Figure 6.5: Effects of $\langle v_i, v_j \rangle$ on $G'$

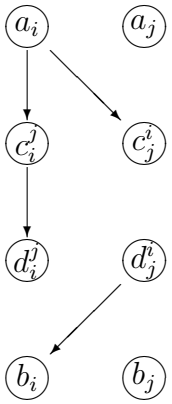



Figure 6.6: Effects of $\langle v_i, v_j \rangle$ on $D_i$

Neither $D_i$ nor $D_j$ contains a cycle, but their union does. Each edge $\langle v_i, v_j \rangle$ in $G$ adds cycles to the union of the DAGs, and these are the only cycles added to the union. Repeating this process for all edges in $G$ produces a new $G'$ and new DAGs $D_1$ through $D_n$. Clearly, if there is an edge in $G$ then there is a cycle in $G'$, and if there is a cycle in $G'$ there is an edge in $G$.

Removing a DAG $D_j$ from the union of DAGs corresponds to removing a vertex and its incident edges from $G$. The vertices $a_j$ and $b_j$ disappear, because they are only contained in $D_j$, and for all $i$ such that there is an edge $\langle v_i, v_j \rangle$ in $G$, the edges $\langle a_j, c_j^i \rangle$,$\langle c_j^i, d_j^i \rangle$,$\langle a_j, c_i^j \rangle$, and $\langle d_j^i, b_i \rangle$ disappear; the vertices $c_i^j$, $c_j^i$, $d_i^j$ and $d_j^i$ remain because they are contained in $D_i$, but they are not part of any cycle and thus may be ignored. Thus, for all $i$ the effects on $G'$ of an edge $\langle v_i, v_j \rangle$ are undone.

Thus, removing DAGs to break all cycles in $G'$ corresponds to removing vertices to totally disconnect $G$. A set of cycle-breaking DAGs in $G'$ corresponds (one-for-one) to a vertex cover of $G$. Furthermore, given $G$, $G'$ can be constructed in time polynomial in the number of vertices in $G$. Thus, CBDR is NP-hard.

A solution to CBDR can be verified in polynomial time, so CBDR is also NP and is thus NP-complete. ∎

It might appear that CBDR is a harder problem than the one needed for the optimization because storage lifetimes are always rooted graphs, while the DAGs constructed in the proof are not, but it is possible to re-root each DAG $D_i$ with the addition of a vertex $d_i$ and edges from $d_i$ to all vertices $d_i^j$. Thus, the problem remains NP-complete even for rooted DAGs.

In fact, I would like to allow heap-to-stack transformations on an even larger class of allocations. Constraining the lifetime of allocated storage to be contained within a DAG guarantees that there will be at most one piece of storage allocated from a given site at any point within the loop. The DAG description is over-restrictive; if the lifetime contains no cycles passing through the allocation site, then at most one piece of storage allocated at the site is active within the loop. The corresponding constraint on the union of the lifetimes is that no allocation site lies on a cycle in the union. Of course, this is a generalization of CBDR, so it is NP-hard, and a solution can be verified in polynomial time, so it is NP-complete.

### 6.2.2 Long allocations

A *long* allocation within a loop is one creating storage that will be used in later iterations of the loop. Moving these allocations onto the stack does not waste space because their storage is guaranteed to be reachable during the execution of the loop.

In a loop, each long allocation adjusts both the stack pointer and the loop stack base, so that the storage remains allocated across edges where the stack pointer is reset to loop stack base. Both long and short allocations can be moved to the stack in the same loop, but their combination can cause some storage to be wasted. Ideally, a long allocation will occur at a point where the stack pointer is equal to the loop stack base; this means that all the storage allocated between the loop stack base and the initial (at loop entry) stack pointer will be devoted to instances of the long allocation and

thus will all be in use.

If this is not the case, short allocations live at the long allocation site will be "pinned" underneath the long allocation. Since short allocations are guaranteed to be live in two or fewer iterations of the loop, this will waste storage. Furthermore, the sizes of short allocations within the loop are unknown, because (as noted above) any short allocation with known size can be performed outside of the loop. The size of the long allocations may or may not be known. This puts the optimizer into a difficult situation; moving all the long allocations to the stack might lead to excessive memory consumption.

One solution to this is to check at run-time the size of the long allocation against the size of the storage that it will pin (equal to the difference of loop stack base and stack height). If it is greater than or equal to this amount, then the waste is limited to 50%. If it is not this large, then it is allocated from the heap instead and stack height base is left alone. The benefits of this approach may be marginal, since the object's address may still be in collectible storage and thus be movable; that is, the possibility of the object's collection can interfere with optimization of expressions including that address. However, it will cut down on the amount of collectible memory used if long-allocated objects are large.

A serious drawback to this optimization is that it is hard to show that an allocation is long. The storage containment graph does not help, because it encodes *may* information, not *must* information. It is also difficult to copy "a" long allocation from the stack to the heap, because many instances of storage resulting from the long allocation will be active (one for each execution of the allocation, by definition).

### 6.2.3 Nested Loops

When loops are nested, it is possible to analyze them from the inside out and treat the inner loops as single unbreakable nodes in enclosing loops. The node must be unbreakable because the stack height cannot fall below (inner) loop stack base after entering the (inner) loop. Thus, any storage stack-allocated at entry to the loop must remain stack allocated until after loop exit. In analysis of outer loops, any allocations escaping inner-loops are treated as allocations at the single unbreakable node. Using this treatment, it is possible to apply the techniques described above to nested loops.

It may happen that an allocation nested within loops $L_1$ (outermost)

through $L_n$ (innermost) is short in loops $L_i$ through $L_n$, but not in $L_{i-1}$. If it is also not long in $L_{i-1}$, then it must be copied to the heap upon exit from $L_i$. This is not always possible; consider this piece of code:

```
L1while ...{
    a ← new_array
    b ← new
    for i = 1 to n
      if f(i)
        then a[i] ← b
        else a[i] ← x
  } ...
  return a
```

Both $a$ and $b$ are short within $L_1$ because they are assigned new values at the top of the loop; clearly, the stack can be reset before executing the first statement of the loop body. It is also clear that the storage pointed to by $a$ can be copied to the heap after leaving $L_1$ because the only pointer to it is $a$. It is not easy, however, to copy the storage to which $b$ points because there are also pointers to it stored in various elements of the array referenced by $a$.

If, on the other hand, an allocation is short in loops $L_i$ through $L_n$ and long in $L_{i-1}$ (assuming that the compiler is able to discover this), the compiler must ensure that not too much space is wasted by other allocations that are still on the stack at exit from $L_i$. This is only possible at compile-time if the objects have constant size.

Nested long allocations are much more difficult to cope with. The difficulty of copying the results of a long allocation to the heap has already been noted. It is also difficult to compute the size of a long allocation, since its size is actually the sum of the sizes of several objects that may be linked by pointers in an arbitrary (perhaps even cyclic) way. This makes it difficult to guarantee what fraction of the stack will be wasted. Thus, if a nested allocation is long in more than one loop than it very likely will not be stack-allocated.

### 6.2.4 Procedure calls

#### Parameter and local allocations

An allocation whose lifetime spans a procedure call may be stack allocated provided that

1. the call is not recursive;
2. the allocation's lifetime does not escape the calling procedure's activation. It may be necessary to use the called procedure's partially reduced update graph to discover that this is so; without this information, worst-case assumptions about global variables imply that the allocation's lifetime is not bounded.

or that

1. the call *is* recursive;
2. the allocation's lifetime does not escape the calling procedure's activation;
3. the allocation's lifetime spans the call to and return from the procedure; this corresponds to a long allocation in a loop.

If neither of these sets of constraints is satisfied and a standard linkage is used, then there is no bound on the number of objects resulting from the allocation that might become unreachable. Unless these objects are known to be small compared to the activation record and the coexisting long (call-spanning) allocations, they should not be stack-allocated because that could lead to excessive waste of stack space.

In some situations it is possible to tailor a recursive procedure call in such a way that allocations not spanning a recursive call can be stack-allocated without wasting space. Suppose there is an allocation $a$ in a procedure $P$, and $a$'s lifetime extends into a recursive call of $P$ but not across it. That is, $a$ is dead in one activation of $P$, but live into *part of* the next. With an ordinary linkage, first the allocation $a$ is performed, then the activation record for the call to $P$ is (stack) allocated. Sometime later $a$ becomes inaccessible while the recursive call's activation record is still in use. In a tailored linkage, the two allocations are reversed. First the activation record is (stack) allocated, then the storage for $a$ is (stack) allocated. Sometime later

$a$ becomes inaccessible. If it is also true that no stack allocations performed since $a$ was allocated are live, then the stack can be restored to reclaim the now-dead space. Note that this requires that $a$'s lifetime not extend past $a$'s allocation in the called procedure, and that it not extend past the construction of the activation record for the second recursive call to $P$. This is clearly a special-case optimization, since it can only be used when all these conditions are met, and it requires generation of a separate, tailored copy of a recursive procedure.

### Result allocations

So far any allocation that might be incorporated into a procedure's results has been performed on the heap because the allocation might be active when the activation record is removed from the stack. No amount of analysis in the called procedure itself can change this, because the lifetime of the result depends upon the calling procedure. If the calling routine preallocates storage for the result and passes it in, then it becomes possible to analyze the lifetime of the allocation within the calling procedure and possibly perform it on the stack. If the result allocation's lifetime is not contained within the calling procedure's activation, then it is part of the calling procedure's result and can be allocated by the calling procedure's caller. It appears that this process removes the need for garbage collected storage, but (as noted by Ruggieri [Rug87]) it is difficult to reserve storage for linked and variably sized objects because their sizes may not be known at compile time.

A second complication arises when results are preallocated for recursive procedures. When storage is reserved, it is *not yet live*; it is wasted until it is used. It can happen that none of the storage preallocated for a series of recursive calls to a procedure is live, and it can also be the case that the number of live instances of the result is bounded by a constant. Thus, wasted stack space can occur.

A third complication is the possibility that no storage will be needed; the called routine might return an existing object. Again, this is a more serious problem for recursive procedures because the number of activations is not bounded, and thus the potential waste (one object per activation) is not bounded.

The situation is much better for non-recursive procedures. Preallocation is still impractical if the result size is unknown, but the number of activations is bounded. Preallocation can result in waste, but this waste is no greater

than the waste produced by local values during a procedure activation. Note that in any case result stack preallocation must satisfy the same constraints as stack allocation of local storage. If a result is stack-preallocated within a loop, then the storage lifetime must be short or long, or else it will waste storage in the same way that a local allocation can waste storage.

# Chapter 7

# Conclusions

I think that this dissertation is very important, because it is the first work that I have seen that proposes anything near a practical approach to analysis of nested objects. Previous work on optimization and garbage collection missed the problems caused by their combination, and previous algorithms to convert heap allocations into stack allocations did not address the problem of excess storage consumption.

## 7.1 Contributions

The third chapter gave examples showing how common optimizations applied to addressing expressions can interfere with garbage collection. Invariant, inductive and redundant expression optimizations create pointers that the garbage collector cannot reliably follow or update. Dead code elimination can remove code that establishes collector invariants. Register allocation makes it difficult to distinguish pointers from non-pointers. The efficiency lost to non-optimization of code is a hidden cost of garbage collection.

Garbage collection and optimizations can coexist. This is done through the use of a protocol defining what the run-time environment should look like just before a garbage collection. The compiler generates code that converts the optimized environment to a collectible environment (the *cleanup map*) and code that reflects change in the collectible environment back into the run-time environment (the *dirtyup* map); calls to the garbage collector are bracketed by calls to these maps. Because garbage collections occur very infrequently and have non-trivial cost, it is expected that the time spent

mapping run-time environments will be more than made up by improved optimizations. Note that if there is no optimization, then these maps are trivial and add little to the cost of a garbage collection.

Optimizations that convert heap allocations into stack allocations have multiple benefits because of interference. The allocation itself is cheaper; less memory obtained from the heap means longer intervals between garbage collections, and thus lower direct garbage collection costs; the address of a stack-allocated object will not change during a garbage collection, and thus (unlike the address of a heap-allocated object) may be treated as a constant across heap allocations; the address of a heap-allocated object *is* constant across a stack allocation, so this optimization tends to extend the ranges over which the addresses of heap-allocated objects are constant. This is ample motivation for heap-to-stack allocation optimizations.

Chapter five introduces the storage containment graph. This graph has a number of useful properties. It correctly models sharing, aliasing and overwriting. By associating information with definition sites instead of with variables it obtains information at least as good as any existing containment analysis. The derivation of this graph falls into a monotone data-flow analysis framework; in a separate proof it is shown that the derivation of an SCG has the Church-Rosser property (and thus a unique solution is obtained independent of any algorithm). However, the derivation is not distributive. In addition, the storage containment graph is invariant over strongly connected components of the control flow graph, thus requiring fewer instances of the graph to precisely describe the containment that may hold in a program. For a language with value-assignment semantics it is proved that only *one* instance of the graph is required for precise information.

Chapter six describes a better approach to heap-to-stack allocation optimizations. The most important constraint in any code optimization should be: Do not break a running program. Existing techniques are either ad hoc or do not satisfy this constraint. The new approach allows a heap allocation to be converted to a stack allocation whenever it can be shown that this will not lead to unbounded waste of stack storage. For allocations within loops this leads to a characterization of allocation as *short* or *long*; a short allocation's lifetime is non-cyclic within a loop, so it may be reclaimed at each iteration; a long allocation has the property that all instances of storage allocated remain live through all iterations of the loop, and thus the storage is not wasted. Associated with optimization of short allocations is the *cycle-breaking-DAG-removal* problem; this is shown to be NP-complete. In-

terprocedural allocation optimizations are discussed in light of "do not break a running program," and some pitfalls are pointed out.

## 7.2 Future Research

This thesis describes an analysis and proposes optimizations; it would be interesting to see how well it works in the real world. Likely languages to compile might be variants of Pascal and Modula without **free**; it is otherwise possible to compile and optimize these languages, so the addition of a garbage collector and the optimizations described here is an incremental task. Application to Lisp, Scheme, ML or Russell might be profitable, but ordinary analysis and optimization of these languages is not as easy because of polymorphism, higher-order functions, and continuations. In addition, the classical run-time representation of Lisp and Scheme data structures does not provide many opportunities for address optimizations. Backus's FP might present good opportunities for using this work.

Numerous loose ends remain; I describe how exact information about object initialization is needed to perform certain forms of dead code elimination in a garbage-collected system, but I do not describe a way to get this information. Without such analysis newly allocated objects must always be initialized.

I describe conversion of heap allocations with long lifetimes into stack allocations, but I provide no help in determining when an allocation is actually long; in fact, the storage containment graph is inappropriate for this because it expresses only *may*-contain information.

I describe an algorithm for generating a storage containment graph, but the time bound that I derive is unappealing. I do not think that the algorithm is really that bad in practice, but I do not have proof of this.

The algorithm presented here for interprocedural containment analysis has exponential worst-case behavior, and minimum-cost cycle-breaking DAG removal is NP-complete. Investigations of approximations for these two problems might be fruitful.

# Bibliography


[AC72] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In Randall Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[ACK81] F. E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 79–101. Prentice-Hall, 1981.

[All70] Fran E. Allen. Control flow analysis. *SIGPLAN Notices*, 5:1–19, 1970. Cited in Hecht [Hec77].

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformations*. PhD thesis, Rice University, 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–461, August 1978.

[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.

[BB77] A. P. Batson and R. E. Brundage. Segment sizes and lifetimes in ALGOL 60 programs. *Communications of the ACM*, 20(1):36–44, January 1977.

[BGS82] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele Jr. An optimizing compiler for lexically scoped LISP. In *SIGPLAN Symposium on Compiler Construction*, pages 261–275, 1982.

[Bob75] D. G. Bobrow. A note on hash linking. *Communications of the ACM*, 18(7):413–415, July 1975.

[Bob80] Daniel G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[Bro75] Frederick P. Brooks, Jr. *The Mythical Man Month.* Addison-Wesley, 1975.

[Bro85] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *Functional Programming Languages and Computer Architecture*, pages 273–288, 1985.

[BS83] Stoney Ballard and Stephen Shirron. The design and implementation of VAX/Smalltalk-80. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 127–150. Addison-Wesley, 1983.

[BW88] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software, Practice and Experience*, pages 807–820, September 1988.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

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

[CG77] D. W. Clark and C. C. Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2):78–87, February 1977.

[CH84] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *SIGPLAN Symposium on Compiler Construction*, pages 222–232, 1984.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Cla79] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–59, January 1979.

[CLZ86] Ron Cytron, Andy Lowry, and Ken Zadeck. Code motion of control structures in high-level languages. In *Conf. Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, 1986.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[Coc70] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5:20–24, 1970. Cited in Hecht [Hec77].

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

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[Cou85] Courant Inst. of Mathematical Sciences, The SETL Project. The SETL optimizer. Source listing, May 1985.

[CS70] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers*. Courant Institute of Mathematical Sciences, New York, 1970.

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

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[FSS83] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems*, 5(1):26–45, January 1983.

[FW79] D. P. Friedman and D.S. Wise. Reference counting can manage the circular invironments[sic] of mutual recursion. *Information Processing Letters*, 8(1):41–44, 1979.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[GW76] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976. Cited in Kennedy [Ken81].

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

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal on Computing*, 1:188–202, 1972. Cited in Tarjan [Tar74] and Kennedy [Ken81].

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[Hug85] John Hughes. A distributed garbage collection algorithm. In *Functional Programming Languages and Computer Architecture*, pages 256–272, 1985.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conf. Record of the Ninth Annual Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Ken72] Kenneth W. Kennedy. Safety of code motion. *Intern. J. Computer Math.*, 3:117–130, 1972.

[Ken81] Kenneth W. Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, 1981.

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN Symposium on Compiler Construction*, pages 219–233, 1986.

[KS75] Kenneth W. Kennedy and Jacob T. Schwartz. An introduction to the set theoretical language SETL. *Comp. & Maths with Appls*, 1:97–119, 1975.

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

[Mar71] S. Marshall. An ALGOL 68 garbage collector. In J. E. L. Peck, editor, *ALGOL 68 Implementation*, pages 239–243. Elsevier North-Holland, 1971.

[MJ76] Steven S. Muchnick and Neil D. Jones. Binding time optimizations in programming languages: Some thoughts toward the design of an ideal language. In *Conf. Record of ACM*

*SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 77–91, 1976.

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

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, 1981.

[New42] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, April 1942. Cited in [Hue80].

[Nie77] Norman R. Nielsen. Dynamic memory allocation in computer simulation. *Communications of the ACM*, 20(11):864–873, November 1977.

[Owi81] Susan Owicki. Making the world safe for garbage collection. In *Conf. Record of the Eight Annual Symposium on Principles of Programming Languages*, pages 77–86, 1981.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PB85] Paul Walton Purdom, Jr. and Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Reinhart and Winston, 1985.

[RLW85] Paul Rovner, Roy Levin, and John Wick. On extending Modula-2 for building large, integrated systems. Technical Report 3, DEC Systems Research Center, 1985.

[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, 1985.

[Rug87] Christina Ruggieri. *Dynamic Memory Allocation Techniques Based on the Lifetimes of Objects*. PhD thesis, Purdue University, August 1987.

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

[Sch76] Jacob T. Schwartz. A coarser, but simpler and considerably more efficient copy optimization technique. SETL Newsletter 176, Courant Inst. of Mathematical Sciences, New York Univ., August 1976.

[SCN84] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 159–166, 1984.

[SS76] Guy Lewis Steele Jr. and Gerald Jay Sussman. LAMBDA: The ultimate imperative. AI Memo 353, Massachusetts Institute of Technology, 1976.

[Ste77a] Guy L. Steele Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful, or LAMBDA: The ultimate GOTO. In *ACM National Conference*, pages 153–162, 1977.

[Ste77b] Guy L. Steele Jr. Fast arithmetic in MacLISP. In *Proceedings of the 1977 MACSYMA Users' Conference*, pages 215–224, 1977.

[Ste78] Guy L. Steele Jr. RABBIT: A compiler for SCHEME. Technical report, Massachusetts Institute of Technology, May 1978.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tar74] Robert E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[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*, pages 157–167, 1984.

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, second edition, 1983.

[Wod71] P. L. Wodon. Methods of garbage collection for ALGOL 68. In J. E. L. Peck, editor, *ALGOL 68 Implementation*, pages 245–262. Elsevier North-Holland, 1971.

[Zel83] Polle T. Zellweger. An interactive high-level debugger for control-flow optimized programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 159–171, 1983.