# Analysis of Pointers and Structures

David R. Chase*
Mark Wegman†
F. Kenneth Zadeck‡

## 1 Introduction

High-level languages could be optimized significantly if compilers could determine automatically how pointers and heap allocated structures are used. Better knowledge of aliasing can improve classical optimizations applied to scalars (common sub-expression elimination, loop-invariant code motion, reduction in strength, constant propagation) by permitting less conservative assumptions about what is affected by an update to storage, and can aid in dependence analysis for purposes of parallelization. In addition, information about the shape and use of linked data structures can be used to apply storage overwriting and allocation optimizations (for instance, reusing storage instead of making a copy).

This problem is a complex one, in part because it is possible to construct unbounded data structures that must necessarily be represented in some finite way. As with almost all program analysis and optimization problems, one must limit the kinds of information one tries to collect, because exact information is generally undecidable or at least very difficult to compute. Our work follows that of Jones and Muchnick [JM81] who summarize the data structures allocated in a heap by making a graph, in which one node corresponds to possibly many nodes in the heap. The major issue is how to choose which heap cells to associate with which nodes. We view the program as a generator for data structures. Each symbolic execution of `X ← cons (A, X)` adds a new node to the data structure. As the data structure grows, it must be compressed by making one node stand for many. We differ from other works principally in the kind of information we use to do the compression.

Most work to date [Sch75a, Sch75b, JM81, Rug87, RM88, LH88, Lar89, HPR89] does this compression by bounding acyclic path length in the modeled data structures; this is known as *k-bounded approximation*. They limit the length of (acyclic) paths to $k$ by truncating long paths with summary nodes containing all paths occurring in the original. The (path-length) $k$-bounded approaches have several flaws: they are potentially very slow (unless $k$ is very small), unbounded structures lose all structure beyond depth $k$, and information provided by the program structure is ignored.

We propose a different way of summarizing linked data structures that has better worst case time bounds, allows a particularly efficient implementation for the sparse (expected) case, preserves information about unbounded data structures, and takes advantage of information provided by the structure of the program. Rather than try to match one part of the data structure with another (in essence solving some variant of the subgraph isomorphism problem), we look at the program for hints about which parts of the data structure are related, and then match those parts. In addition, a simple extension of this analysis discovers data structures that are "true lists" and "true trees"; that is, it discovers data structures within which there is no aliasing at all. Earlier program analyzers and optimizers make good use of this information [Lar89], but must rely on assertions made by the programmer to obtain it. The analysis presented here obtains this useful information automatically.

In Section 2, we describe the abstract model for programs we can analyze and describe the information our algorithm determines. In Section 3, we present a simple though inefficient algorithm that demonstrates some of the aspects of our technique. In Section 4, we extend the technique in Section 3 to discover more information, such as whether a data structure is a list or tree. In Section 5, we construct an efficient algorithm that obtains the same results as our simple algorithm. In Section 6, we extend our algorithm interprocedurally, and in Section 7, we compare our results to those in the literature.

## 2 Background

We describe below the simplified language for programs that our algorithm accepts and the model of program storage that our algorithm constructs.

### 2.1 The Model of the Program

The algorithms presented here handle programs containing Lisp-like structures. All our examples are written in pidgin Algol with Lisp-like data structures. The standard operators such as `car`, `cdr` and `cons` are supported; `X.car` means `(car X)`.[1] Furthermore, the side-effecting operators `rplaca` and `rplacd` are supported; these are written as assignments to the `car` and `cdr` fields on the left side of the assignment operator (e.g. `X.car ← ...`).

In our framework, pointers are simply references to nodes with a fixed number of fields, some of which are pointers. Our examples use cons cells. We consider only three operation on pointers: allocate new, update specific field, and select specific field.[2] We do not allow the unrestricted pointer arithmetic that is possible in languages such as C,

*1160 Laurel Street #2, Menlo Park, CA 94025.

†IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

‡Computer Science Dept., P.O. Box 1910, Brown University, Providence, RI 02912.

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

[1]The Pascal notation for this would be `X ↑ .car`. In C, `X → car` or `(*X).car`.

[2]Checking for pointer equality is also allowed but is irrelevant to this analysis.

and we do not deal with aliasing between variables. We assume that some form of type checking (either static or dynamic) is performed to assure that pointers are used only in this way.

Program structure is modeled by a *control flow graph* (CFG). Programs can be composed of most common control structures, including those that give rise to irreducible control flow graphs. Interprocedural analysis is discussed in Section 6. The structures that cannot be handled in this framework are those that rely on label or procedure variables and self-modifying code.

For convenience, we require that the statements in the program contain only simple binary expressions and binary access paths. Complicated expressions must be broken into a series of binary expressions that assign to unique temporary names, as shown in Figure 1. Because of this simplification of expressions, access paths are also very simple. An access path has the form `A.B` where `A` is a simple variable name and `B` is a field name (either car or cdr in our examples). We also break statements so that any statement that allocates storage must assign it to a simple variable. Furthermore, we separate the allocation of new storage from its initialization.

```
A + B + C                   T1 ← A + B;  T1 + C
X.car.cdr                   T2 ← X.car;  T2.cdr
cons(cons(X,Y) Z)           T3 ← cons();
                            T3.car  ← X;
                            T3.cdr  ← Y;
                            T4 ← cons();
                            T4.car  ←   T3;
                            T4.cdr  ← Z;
Z.car ← cons(X,Y)           T5 ← cons();
                            T5.car  ← X;
                            T5.cdr  ← Y;
                            Z.car ← T5
```

Figure 1. Complicated and simplified expressions.

Sections 4 and 5 utilize features of *static single assignment* (SSA) form [CFR$^+$89a, CFR$^+$89b]. Here we give a brief review.

The transformation of a program into SSA form retains the semantics of the original program and has two useful properties:

1. Each programmer-specified use of a variable is reached by exactly one assignment to that variable.

2. The program contains $\phi$-functions that distinguish values of variables transmitted on distinct incoming control flow edges.

A $\phi$-function has the form $U \leftarrow \phi(V, W, \ldots)$, where $U, V, W, \ldots$ are variables and the number of operands $V, W, \ldots$ is the number of control flow predecessors of the point where the $\phi$-function occurs. The control flow predecessors of each point in the program are listed in some arbitrary fixed order, and the $j$-th operand of $\phi$ is associated with the $j$-th predecessor. If control reaches the $\phi$-function from its $j$-th predecessor, then $U$ is assigned the value of the $j$-th operand. Each execution of a $\phi$-function uses only one of the operands, but which one depends on the flow of control just before the $\phi$-function.

The transformed program is defined to be *in SSA form* if, for every original variable $V$, $\phi$-functions for $V$ have been inserted and each mention of $V$ has been changed to a mention of a new name $V_i$ such that the following conditions hold:

1. If a *CFG* node $Z$ is the first node common to two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes $X$ and $Y$ containing assignments to $V$, then a $\phi$-function for $V$ has been inserted at entrance to $Z$.

2. Each new name $V_i$ for $V$ is the target of exactly one assignment statement in the program text.

3. Along any control flow path, consider any use of a new name $V_i$ for $V$ (in the transformed program) and the corresponding use of $V$ (in the original program). Then $V$ and $V_i$ have the same value.

## 2.2 The Storage Shape Graph

For each statement in the program, we construct a *storage shape graph* (SSG). When the analysis is complete, each SSG is a finite, conservative summary of all pointer paths into and through allocated storage that could arise by executing any path to (terminating after execution of) the statement for which the SSG was computed. All SSGs for a program contain a set of nodes: one node for each simple variable (*variable* nodes), nodes corresponding to heap allocated storage (*heap* nodes), and a special node representing all atoms including `nil`[3]. Heap storage is typically allocated by statements such as `new` in Pascal and `cons` in Lisp. Each node contains fields reflecting the record structure of the memory allocated by the program; for the purposes of this paper, these are **cons** nodes with fields **car** and **cdr**. Variable nodes and fields of heap nodes in the SSG model locations in run-time data structures where pointers may be stored.

Edges in an SSG model pointer values. Edges in an SSG are directed towards heap nodes from variable nodes and from fields within heap nodes. Edges are not directed from fields toward variable nodes, and variable nodes do not contain fields. Generally, more than one edge can leave a field or variable because the pointer(s) that edges model can have different values corresponding to different paths through the program.

The SSG after $s$ models the heap for all paths ending at $s$. For any particular path ending in $s$, there may be nodes and edges in the SSG after $s$ that are not required for that path. The edges represent the information content of the analysis. The SSG contains the following information about data structures at run time: if two access paths through the SSG end in different places they cannot end in the same place in the actual heap. The useful information is the negative information; one SSG is more precise than another if it lacks an edge.

Note that in general more than one graph can satisfy these properties; different graphs can have different sizes and can preserve different information. Choice of the right graph is not a well-defined problem, since there is always the possibly that adding one more node will preserve some crucial piece of information.

Cycles in an SSG may represent cycles in a run-time data structure, but they may also represent an unbounded acyclic data structure. The SSG in Figure 2 describes the data structures possibly addressed by variables `W` and `X` after executing Statement 8 of the program in that figure. This SSG shows that a data structure addressed by `W` at that point in the program could be a list of even length or a cycle of even length (or any other data structure that can be summarized by this graph). In either case, it is known that the even and odd `cons` cells on the spine of the data structure are not aliased, and that the `car` fields of those cons cells point alternately to 1 and 2. It is possible, but

[3]Merging all atoms does not affect the quality of alias analysis.

not certain, that X and W.cdr point to cells that are aliased. In fact, inspection of the program reveals that X and W.cdr always point to the same cell at this location, and the data structure is in fact a list, but here the SSG does not provide that much information. In Section 4.3 we show how to obtain that information.

```
1   W ← nil
2   while (...)
3     X ← cons()
4     X.car ← 1
5     X.cdr ← W
6     W ← cons()
7     W.car ← 2
8     W.cdr ← X
9   end
10  Z ← W
11  while (...)
12    Y ← Z.cdr
13    Z ← Y.cdr
14  end
```

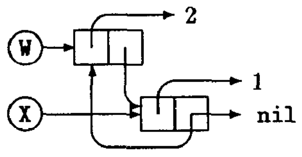

Figure 2. Two-element alternating list and its SSG after Statement 8.

The following three operations are the only ways to construct an SSG: (1) nodes may be added, (2) edges may be added or (3) nodes may be *merged*. When two nodes are merged, all edges pointing into either of them point into the merged node and, field by field, all edges pointing out of them must point out of the merged node.

## 3 The Simple Algorithm

In our simple algorithm, all SSGs for a program contain the same set of nodes: one node for each simple variable (*variable* nodes) and one node for each statement in the program that allocates a data structure, such as new in Pascal and cons in Lisp (*heap* nodes). A naive representation of the graphs built for a program of $S$ statements will contain $S$ SSGs, and each of these will contain $O(S+V)$ nodes, for a total of $O(S^2 + V \times S)$ nodes. A more efficient representation (for typical programs) is described in Section 5.

In this simple algorithm, the SSG $G$ for a statement $s_1$ contains the following information about data structures at run time:

- If $A$ is a variable and $B$ is the heap node corresponding to statement $s_2$, and if there is an edge from $A$ to $B$ in $G$, then after some execution of $s_1$ the variable $A$ *may* point to some data structure allocated at statement $s_2$.

  If there is no such edge, then after all executions of $s_1$ the variable $A$ cannot point to any structure allocated at statement $s_2$.

- If $A$ is field $F$ in the heap node corresponding to statement $s_2$ and $B$ is the heap node corresponding to statement $s_3$, and if there is an edge from $A$ to $B$, then after some execution of $s_1$ the $F$ field of some structure allocated at $s_2$ *may* contain a pointer to some structure allocated at $s_3$.

  If there is no such edge, then after all executions of $s_1$ there is no structure allocated at $s_2$ whose $F$ field contains a pointer to any structure allocated at $s_3$.

The motivation behind never merging nodes from different cons statements is that nodes allocated in different places probably are going to be treated differently, while all nodes allocated at a given place will probably be updated similarly.

### 3.1 The Simple Fixed-Point Algorithm

The initial, optimistic approximation for a statement's SSG is the graph containing all the nodes allocated in the program, but with no edges leaving any of them. So, each SSG is originally identical with the same set of nodes. The most conservative approximation (and the one containing the least information) is the graph with all possible edges (this is the complete graph, except that edges cannot be directed from fields to variable nodes). The analysis proceeds by adding edges to the SSGs, and its goal is to find the SSG with the smallest number of edges that is still a conservative approximation to the actual storage; i.e., the SSG that has every edge it needs, but no more.

The algorithm presented here is a modification of Wegbreit's iterative data flow analysis technique [Weg75]. We start out with a worklist that is initialized to contain the entry statement in the program. At each step we take a statement off the worklist and attempt to evaluate it. If the statement can be evaluated *and* some new edges are added to its SSG, we put the successors of the statement onto the worklist. The algorithm terminates when the worklist becomes empty.

If the statement is a join point in the program (i.e., if two or more statements have this statement as a successor), then each node in the SSG entering this statement will have, as its edge set, the union of the edge sets for the corresponding nodes in the SSGs for the predecessor statements.

If the right hand side of an assignment is a variable X, the r-value is the set of nodes reached by edges leaving X. If the right hand side of an assignment is a field X.cdr, the r-value is the set of nodes reached by edges leaving fields in *l-value*(X.cdr). *l-value*(X.A) is the set of fields labeled A in nodes reached by edges leaving the variable X, as shown in Figure 3. If the right hand side of a statement is a cons, the r-value is the heap node corresponding to that statement.

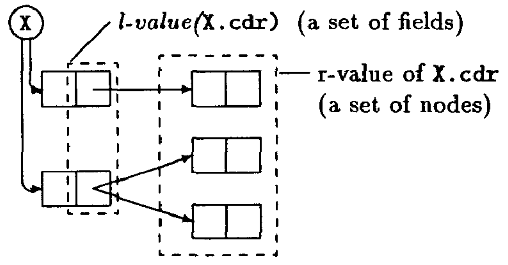

Figure 3. Example of *l-value* and *r-value*.

Each statement modifies edges in the graph in a manner reflecting the semantics of that particular statement. For assignment to a simple variable, the semantics are to *replace* the edges leaving the variable with the r-value for the statement. This type of assignment is called a ***strong update***. For assignment to a field of a node addressed by a variable (e.g., X.cdr ← Y assigns to the cdr field of nodes addressed by X), the semantics are to *add* the r-value to the sets of edges that are directed from fields addressed by the left hand side. Assignment in this fashion is called *weak update*, since edges directed from the addressed fields are not deleted before the assignment occurs. Under appropriate conditions, strong updates can be performed on assignment to fields so as to yield better information, as discussed in Section 4.2.

Some intuition for why this technique works well can be gained by considering type schemes. While the principal use of type schemes is to detect errors at compile time, they have also been used by compilers for simple alias analysis. The idea is that a modification to one type of structure cannot be applied to a structure with a different type. If a pointer can point to only one type, then an update from that pointer cannot modify any objects of different type.

In a language like Pascal, each type corresponds to a disjoint set of nodes. The set of nodes corresponds to the locations at which a type can be allocated. An allocate statement in Pascal takes a pointer as an argument; since a pointer can point to only one type, the set of nodes is disjoint. The problem we face in designing a pointer analysis scheme is choosing which nodes to merge and which not to merge. By choosing to merge only nodes that come from the same allocation site, we do not merge nodes that would (or could) be given different type declarations.

In Pascal, a declaration limits the number of different types a pointer can point to, and compile time analysis checks that they cannot point to additional types. In our algorithm, we discover at compile time the set of nodes a given pointer can point to. Any program that would cause our algorithm to add an edge in the SSG between a node within the set for one type to a node of a type which it cannot point to would be flagged as an error by a Pascal compiler. Thus, if at the end of our analysis, all nodes of a given type were merged, there would be no more edges between them than those specified by a Pascal type declaration.

Type declarations often give a good description of the shape of data structures. Our SSGs differ principally in that there is a different SSG for each statement in a program, and it would be impractical to have the programmer include new declarations of types for each statement.

## 4 Extensions to the Simple Algorithm

In this section, we present several synergistic extensions to the simple algorithm: allowing more than one SSG node to be kept per allocation statement (Section 4.1); allowing strong update to be done to SSG nodes in certain cases (Section 4.2); and associating reference counts with the SSG nodes to allow us, for example, to detect tree-like structures (Section 4.3).

### 4.1 Maintaining Multiple Instances of a cons Node

In the simple algorithm (Section 3), we assumed that the SSGs contain only one node for each cons statement. While we can distinguish structures composed of elements allocated at different cons statements, we need additional detail to determine other information. We cannot deduce that the data structure is acyclic, and thus the cons cells pointed to by X on different iterations of the loop are always different.

Our solution is to allow SSGs to contain multiple nodes from a single cons statement. Each of these nodes is called an *instance* of that cons statement. The execution of a cons statement causes a new instance to be created, and a pointer to that instance is assigned to the target of the left hand side of the assignment statement.

A natural question is, "How many instances are enough?" The answer depends on the context. What is desired is to keep only the *interesting* instances.

We classify variables into two categories: (1) a *deterministic* variable points to one SSG node and may possibly point to nil, and (2) a *non-deterministic* variable points to more than one SSG node. Being deterministic is an important property for variables to have. Section 4.2 exploits this property to perform strong update.

We call an SSG node interesting if it is pointed to by a deterministic variable. We apply the rule that uninteresting instances are merged as soon as they become uninteresting. We use the cons statement to generate new instances and the assignment statements and meet operations to merge instances. We still require that only instances of the same cons statement can be merged together. There can be at most as many SSG nodes with deterministic variables pointing into them as there are variables in the program. For each cons statement there can be at most one instance not pointed to by a deterministic variable. We call this instance the *summary node*. Thus, the total number of nodes in any SSG is bounded by the sum of the number of allocation statements and the number of variables.

Variables can change their values at two places: assignments and join points. At these points, an SSG node that no longer is pointed to by a variable becomes uninteresting and the SSG node may be merged.

#### 4.1.1 Merging After Assignments

An SSG node $n$ becomes uninteresting after an assignment statement $s$ assigning to variable V if (1) before evaluation of $s$, V was a deterministic variable pointing to $n$ and (2) after evaluation of $s$, no deterministic variable points to $n$.

If $n$ has just become uninteresting and there are any other uninteresting instances of the cons statement of which $n$ is an instance, they are merged.

#### 4.1.2 Merging At Join Points

The meet operation is performed at any join point in the program. A very naive meet would union the sets of nodes from the entering SSGs and fix up the variables to point to the union of the nodes they pointed to in the entering SSGs. Unfortunately, this meet is unacceptable, since the number of nodes in the SSGs will grow without bound. Therefore, SSG nodes must be merged when performing the meet.

Our goals in constructing this merging scheme are:

1. The number of SSG nodes is bounded. In particular, at the end of the merge, there is at most one instance of a cons statement without a deterministic variable pointing to it.

2. The largest number of deterministic variables are preserved.

3. As few edges as possible are added to the graph.

4. The matching can be performed efficiently.

5. Preserve information can be used to perform strong updates to fields (see Section 4.2).

We wish to form a meet between the two SSGs in Figure 4, and we wish to choose the "best" merge. The choice in Figure 5 is unacceptable both because general application of this strategy leads to unbounded graph growth, and because all variables are made non-deterministic. The two choices in Figures 6 and 7 are better, but without additional information an arbitrary choice must be made. Figure 6 keeps X and Y deterministic, while Figure 7 keeps Z deterministic. One way to choose is to impose an order on the variables and always perform SSG node merges in the same (variable-by-variable) order. Another choice is to give up, and use Figure 8. We suggest another heuristic, however, that we believe is more appropriate.

At a join point at which SSG nodes are being merged, psuedo-assignments ($\phi$-functions) are also inserted for variables. The variables that deterministically reference SSG nodes at the CFG predecessors can be separated into those that are arguments of $\phi$-functions at the join point and those

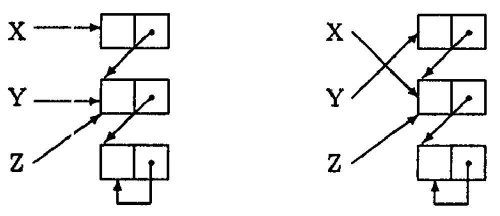

Figure 4. Two graphs to merge.

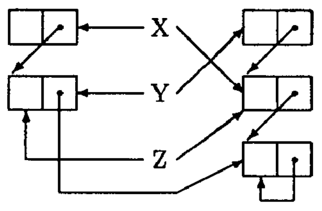

Figure 5. A large, unacceptable merge.

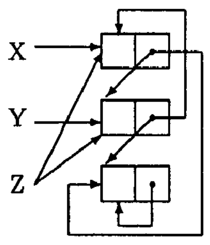

Figure 6. X and Y remain deterministic.

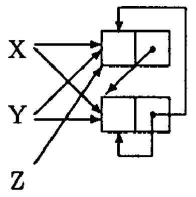

Figure 7. Z remains deterministic.

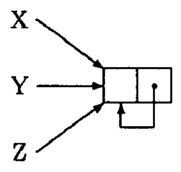

Figure 8. No deterministic variables.

that are not[4]. If references to SSG nodes from variables involved in psuedo-assignments are ignored, merging conflicts will not arise.

Therefore, the first step in merging two SSGs is to pair up SSG nodes in the CFG predecessors on a per-allocation-site, per-variable basis for the variables that are deterministic and not involved in psuedo-assignments. Doing this will not lead to any conflicts, but it may cause some other variables to become non-deterministic. For example, Figure 6 results if X and Y are not involved in psuedo-assignments but Z is; X and Y remain deterministic, but Z becomes non-deterministic. Figure 7 results if Z is not involved in a psuedo-assignment. Figure 8 results if all three variables are involved in a psuedo-assignment.

The second step examines any SSG nodes pointed to by deterministic variables not paired by the first step; that is, those that occur in $\phi$-functions. Ignoring non-deterministic variables, two SSG nodes $n_{left}$ and $n_{right}$ allocated at the same statement may be paired up if the sets of variables[5] referencing $n_{left}$ and $n_{right}$ are identical. If the sets of variables are not identical, but the differing variables do not point to *any* node in the other SSG (that is, they are nil, atom, or empty), then the two nodes may be paired up. For example (or as a special case), if a variable is the sole reference to a node $n$ in one SSG and nil in the other SSG, then $n$ is not merged into its corresponding summary node.

After all those SSG nodes have been paired up and merged, all the remaining SSG nodes are merged into their summary nodes on a per-statement basis.

The above merging conditions ensure that only one instance of an allocation site's nodes is not referenced by any deterministic variables (that is, in any SSG there can be at most one cons instance for each variable and one cons instance for each statement). This does not imply that the number of instances cannot grow at the merge point. Consider the example in Figure 9. As the loop (a join point) is entered from above, both X and Y are redefined in $\phi$-functions. In the SSG transmitted from the preceding statements, X addresses a cons node (call it $n_1$) and Y is nil. In the SSG transmitted from the end of the loop, X is nil and Y addresses a cons node (call it $n_2$). Should $n_1$ and $n_1$ be paired up, merged into a summary for their allocation statement, or left separate? By the meet given above, they remain separate, even though both nodes model the same run-time location. This is counterintuitive but correct, because at the top of the loop X and Y never simultaneously refer to the same address (unless it is nil).

```
X ← cons()
Y ← nil
while()
   Y ← X
   X ← nil
end
```

Figure 9. Program in which the number of instances after the join is greater than before the join.

## 4.2 Strong Updates

An update statement can either be a *weak update* or a *strong update* operation. In strong update, the field that has been changed must point to a new set of locations rather than the old. In weak update, the field that has been changed points to the new set of locations in addition to the old. It is desirable that as many as possible of the updates be strong since a strong update provides better information. Unfortunately, while it is always correct to perform a weak update, it is not always correct to perform a strong update.

Strong update can be applied in two situations. First, an assignment statement can perform a strong update if (1) its l-value is a single SSG node and (2) that node

[4]For the sake of simplicity, assume that the $\phi$-functions for variables were discovered in a pre-pass, not by the incremental algorithm described in Section 5.

[5]To be precise, the sets of SSA variables with renaming subscripts stripped off.

corresponds, on all execution paths, to at most one location in the heap. If our algorithm cannot guarantee that both these conditions are met, a weak update must be performed.

Condition (1) is satisfied if the assignment is to a variable or if the assignment is to a SSG node reached via a deterministic variable.

Condition (2) is satisfied for SSG nodes that represent single heap nodes. A node in the SSG corresponds to a single heap node unless *on some execution path* it has been merged with another SSG node.

Careful treatment of merges allows detection of SSG nodes that represent single heap instances. All instances are flagged as either *single instance nodes* or *multiple instance nodes*. A single instance is guaranteed to correspond to at most one heap location. Multiple instance nodes may correspond to any number of heap locations.

An instance is generated at a cons statement, at which point it corresponds to a single heap location. An instance can become a multiple instance node only if it is merged under certain conditions. If instances $n_1$ and $n_2$ are merged to form instance $n_3$, $n_3$ is a multiple instance node if any of the following conditions hold:

- Instances $n_1$ and $n_2$ are merged after an assignment statement (i.e., not at a join statement).
- At least one of $n_1$ and $n_2$ is already marked as multiple instance nodes.
- Instances $n_1$ and $n_2$ were associated with the same entering CFG edge at a join point.

The second situation in which strong updates can be applied arises when a variable V points to several SSG nodes and there are *no* other edges into any of those SSG nodes.[6] The collection of SSG nodes models a run time situation in which one SSG node is referenced by V and the rest are unreachable (e.g., garbage for the collector). Whatever node V references will change, and the rest are not observable, so it is safe to update all nodes referenced by V.

Our propagation algorithm is organized around a worklist. In order to deal with strong updates, the order in which the statements are processed must be handled carefully. Two invariants are maintained: (1) when a statement is visited, at least one of its immediate predecessors has already been visited, and (2) an edge, once added to a statement's SSG, is never deleted. Because an SSG contains a finite number of edges, the second condition allows a trivial proof of termination.

```
X ← cons()

if FirstTime
   then do
      Y ← X
      FirstTime ← false
   end
X.car ← ...
Y.car ← ...
```

Figure 10. Program containing a legal use of an uninitialized variable.

To maintain the second invariant in the presence of strong update, processing of some update statements must block temporarily. In some cases, it is not possible to evaluate the l-value of a statement of the form Y.car because Y may have no value or the value nil. This situation arises only if the program is errant or if the algorithm will later discover a path on which Y has a different value.[7]

A strong update to a node *n* stops edges leaving *n* from propagating from an earlier SSG to a later SSG. Any right hand side of the form Y.car, where Y has no value or has the value nil, is the site of a potential strong update. Rather than allow edges to be propagated past a statement that may later perform a stong update, we defer evaluation until Y is given a value. This problem arises when the iteration has taken a path through the program in which an uninitialized variable is accessed; as shown in Figure 10, this can happen in a correct program. Such fragments can commonly be found inside subroutines and loops. Most static analysis techniques (including this one) ignore the values of predicates. They are unable to understand that the FirstTime flag assures that the assignment to Y.car is safe.

A traversal of the program in Figure 10 might reasonably start by processing it as if FirstTime were false; first the assignment to X, then the if test, and then the statements following the if. On such a traversal, interpretation of the assignment to X.car would add some edges to the SSG node of the first statement. This is followed by the assignment to Y.car, but here processing must block because Y still has no edges leaving it.

Interpretation proceeds on some other path, and at some later point the if-true branch will be processed, causing Y to become defined. The statements following the if are added to the worklist (again), but this time Y.car can be correctly evaluated.

If the algorithm terminates without visiting a statement *s*, then all executions of the program that include a path through *s* will raise an error (or have undefined behavior.)[8]

We keep extra instances of SSG nodes that have variables pointing to them because these instances correspond to the heap cells that the program can manipulate easily. By keeping these instances separate, strong updates can be performed on them and their references determined before they are merged with the other nodes.

## 4.3 Heap Reference Counting

Additional information about storage structure can be obtained by following the storage-shape analysis presented here with a reference counting analysis similar to that of Hudak [Hud86] and Boehm and Hederman [BH88, Hed88] (Barth uses a somewhat similar analysis for "compile-time garbage collection" [Bar77].) The combined analysis can discover that a list is a "true list", and thus makes possible optimizations like those described and implemented by Larus [Lar89], without requiring programmer declarations.

Heap reference counting statically approximates the number of references to a heap cell from other heap cells; that is, it models a run-time count of heap references without counting references from variables[9]. The goal of heap reference counting is to discover portions of the storage shape

[6]Because of the meet rules, these conditions are satisfied only when each of the SSG nodes comes from a different cons statement and each of the SSG nodes is a single instance node.

[7]An update to a deterministic variable may attempt to update nil. In this case, the heap location cannot be modified and the strong update would appear to be unjustified. However, any attempt to update nil is necessarily an error and we do not worry about getting correct results on errant programs.

[8]Note that the algorithm will not detect all errors of this sort. We think that the subset it does detect is probably uninteresting.

[9]This is similar to the actual reference counting schemes described by Deutsch and Bobrow [DB76] and Rovner [Rov85].

graph that model lists or trees (as opposed to graphs with sharing or cycles); that is, the goal is to discover portions of the graph in which all nodes have heap reference count equal to one.

The reference count lattice is $\{0, 1, \infty\}$ with meet operation max. Each SSG node has an associated heap reference count (HRC). The reference count of a node $n$ is modified whenever a field of an SSG node is updated to point to $n$. If the update adds a pointer to $n$, the HRC for $n$ is incremented ($\infty + 1 \rightarrow \infty$). We do not attempt to reduce heap reference counts. Whenever two nodes are merged (at either a join point or an assignment), the HRC for the joined node is the maximum of the HRCs for the nodes that were joined.

HRC becomes a more powerful tool when combined with the use of multiple instances, as shown in Figure 11. This example builds a list in the usual way, by allocating new SSG nodes and using the previous list as the cdr of the new list. With this storage shape graph, we can annotate each node with a HRC (shown above each node).

After statement (2) X addresses either nil or the most recently allocated cons cell, which has a reference count of zero. The older cells each have a reference count of one.

Statement (3) allocates a new cell and the SSG gets bigger. Since X and T point to different nodes, neither is merged with the summary node. The new node, lacking any heap references, has a reference count of zero.

Statement (4) assigns the head of the old list into the cdr field of the new cons cell; this increments its reference count in the resulting graph.

Statement (5) makes X point to the newly allocated cell. After statement (5) has been executed, no variables point to the middle cons cell, and it is merged with the summary node. The maximum (meet) of one and one is one, so the reference count of the summary is unchanged.

Notice that the graph referenced by X.cdr (if not nil) has reference count of one throughout. Any reachable nodes are unshared if a subgraph of the storage shape graph has the following properties:

1. The root of the subgraph is not reachable from the other SSG nodes in the subgraph;

2. all SSG nodes except the root have HRC = 1.

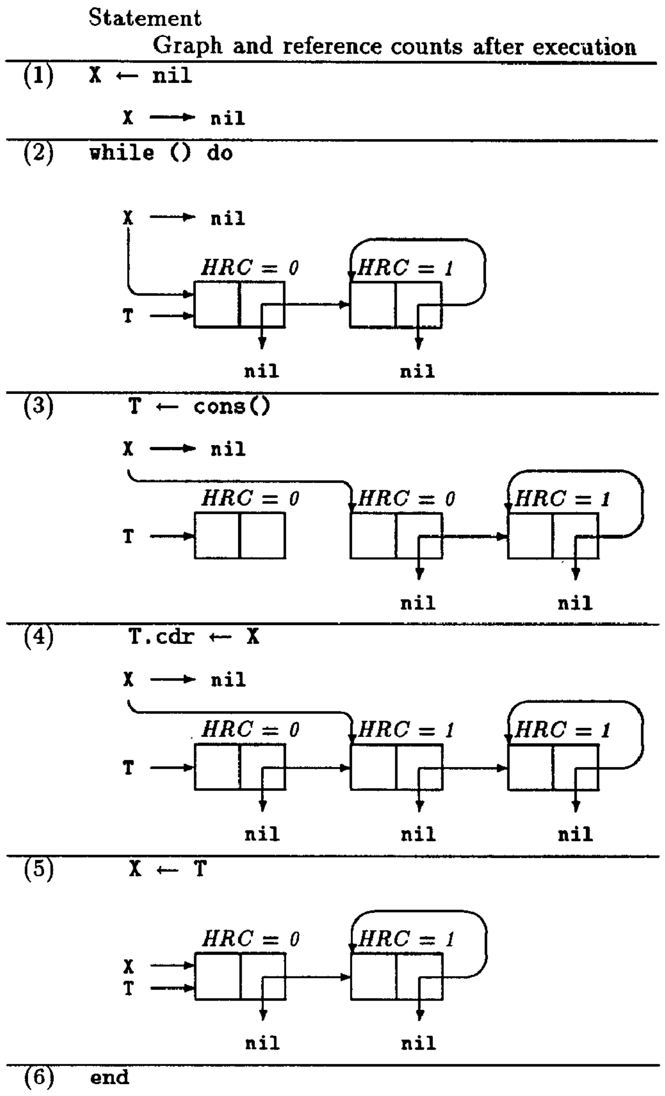

Figure 11. List builder and reference counts.

The intuition behind this is that if all nodes in a connected graph have HRC = 1, then the graph is either a tree or a simple cycle. If the root node is not part of a cycle, then what is reachable (i.e., has real reference count not equal to zero) must be a tree. Given a storage shape graph annotated with reference counts, a correctly rooted path that lies within a region of HRC = 1 nodes is guaranteed to be a noncyclic at run time. Note that this is not the case for paths whose first edge is from a variable directly into the non-rooted subgraph; these may be cyclic (but the presence of a reference from a variable does not invalidate the property for a correctly rooted path).

This analysis is capable of discovering that a "tree" built from the top down (as opposed to the bottom-up construction in the example here) is in fact a tree, and can also discover that the destructive concatenation of two proper lists is also a proper list.

Further refinements of reference counting can be used to extend these results. Each node can also maintain a count for the number of references from a specific field of any node allocated at a given statement; that is, an additional count for each valid statement-field combination. With this extension, it is possible to discover non-cyclic paths through cyclic structures. For example, a doubly linked list is cyclic, but paths through the "next" field are not cyclic. For nodes modeling this list, the count of references from next fields will be one, and as long as paths traverse only edges rooted at next fields, different paths (rooted in a cycle-free node, as above) will address different nodes.

The meet of two of these counts is max, and a count is incremented whenever a reference from a corresponding field is created by an update. If counts are recorded only when they exceed zero[10], then only a linear increase in space will result since there will always be at least as many edges as counts.

## 5 An Efficient Implementation

The simple conceptual version of our algorithm presented in Section 3 may be too slow for practical use. Before we present our efficient algorithm, however, it is useful to analyse the simple algorithm.

If a program has $S$ statements, then there can be as many as $S$ allocation statements and $S$ summary nodes. A variable can deterministically point to only one node, so

[10] Actually, these counts need be maintained only when they exceed one, since no edges from fields-of-nodes-from-a-given-statement means zero, some edges means one, and more is a special case.

there can be $2 \times V$ non-summary nodes in the SSG, where $V$ is the number of variables. Thus, the number of nodes in each SSG is bounded by $S + V$. We denote the maximum of the in-degree of a node and the maximum out-degree of fields within nodes and variables by $T$, and thus the number of edges in each SSG is bounded by $(S + 2 \times V) \times T$. In the worst case $T = S + V$. Since there are $S$ different SSGs, the storage required can be $O(S \times (S + V) \times T)$.

Each statement can be visited as many times as it has edges in its SSG, $O((S + V) \times T)$. Every time a join node is visited, the SSGs from the two entering edges are merged[11], so the work involved can be $O((S+V)\times T)$. There are $O(S)$ statements, and hence the total work can be $O(S\times(S+V)^2 \times T^2)$. While other work is done, this is the dominant cost.

We improve the algorithm in two ways:

1. We modify the data structures so that they take advantage of any sparseness in the SSGs, as described in Section 5.1.

2. We modify the algorithm so that changes to the SSG are propagated directly to where the information is used, as described in Section 5.2.

Each of these modifications allows us to replace a factor of $S$ by a smaller term. Figures 12 through 17 contain pseudocode for the algorithm.

## 5.1 Using Sparseness

The SSGs are sparse in two ways:

1. Each SSG is likely to be sparse. Empirical studies of heap allocated structures have found that the reference counts of over 90% of the cells never exceed one [CG77], and systems relying on this assumption have worked well [DB76, SCN84, Rov85]. Thus, one expects that an SSG should also be sparse. Given that the SSG is adequately sparse, we can assume that $T$ is $O(1)$.

2. All SSGs have the same nodes and differ only in their edges. Because each SSG is sparse, an update tends to change only a small number of nodes. Hence, the edge set of an SSG is very similar to that of the SSG's preceding statements.

So far, the data structure used to represent the information algorithm has one graph per statement. Each graph has the same set of nodes, but the set of edges leaving corresponding nodes differs. Here we invert that representation using only one set of nodes. Each node has a list of its different edge sets. Associated with each edge set is the location (the statement) at which that set is valid. A node has an edge set at a location $l$ if the edge set for that node differs from any of the edge sets of the immediate predecessors of $l$ in the control flow graph.

The question is: which statements need edge sets at which SSG nodes? In straight-line code, this is easy to answer: the only statements that need edge sets are the updates to that SSG node. Real programs contain branches and joins, however, and to handle them, we use an idea borrowed from SSA form. In translating to SSA form, a $\phi$-function for variable $V$ is inserted at any join point that different assignments to $V$ reach. Here, we treat each SSG node as if it were an SSA variable. An update to an SSG node can be thought of as an assignment to an SSA variable. At the control flow graph join points, where different updates to the SSG node meet, the edge sets must be unioned together. In SSA form a $\phi$-function would be inserted to join the differing assignments. Here, a new edge set is inserted to represent the union of the incoming edge sets. We call these new edge sets $\phi$-functions, as in SSA form.

Just as in SSA form, it is desirable to insert the minimum number of $\phi$-functions. Unfortunately, the efficient SSA calculation algorithm cannot be directly used here. In computing SSA form, the side effects are visible before the computation begins, and an assignment to a variable modifies that variable only. Here, however, side effects are discovered as the analysis proceeds; an assignment to a field in one SSG node (e.g., `x.car`) *may* later be an assignment to the same field of other nodes. Because of this, $\phi$-functions will be inserted on the fly as side-effects are discovered.

Some aspects of the SSA algorithm are used. The *dominance frontier* is computed and used to determine where to locate the $\phi$-functions. The dominance frontier $DF(X)$ of a CFG node $X$ is the set of all CFG nodes $Y$ such that $X$ dominates a predecessor of $Y$ but does not strictly dominate $Y$. In SSA form, the defining assignment dominates all of the regular uses, and dominates all of the immediate predecessors of $\phi$-function uses.

The algorithm for computing SSA form connects uses to definitions in a single pass over the program. We do not want to make a pass over the program every time we discover a new assignment, so a new data structure is required. After $\phi$-functions have been inserted, the defining assignment for a use can be located by searching up the dominator tree for the nearest assignment. A new data structure is used to make this search efficient.

Let $T$ be a tree, which we call a *master tree*, with a set of nodes $N$. $T'$ is a skeleton tree for $T$ if (1) the set of nodes $N'$ in $T'$ is a subset of $N$ and (2) the parent of $n$ in $T'$ is the closest ancestor of $n$ in $T$ that is in $N'$. We need to perform several operations on skeleton trees: insert a new node, find parent, find all children. These operations can all be performed in a straightforward manner[12] in time $O(|N'|)$, or in time $O(\log(|N'|))$ time using the techniques in [CWZ90].

In this algorithm, the master tree is the dominator tree of the control flow graph. For each node in the SSG, we maintain a skeleton tree containing the use and definition locations of the node. Each definition point in this tree contains an SSG edge set. This tree is also used to find the uses reachable from any definition. To determine the edges at a use, the nearest definition point is located by walking up the skeleton tree from the use point until the edge set is found.

Each SSG has the same set of nodes. To take advantage of sparseness, we require that the value of a node in the SSG for statement $S$ can usually be derived from the value of the corresponding node in the SSG(s) for the CFG predecessor(s) of $S$. The corresponding node can be easily found in the simple algorithm because each SSG node corresponds to a single cons statement.

We allocate for the whole program a fixed set of SSG nodes, one SSG node for each cons site to hold the summary, one SSG node for each variable, and one SSG node for each variable to point to deterministically, which we call

[11] We assume for simplicity that no more than two edges come into a join node.

[12] Find parent and find all children are trivial. Insert can be done as follows: by using a preorder numbering of the master tree and a count of the number of descendants of a node in the master tree, one can in unit time tell whether one node is a descendant of the other. To insert a node in the skeleton tree, walk down from the root of the skeletal tree until you find a node that is the parent of the node you are inserting.

that variable's cons cell. Each allocation site has a single summary node. The summary nodes and the SSG nodes corresponding to variables match up in a one-to-one manner between statements. In the variable's cons cell, the correspondence is not straightforward. Consider the following code sequence:

```
1    X ← cons1
2    Y ← X
3    X ← cons3
```

In statement (1), X's cons cell holds the value of $\text{cons}_1$. In statement (2) X and Y must point to the same cell, and we choose (somewhat arbitrarily) to use X's cons cell, since the value is already there. In statement (3) X's cons cell must be used for the value of $\text{cons}_3$. The value produced by $\text{cons}_1$ is now stored in Y's cons cell. To move the cell from X's cons to Y's cons requires copying the values from cons to Y's cons, finding all pointers to X's cons cell, moving those pointers to Y's cell, and initializing all of the fields in X's cons cell.

Thus, certain operations on one SSG node may bring about a causing a certain amount of *shuffling* in the SSG nodes that are deterministically pointed to. The variables and SSG nodes that point to shuffled SSG nodes must actually be updated to reflect the change. These updates are treated like all other updates to variables' SSG nodes and their impact must be propagated throughout the program. This shuffling effect can happen not only at assignments but also when instances are merged.

To perform the shuffle operation, we need to find all nodes pointing at a given node. To find them efficiently, a list can be maintained for each node of every reference to that node. Every time a reference is added to that node, this list needs to be updated. The list can be treated as an extra field of a node, and maintained on a skeleton tree. At a skeleton tree node for this field, a new copy of the list is maintained with whatever changes are appropriate.

When an assignment to a variable takes place, and that variable no longer deterministically points to its former cons cell, that cell must either be shuffled to a different variables cons cell or merged with the summary node. A pass is made over the list to see if any variables point to it deterministically, and the first one found is choosen. All changes to various fields caused by the shuffling are propagated in the normal manner.

At a join point, a variable may become nondeterministic. This forces a shuffle if a node was stored in that variable's cons cell. A shuffle in the statements leading to a join point may also trigger a shuffle at the join point. But it is also possible at a join point to have the node stored in different variable's cons cells on different entering CFG edges. In that case, one of the cons cells is chosen at the join point (if one had been chosen from one of the entering CFG edge's, and the other CFG edge changes, we stay with the old choice). For this SSG a *forwarding* pointer is kept from the node not chosen to the one which is chosen. When a change is propagated to the node not choosen, rather than update that node, the algorithm must recognize that there is a forwarding pointer and update the node pointed to by the forwarding pointer. If a shuffle is performed at the join point, moving a node, all forwarding pointers to the node must also move.

Heuristics may improve shuffling behavior. Some possibilities include an arbitrary ordering of the variables to determine which is chosen at a shuffle, and a preference for shuffling to variables not involved in pseudo-assignments.

## 5.2 The Propagation Algorithm

In the simple algorithm, whenever the evaluation of an assignment changes the edge set for the target, the control flow graph successors of the basic block are added to the worklist for future reevaluation. In the efficient algorithm, on the other hand, only the statements using the values computed at the assignment are queued for reevaluation. Thus the reevaluation need not be done on statements between the definition site and the uses.

Strong update causes a problem: a basic block containing an apparently erroneous update is treated as not executable, and attributes do not propagate through it. A technique borrowed from constant propagation [WZ88] is used to mimic the careful evaluation required by strong updates. The incremental algorithm has two global worklists, and elements are chosen from whichever one has elements of work that need to be done. The first worklist contains basic blocks; the second worklist contains uses whose values may have changed since they were last visited.

We track whether a basic block has ever been visited and whether it has been successfully executed to completion. When a basic block is pulled off the worklist it is marked as "visited" and is traversed statement by statement, locating uses and definitions within the basic block and queueing uses onto the second worklist. In certain situations (specifically, field selection through an uninitialized variable), processing cannot proceed and the basic block must be marked as "visited" but not completed. If processing reaches the end of a basic block, the basic block is flagged as successfully executed to completion. When a basic block is successfully completed, its CFG successors are placed in the first worklist.

When a definition is discovered, $\phi$-functions must be inserted at the blocks in its dominance frontier. This insertion is deferred until an error-free path is found from the definition site to the block in the frontier. The deferred insertion is accomplished by queueing $\phi$-functions in a per-block worklist at the predecessor(s) of the block in the frontier. If the predecessor executes to completion, then a path has been found and the deferred $\phi$-functions can be inserted.

When an item is pulled off the second global worklist it is tested to see if it is the same as before; if not, the effect of its new value is propagated. Any updates done as part of this statement cause additional items to be placed on the second worklist. If a previously uninitialized variable appearing in a selection becomes defined, then a previously stuck statement may become executable and the basic block may be executed to completion and its successors added to the first worklist.

When merging SSG nodes $n_1$ and $n_2$, not only are $n_1$ and $n_2$ treated as being updated, but so are the nodes that point into $n_1$ and $n_2$.

One important property of this analysis is that the number of basic blocks flagged as visited and successfully executed increases monotonically as the algorithm proceeds; thus, the efficient/incremental algorithm may be freely used without regard for order of execution. This is insured (partially) by placing on the second worklist only those uses that can be reached by some simulated path through the CFG. A more subtle property is also required to insure convergence of the efficient algorithm: weak updates without intervening uses may be performed in any order because a weak update can only add edges, and conversion of a strong update into a weak update cannot delete any edges from any SSG.

The algorithm presented here consists of a driver, a subroutine to process basic blocks from the worklist, a

```
do
  WORK ← {ROOT}
  visited ← {ROOT}
  thru ← {}

  while WORK not empty do
    take item from WORK
    case item in
      BLOCK :  DoBlock(item)
      USE : DoUse(item)
    endcase
  end
end
```

Figure 12. Driver.

```
DoUse(use)
  (oldval, definer, stmt, type, key) ← use
  delta ← eval(definer, stmt) - oldval
  if delta not empty then
    case type in
      LHSSEL  :
        forall n in delta do
          forall f in
               addr(n.sel(lhs(stmt)),stmt) do
            AddDef(f, stmt,
                   eval(rhs(stmt), key))
          end
        end
        if previously there were no fields
           in the l-value of stmt and now
           there is at least one,
        then
           add stmt's block to WORK.
        endif
      RHSSEL  :
        forall n in delta do
          forall f in
               addr(n.sel(rhs(stmt)),stmt) do
            AddUse(f, stmt, RHS, stmt)
          end
        end
      RHS, PHI :
        forall f in addr(lhs(stmt), stmt) do
          forall u in defs(f)(stmt).uses do
            add u to WORK
          end
        end
    endcase
  endif
end DoUse
```

Figure 13. Uses.

subroutine to process uses of changed definitions, and subroutines to insert new uses, new definitions, and new $\phi$-functions. The insertion subroutines also schedule items onto the worklist as necessary; a new use propagates a value forward, and new definitions can affect the values seen at existing uses. AddPhi and AddDef are mutually recursive; a new definition must propagate to $\phi$-functions at its dominance frontier, but $\phi$-functions are definitions.

```
DoBlock(B)
  S ← nextstmt(B)
  stuck ← false
  while S not equal last(B) and not stuck do
    case rhs(S) in
      'x' : AddUse(x, S, RHS, S)
      'x.a' : AddUse(x, S, RHSSEL, S)
              forall y in addr(x.a,S)
              AddUse(y, S, RHS, S)
      'φ' : ignore it
    endcase
    case lhs(S) in
      'x' : AddDef(x, S, eval(rhs(S),S))
      'x.a' : AddUse(x, S, LHSSEL, S)
              if no edges leaving x
              then stuck ← true
              else
                forall y in addr(lhs(S),S)
                  AddDef(y, S, eval(rhs(S),S))
              endif
    endcase
    if not stuck then S ← successor(S) endif
  end
  if stuck
  then nextstmt(B) ← S
  else
    thru ← thru + B
    forall (N,C) in queue(B) do
      AddPhi(N, B, C)
    end
    forall C in succ(B) - visited do
      add B to WORK
      add B to visited
    end
  endif
end DoBlock
```

Figure 14. Blocks.

The algorithm uses the following terms:

df(x) Dominance frontier of x (a set of block-to-block edges).

defs(x) Definitions for x. Defs(x)(p) is the definition for x reaching point p. A definition is a triple (stmt, value, uses).

WORK A list of uses and blocks. Basic blocks on the list are eligible for execution; uses on the list have changed definitions. Execution of a basic block B begins at nextstmt(B). For simplicity, each basic block has empty statements first(B) and last(B) where $\phi$-functions may be attached.

use A "use" is a quadruple (value, stmt, type, key), where value is the most recently computed "value" (nodes addressed) seen at this use, stmt is the statement at which the use occurs, type is the type of use, and key is the statement with which the use is associated (and differs from stmt only for uses in $\phi$-functions).

ROOT The first basic block in the program.

```
AddUse(field, stmt, type, key)
  (dstmt, dvalue, duses) ←
    defs(field, stmt)
  defs(field, dstmt) ←
    (dstmt, dvalue, duses + (stmt, type, key))
  create a use u =
    (eval(field, key), stmt, type, key)
  add u to WORK.
end AddUse
```

Figure 15. AddUse.

```
AddPhi(field, defb, useb)
  if defb not in thru then
    add (field, useb) to queue(defb)
  else if $\phi$ for field in useb then
    AddUse(field, first(useb),
          PHI, last(defb))
  else
    newnodes ← {}
    forall B in pred(useb) intersect thru do
      AddUse(field, first(useb), PHI, last(B))
      newnodes ← eval(field, last(B))
      end
    AddDef(field, useb, newnodes)
  endif
end AddPhi
```

Figure 16. AddPhi.

visited The basic blocks that have been placed on the queue at least once.

thru The basic blocks that have been executed to completion.

lhs(s) Lhs expression of $s$.

rhs(s) Rhs expression of $s$.

pfx(e) Prefix of a selection expression.

sel(e) Selector of a selection expression. A useful abuse of notation will be "addr(x.sel(lhs(stmt)))", meaning "use the field selector occurring in the left-hand side of stmt to pick a field of the SSG node x".

addr(e,S) Fields/variables addressed by an expression.

deref(f,S) Nodes reached by traversing arrows leaving the field or variable f.

eval(e,S) {deref(f,S)|f $\in$ addr(e,S)}.

Note that the parameter S is necessary to indicate the statement at which the expression, field, or variable is being evaluated. As an abuse of typing, singleton sets and single elements are used interchangeably where convenient.

## 5.3 Time Bounds for the Efficient Algorithm

The size of the inverted data structure is dependent on the product of the number of nodes in the SSG ($O(S+V)$) and the size of the structures under them. Each assignment statement to a heap element (e.g. X.car ← ... ) can update

```
AddDef(field, stmt, value)
  (dstm, dval, duses) ← defs(field, stmt)
  uses ← {u in duses | stmt dominates u}
  defs(field, dstm) ←
           (dstm, dval, duses-uses)
  defs(field, stmt) ← (stmt, value, uses)
  forall u in uses do
    add u to WORK.
  end
  forall edges (b1,b2) in df(stmt) do
    if def of field at stmt reaches last(b1)
    then AddPhi(field, b1, b2)
    endif
  end
end AddDef
```

Figure 17. AddDef.

no more than $O(T)$ nodes. An assignment to a variable can cause a merge of two nodes, which causes an update of all $O(T+1)$ nodes pointing to it. Each actual update may cause $\phi$-functions to be added to the skeleton tree. These are at the dominance frontier of the update and recursively to the dominance frontiers of *those* $\phi$-*functions*. This is exactly like adding an assignment to a program in SSA form and seeing how many more pseudo-assignments are added. Results in [CFR$^+$89a, CFR$^+$89b] show that in practice the number of pseudo-assignments per assignment tends to be a small constant. However, since in the worst case it could be $O(S)$, we call the average number of potential updates added $P$. The number of SSG nodes in all of the skeleton trees will be $O(S \times T \times P)$.

Since the number of edges is proportional to the overall running time of the algorithm and inversely proportional to the quality of the information gathered, it may be perfectly reasonable to abort the algorithm if the total number of edges approaches becoming quadratic, since the information would be quite poor. This can help ensure that $T$ remains small.

The cost of adding an update or potential update has two parts: updating the immediate node and propagating the effects to all new uses of the node. The most expensive part of updating a node is the potential insertion in the skeleton tree. With the sophisticated insertion algorithm, this takes $O(\log(S))$, and thus the cost of update only for a node is $O(S\log(S) \times T \times P)$. Updating a variable can cause a merge and updates of nodes pointing at the merged node. The cost of updating the nodes pointing to the merged node is $O(S\log(S) \times T \times P)$; the cost of the merge itself depends on the number of fields $F$ in the node and is $O(S\log(S) \times F \times P)$.

The uses of a new edge can be found in time $O(T \times$ the number of uses of that node). For each use, if a variable node has just been updated, $O(T)$ steps are done on the LHS and $O(\max(T\log(S), T^2))$ on the right hand side to recompute the new set of nodes pointed to. If a heap node has just been updated, $O(\max(\log(s), T))$ steps are required. A node can be updated at most $O(T)$ times. The $O(S)$ uses of variables in the program contribute $O(\max(S \times T^3), S\log(S) \times T^2))$ work. There are potentially $O(S \times T)$ uses of nodes, and they contribute $O(\max(S \times T^3), S\log(S) \times T^2))$ work as well. Thus the time bound is $O(\max((S\log(S) \times T \times P), (S\log(S) \times F \times P), (S \times T^3), S\log(S) \times T^2))$.

## 6 Interprocedural Analysis

Our method for interprocedural analysis is straightforward: we model procedure calls with a branch to get to the procedure, a labeled branch to return and assignments to model parameter passing. In this section we examine the details and the consequences of analyzing a program as a single procedure.

A call becomes an assignment to a label, some assignments to model parameter passing and an assignment to a label variable of the location immediately following the call. A return is simply a case statement using the label and a jump. Thus, in each branch of the case statement in the flow graph, the case statement has edges leading to all locations following calls to the procedure, and we lose the information that a return is associated with a call. We have a different labeled variable for each procedure, and our graph reflects the fact that the return from procedure "`foo`" cannot return to the statement following a call to a different procedure "`bar`". (This is why we use a case statement rather than a labeled jump.) Modeling value or value returns via assignments is straightforward. The assignments for the return can be put in the case statement.

At first one might think reference parameters should be modeled with pointers: any variable that is passed by reference becomes a pointer to a node holding the actual value that the programmer associates with the variable. Unfortunately, this interferes with our heuristics for keeping multiple copies of a node (see Section 4.1), so we suggest a different technique based on the observation (in [WZ88]) that, at least for the purposes of analysis, call by reference can be modeled by call by value-result and some extra assignments. There are three cases:

1. If the call by reference parameter is never aliased with any other variable, then it can be replaced by a call by value-return parameter.

2. If the call by reference parameter is always aliased with a set of variables, then it can be replaced by a call by value-return parameter. Any assignment to one of those variables is replaced by assignments to all of them; wherever possible, these assignments may be done using strong updates.

3. If the call by reference parameter may sometimes be aliased with a set of variables, then for the purposes of analysis it can be replaced by a call by value-return parameter. An assignment to one variable can be replaced by a *may assignment* to the other variables. For our analysis here, an assignment to one variable can be modeled as a weak update to all the other variables. At runtime, this must still be implemented as call by reference.

The analysis to determine which parameters can be aliased can be performed by techniques given in [Bar78, Ban79, Bur87, CK88, Mye81, WZ88]:

A major drawback of this analysis is that a procedure, which may be called from several places, may allocate cons nodes. Those nodes all come from the same static allocation statement and hence we have a single summary node for all of them. We can hope that the calling procedures all use the nodes in the same way and update them in similar ways, so that the analysis of one program section does not create an SSG more general than is needed by a different section.

If a procedure allocates nodes and keeps those nodes pointed to by variables, the heuristics in Section 4.1 may keep those nodes separated from the summary node for that allocation site. This requires that the calling program also use variables to separate the nodes. However, there will still be only one summary node for all the nodes allocated in the called procedure.

Procedures may be integrated into the program. This is particularly sensible where they are small and where they allocate only a few cons cells. The entities our technique analyzes are the individual cons sites. Consider a leaf procedure called from many places whose main purpose is to allocate (and possibly initialize) a piece of storage that is returned. We lose information if we don't integrate the procedure. In the worst case, if the cons procedure itself is not integrated, then all distinction between different allocation sites will be lost.

Interprocedural analysis algorithms have tended to use the following framework [Bar78, Ban79, Mye81, Bur87, Cal88, CK89]:

1. A summary is built of each procedure that describes the effects of calling that procedure on the parameters and global variables. These summaries are generally *flow-insensitive*, i.e., they discard all of the information about paths within the procedure.

2. A *call graph* is constructed. A call graph is a directed multigraph in which the nodes represent the procedures and there is an edge from procedure `X` to procedure `Y` for each call from `X` to `Y`.

3. A data flow analysis problem is solved over the call graph using the summary information as the initial conditions.

4. The interprocedural information computed in the previous step is used to optimize each procedure.

This framework has been justified through two assumptions: (1) little information is lost during by summarizing, and (2) even if the loss is substantial, you could not afford to do otherwise. While historically correct, we feel that these assumptions need to be reassessed.

- Our analysis associates the information with the individual cons sites on the assumption that memory from different cons sites is used in different ways. Flow-insensitive summaries destroy such knowledge.

- Most modern machines have substantial amounts of real memory and extremely large virtual address spaces. On such machines, performing summarization to save space is unnecessary.

- Algorithms based on propagating bit-vectors have quadratic time bounds at best (the width of the bit vector times the number of nodes in the call graph). With nonlinear algorithms, care must be taken to limit growth in the significant parameters. We expect our algorithm to be much more efficient than previous algorithms; thus the number of nodes can grow in a reasonable way without the performance becoming unreasonable.

- In order to take advantage of the expected sparseness in the information, our algorithm uses explicit lists of edges rather than bit-vectors. The speed of our algorithm (and the quality of the information gathered) is inversely proportional to total size of these lists: the smaller the list, the faster the algorithm runs (and the better the information discovered). This has the following counterintuitive property: an algorithm using the non-summarized version (which may be considerably larger than a summarized version) may actually perform better because the higher-quality fixed point is closer to the initial conditions.

While we generally expect our algorithm to be linear, one case raises some concern: if one kind of data structure that has been allocated at many points in the program is manipulated by a single function, that function may be expensive to analyze. This is because the variables within that function may point to a large number of nodes, each of which has been allocated at a different allocate statement.

## 7 Comparison With Other Work

Extant techniques view the program as a generator of data structures. Each cons statement in the program adds nodes to the data structure. When the size of the data structure exceeds some bound, these nodes are combined into summary nodes. The limit on graph size is often based on a bound, $k$, on the maximum acyclic path length [Sch75a, Sch75b, JM81, Rug87, RM88, LH88, Lar89, HPR89], hence these are called $k$-bounded techniques. If the data structures being analyzed are recursive, the size of the data structure will be unbounded. Therefore, graphs for programs with recursive data structures must contain summarized components.

As an example, consider the first loop in the program in Figure 2 with the path length bounded to three cons nodes. At the end of the second iteration of the loop, a summary is required. The graph before summary is shown in Figure 18. Summarization must discover a shorter representation for

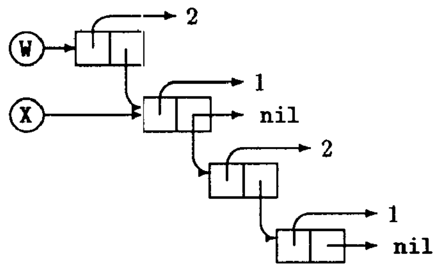

Figure 18. Before summary.

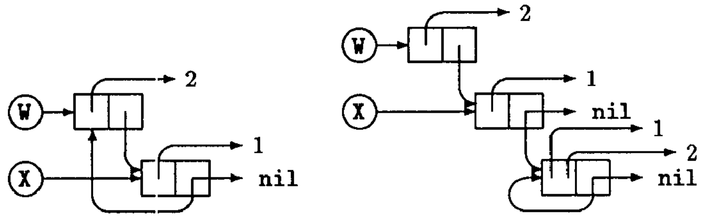

Figure 19. Two possible truncations.

this list that still contains all paths in this list. Figure 19 shows two possible truncations of this list. Existing truncation processes shorten the list as little as possible, and hence end up with the right-hand alternative. However, the left-hand alternative (which is what we obtain) contains more useful information because it retains the periodic structure of the list.

The $k$-bounded techniques have several disadvantages:

- Choosing to limit path length or node label size to $k$ makes possible a blowup exponential in $k$; a complete binary tree of depth $k$ contains $2^k - 1$ nodes. In practice $k$ is chosen very small (2 or 3) so blowup is not a problem, but this choice greatly reduces the amount of information that can be retained.

- Choosing to summarize together all nodes deeper than $k$ in a data structure ignores any hints provided by program structure. When summarizing, it is useful to combine nodes with similar attributes so that as little information as possible is lost in their meet. Two nodes allocated at the same statement are more likely (we believe) to be similar than two nodes occurring deep within a data structure, and the analysis should take advantage of this.

- $k$-bounded truncation makes it difficult to maintain information about the elements of a list. Truncation is typically done at the deepest node in the structure: all nodes deeper than $k$ are merged into their immediate ancestor. When this ancestor is a node in the spine of the list, the structure and aliasing of the elements of the list are summarized with the structure and aliasing of the list spine. The result provides little information about list or element structure.

  For example, consider truncation of the graph shown in Figure 20, where the deepest cons node is `X.cdr.cdr.car`. A naive truncation of that tree, shown in Figure 21, summarizes the node with its parent, introducing a cycle to describe a structure that is not only acyclic but also of small bounded size. Any differences in the attributes of the list spine and the list elements are also lost. Figure 22 shows the final summary after continued growth of the list.

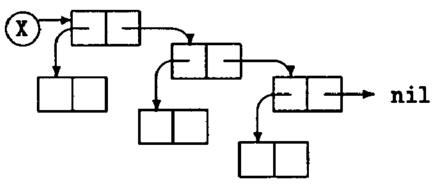

Figure 20. List before summary.

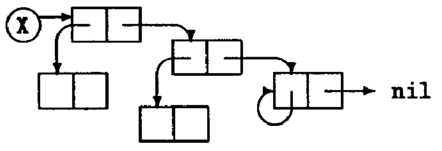

Figure 21. A bad summary.

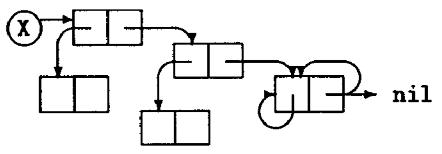

Figure 22. Summary after additional growth.

- The process of truncation is fairly complex (and differs greatly among those papers that use the technique).

The method described in this paper has none of these drawbacks. Nodes combined in the same summary are allocated at the same site (thus exploiting program structure); the elements of a list are (typically) summarized separately from the spine of the list; the truncation process is simple.

Jones and Muchnick [JM82] were the first to describe a method for analysis of unbounded data structures based on summarizing multiple nodes in a data structure into a single node based on a fixed node set. However, their model does not include updates. In one example, they, like us, choose summary nodes based on points in the program. They attempt to get information that is more precise than ours, but at a substantial time cost.

Work by Chase [Cha87] uses similar techniques for choosing nodes for summaries, but does not provide an efficient algorithm ($O(n^4)$ for programs with side effects, where $n$ is the number of statements in the program), and does not address strong updates or the association of additional attributes with graph nodes. Ruggieri [Rug87] provides a complexity result of $O(|V|^3 \times |S| \times n)$, where $S$ is the number of allocation sites, $n$ is the number of statements in a program, and $V$ is the number of variables and "subvariables" (but in the worst case $|V|$ is exponential in a parameter $k$).

$k$-bounded methods described by Larus [LH88, Lar89] can obtain strong updates in some situations. His conditions for strong update are similar to ours; he does not discuss the constraints placed by strong update on the order in which statements are processed. Other work appears not to discuss strong updates.

For languages with side effects, only Hendren's technique [HN89] automatically discovers that recursive data structures are unshared (i.e., lists and trees). This information is very important; given declarations identifying true lists and trees, Larus has performed additional analysis and useful optimizations on Lisp functions manipulating linked data structures [Lar89]. Hendren's work discovers trees and DAGs for use in dependence analysis. It is not clear whether her work can be easily extended to provide lifetime analysis or to allow value approximation within heap cells, or if it can obtain the information provided by extended models of references counting.

Deutsch [Deu90] describes a similar analysis for strict higher-order functional languages with lexical scoping, polymorphism, and first-class continuations. He does not treat side-effects (there are none) and doesn't address efficiency of analysis.

Stransky also proposes a similar analysis in his thesis [Str88], but we are unable to compare our work with his because our French is not adequate. His work appears not to address strong updates, tree detection, or efficient algorithms.

## 8 Future Work

While there are many aspects of the analysis problem that we have addressed, we realize that there are still many unresolved issues:

**Implementation:** We have made some assumptions about the sparseness of certain structures. Based on these assumptions, we have developed techniques and data structures that we feel may perform well in practice. Measurements of real programs are necessary to prove these assumptions.

**Enhancements to the information:** Programs often take a pre-existing data structure and modify it. For example, we do not get very good information when analyzing a program that reverses a list in place. There are at least two problems. (1) We have restricted ourselves to using only one summary node. As the list is reversed, we need to have one summary node for the elements that we have finished processing and one for those we have yet to start processing. These two nodes cannot be merged and retain the information we want. (2) We need to make strong updates to the node whose fields we are reversing in this iteration of the loop. Doing so requires a single-instance node. We need a mechanism for unmerging a summary node into a summary and a single-instance node.

While we can fit multiple summaries and unmerging into our framework, we must develop the mechanisms that trigger their use in a wide variety of programs. We have developed mechanisms which work on isolated examples, but a good general mechanism has so far eluded us.

**Telling the programmer:** The information that we automatically collect may help programmers understand what their program does. Currently data structure editors help give an idea of what a dynamically produced data structure looks like. It is possible that an editor could be built on top of the technology discussed in this paper to look at all possible data structures buildable by a program.

## 9 Conclusion

The analysis presented above obtains useful information about linked data structures. We summarize unbounded data structures by taking advantage of structure present in the original program. The worst-case time bounds for a naive algorithm are high-degree polynomial, but for the expected (sparse) case we have an efficient algorithm. Previous work has addressed time bounds rarely, and efficient algorithms not at all.

The quality of information obtained by this analysis appears to be (generally) better than what is obtained by existing techniques. A simple extension obtains aliasing information for entire data structures that previously could be obtained only through declarations. Previous work by Larus has shown that this information allows worthwhile optimization.

We believe that practical analyses based on this work can be used in compilers for languages that provide linked data structures.

## References

[Ban79] J. B. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Conf. Rec. Sixth ACM Symp. on Principles of Programming Languages*, pages 29–41, January 1979.

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

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Comm. ACM*, 21(9):724–736, September 1978.

[BH88] H.-J. Boehm and L. Hederman. Storage allocation optimization in a compiler for Russell. Submitted for publication, July 1988.

[Bur87] M. Burke. An interval-based approach to exhaustive and incremental data flow analysis. Technical Report 12702, IBM, Yorktown Hts., New York, September 1987.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, 23(7):47–56, July 1988.

[CFR+89a] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Languages*, pages 25–35, January 1989.

[CFR+89b] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical Report RC 14756, IBM, July 1989.

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

[Cha87] D. R. Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Dept. of Computer Sci., Rice U., August 1987.

[CK88] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 57–66, June 1988. Published as SIGPLAN Notices Vol. 23, Num. 7.

[CK89] K. D. Cooper and K. W. Kennedy. Fast interprocedural alias analysis. *Conf. Rec. Sixteenth ACM Symp. on Principles of Programming Languages*, pages 49–59, January 1989. Austin, Texas.

[CWZ90] D. R. Chase, W. Wegman, and F. K. Zadeck. Fast insertion, deletion, and lookup in sparse ansestor trees. Technical Report CS-90-07, Dept. of Computer Sci., Brown U., March 1990.

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

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. *Conf. Rec. Seventeenth ACM Symp. on Principles of Programming Languages*, pages 157–168, January 1990.

[Hed88] L. Hederman. Compile time garbage collection. Master's thesis, Dept. of Computer Sci., Rice U., 1988.

[HN89] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *Proc. 1989 International Conf. on Parallel Processing*, II:49–56, 1989.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *Proc. SIGPLAN'89 Symp. on Compiler Construction*, June 1989. Published as SIGPLAN Notices Vol. 24, Num. 7.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 351–363, 1986.

[JM81] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[JM82] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Conf. Rec. Ninth ACM Symp. on Principles of Programming Languages*, January 1982.

[Lar89] J. R. Larus. Restructuring symbolic programs for concurrent execution on multiprocessors. Technical Report UCB/CSD 89/502, Computer Sci. Dept., U. of California at Berkeley, Berkeley, CA, May 1989.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 21–34, July 1988. Published as SIGPLAN Notices Vol. 23, Num. 7.

[Mye81] E. W. Myers. A precise interprocedural data flow algorithm. *Conf. Rec. Eighth ACM Symp. on Principles of Programming Languages*, pages 219–230, January 1981.

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 285–293, January 1988.

[Rov85] P. 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, Palo Alto, Ca. 94304, 1985.

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

[Sch75a] J. T. Schwartz. Optimization of very high level languages—I: Value transmission and its corollaries. *Computer Languages*, 1:161–194, 1975.

[Sch75b] J. T. Schwartz. Optimization of very high level languages—II: Deducing relationships of inclusion and membership. *Computer Languages*, 1:197–218, 1975.

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

[Str88] J. Stransky. *Analyse sémantique de structures de données dynamiques avec application au cas particulier de langages LISPiens*. PhD thesis, Université de Paris-Sud, Centre d'Orsay, June 1988.

[Weg75] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. on Software Engineering*, SE-1(3):270–285, September 1975.

[WZ88] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. Technical Report CS-88-02, Dept. of Computer Sci., Brown U., February 1988.