Solving Shape-Analysis Problems in Languages with Destructive Updating

Mooly Sagiv*† and Thomas Reps†
University of Wisconsin

Reinhard Wilhelm‡
Universität des Saarlandes

Abstract

This paper concerns the static analysis of programs that perform destructive updating on heap-allocated storage. We give an algorithm that conservatively solves this problem by using a finite shape-graph to approximate the possible "shapes" that heap-allocated structures in a program can take on. In contrast with previous work, *our method is even accurate for certain programs that update* **cyclic** *data structures*. For example, our method can determine that when the input to a program that searches a list and splices in a new element is a possibly circular list, the output is a possibly circular list.

1 Introduction

This paper concerns the static analysis of programs that perform destructive updating on heap-allocated storage. It addresses problems that can be looked at — depending on one's point of view — as *pointer-analysis* problems, *alias-analysis* problems, *storage-analysis* (*shape-analysis*) problems, or *type-checking* problems. The information obtained is useful, for instance, for generating efficient sequential or parallel code.

Throughout most of the paper, we will emphasize the application of our approach to shape-analysis problems. The goal of shape analysis is to give, for each program point, a (finite) characterization of the possible "shapes" that the program's heap-allocated data structures can have at that point. We will illustrate our approach by means of a running example in which we apply our analysis technique to a program that uses destructive updating operations to reverse a list. This example also illustrates the connection between shape analysis and type checking: It demonstrates how a sufficiently precise shape-analysis algorithm is able to verify that the destructive-reverse program does indeed return a list whenever its argument is a list. The application of our work to pointer-analysis and alias-analysis problems is discussed in Section 5.2.

This paper develops a new shape-analysis algorithm that provides conservative information about the possible "shapes" that heap-allocated structures in a program can take on. For certain programs — including ones in which a significant amount of destructive updating takes place — our algorithm is able to verify shape-preservation properties. Examples of such properties include: (i) when the input to the program is a list, the output is (still) a list; (ii) when the input to the program is a tree, the output is (still) a tree; and (iii) when the input to the program is a circular list, the output is a circular list. For instance, our method can determine that "list-ness" is preserved by (i) a list-reversal program that performs the reversal by destructively updating the input list, and (ii) a list-insert program that searches a list and splices a new element into the list. Furthermore, our method can determine that the list-insert program also preserves "circular list-ness".

These are rather surprising capabilities. None of the previously developed methods that use graphs to solve shape-analysis problems are capable of determining that "list-ness" is preserved on these examples (or examples of similar complexity) [JM81, JM82, LH88, CWZ90, Str92, PCK93]. Previous to this paper, it was an open question whether such precision could ever be obtained by any method that uses graphs to model storage usage. Furthermore, as far as we know, no other shape-analysis/type-checking method (whether based on graphs or other principles [HN90, Hen90, LR91, Deu92, CBC93, Deu94]) has the ability to determine that "circular list-ness" is preserved by the list-insert program.

What does our method do that allows it to obtain such qualitatively better results on the above-mentioned programs than previous methods? A detailed examination of the differences between our algorithm and previous algorithms is deferred to Section 6; however, a brief characterization of some of the differences is as follows:

- Previous methods have used allocation sites to name shape-nodes [JM82, CWZ90, PCK93]. Allocation-site information imposes a **fixed partition** on the memory. In contrast, our approach deliberately drops information about the *concrete locations*. There is only an indirect connection to the run-time locations: Shape-graph nodes are named using a (possibly empty) *set of variables*. The variable set of a shape-graph node in the shape-graph for program-point v consists of variables that, for some execution sequence ending at v, must all point to the same run-time location.
- Like other shape-analysis methods, our method clusters collections of run-time locations into *summary nodes*. In our approach, nodes that are not pointed to by variables are clustered into a single node. Chase, Wegman, and Zadeck observed that their analysis method cannot handle programs such as the list-reversal program be-

To appear in the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, on January 21-24, 1996.

[*] On leave from the IBM Israel Scientific Center. Part of this research was done while visiting the Universität des Saarlandes, partially supported by SFB 124-VLSI-Design Methods and Parallelism of the Deutsche Forschungsgemeinschaft.

[†] Supported by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937). Address: Computer Science Department; 1210 West Dayton Street; Madison, WI 53706; USA. Email: {sagiv,reps}@cs.wisc.edu.

[‡] Address: Fachbereich 14 Informatik; 66123 Saarbrücken; Germany. Email: wilhelm@cs.uni-sb.de.

cause it lacks a way to materialize ("un-summarize") summary nodes at certain key points of the analysis [CWZ90, pp. 309]. Our shape-node naming scheme allows our method to materialize copies of the summary node (as non-summary nodes) whenever a pointer variable is assigned a previously summarized run-time location.

- In the analysis of an assignment to a component, say $x.cdr$:= **nil**, our method always removes x's *cdr* edges. Previous methods either never remove these edges [Str92] or have some heuristics to remove such edges under certain conditions [JM81, LH88, CWZ90, PCK93]. (This unusual characteristic of our method is enabled by both the node-naming scheme and the materialization technique.)
- We use *sharing* information to increase the accuracy of the primitive operations used by our method. More specifically, we keep track of shape-nodes that may be the target of more than 1 pointer from the heap. For example, when a linked data structure is traversed, say via a loop containing an assignment $x := x.cdr$, the sharing information is used to improve the precision of the materialization operation, which allows our algorithm to determine that x points to a list element on every iteration. The limited form of sharing information used in [JM81, CWZ90] does not allow these methods to determine this fact.
- The shape-node names also provide information that sometimes permits our method to determine that a shared-node becomes unshared (e.g., this occurs in the program that performs an insertion into a list). With the Chase-Wegman-Zadeck method, once a node is shared it remains shared forever thereafter. For programs that operate on lists and trees, the non-graph-based method of Hendren [Hen90] is sometimes able to determine that a shared-node becomes unshared. However, this method does not handle data structures that contain cycles.

An experimental implementation of the analysis method has been created; the examples presented in the paper have been prepared with the aid of this implementation.

The remainder of the paper is organized as follows: Section 2 introduces the terminology and notation used in the rest of the paper. Section 3 presents a concrete semantics for a language with destructive updating, in terms of "shape-graphs" in which nodes represent run-time locations. Section 4 introduces an abstract domain of "static shape-graphs" and shows how they can be used to approximate the sets of shape-graphs that arise in the collecting semantics. Section 5 summarizes a few extensions to our basic approach. Section 6 discusses related work. Due to space constraints, we have omitted discussions of (i) other elaborations and extensions of our basic approach, (ii) a proof that our abstract semantics of static shape-graphs is safe with respect to the concrete semantics. This material can be found in [SRW95].

2 Terminology and Notation

2.1 The Language

We assume we are working with an imperative language that meets the following general description: A program consists of assignment statements, conditional statements, loops (while, for, repeat), read statements, write statements, and goto statements.[1] The language provides atomic data (e.g., integer, real, boolean, etc.) and constructor and selector operations (e.g., **nil**, **cons**, **car**, and **cdr**[2]), together with appropriate predicates (equal, atom, and null). We assume that a read statement reads just an atom and not an entire list, tree, or graph.

A program is represented by a control-flow graph $G = (V, A)$, where V is the set of vertices and $A \subseteq V \times V$ is the set of arcs. G has a unique start vertex, which we assume has no predecessors. The other vertices of the control-flow graph represent the statements and predicates of the program in the usual way; $st(v)$ denotes the statement or predicate of vertex v.

Normalization Assumptions. For expository convenience, we will assume that programs have been normalized to meet the following conditions:

- Only one constructor or selector is performed per statement.
- An expression $\mathbf{cons}(x, y)$ is executed in three steps: (i) allocate the **cons** cell and assign it to a new temporary variable; (ii) assign x into the *car* component; (iii) assign y into the *cdr* component.
- In each statement, the same variable does not occur on both the left-hand and right-hand side.
- Each statement of the form $l := r$ where $r \not\equiv$ **nil** is preceded by an assignment of the form $l :=$ **nil**.
- All allocation statements are of the form $x :=$ **new** (as opposed to $x.sel :=$ **new**).

Thus, for every vertex $v \in V$ in which a pointer manipulation is performed, $st(v)$ has one of the following forms: $x :=$ **nil**, $x.sel :=$ **nil**, $x :=$ **new**, $x := y$, $x := y.sel$, or $x.sel := y$, where $y \not\equiv x$. (In our implementation, the work of putting a program into a form that meets these assumptions is carried out by a preprocessor.) Note that the number of temporary variables that are introduced to meet these restrictions is, in the worst case, linear in the size of the original program. □

The normalization assumptions are not essential, but simplify the presentation. For example, the last assumption allows us to separate the "kill" aspects of a statement (e.g., $x :=$ **nil**) from the "gen" aspects (e.g., $x := y.sel_0$, assuming x points to **nil**) in the semantics. (See Figures 2 and 6.)

Example 2.1 Figure 1 shows (a) a program that performs a list reversal via destructive updating, (b) the program in normalized form, and (c) the control-flow graph of the program in normalized form. The list initially pointed to by variable x is transformed into its reversal. After each iteration, y points to the reversal of a successively longer prefix of the original list. □

To simplify the formulation of the analysis method, it will be stated for a single fixed (but arbitrary) program. The set of pointer variables in this program will be denoted by $PVar$.

2.2 Shape-Graphs

Both the concrete and abstract semantics are defined in terms of a single unified concept of "shape graph", which

[1] The treatment of procedures is discussed later, in Section 5.3.

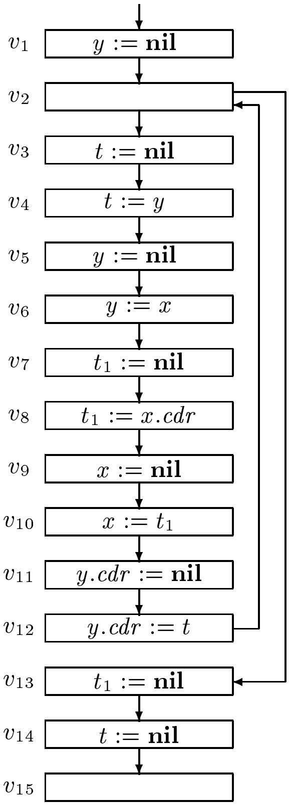
[2] Throughout the paper, our presentation is couched in terms of the Lisp primitives for manipulating heap-allocated storage. However, this is not due to any basic limitation of our method; our shape-analysis algorithm extends readily to the case of pointers to user-defined types that have more than two fields.

```
/* $x$ points to an unshared list */
$y$ := nil
while $x \neq$ nil do
    $t := y$
    $y := x$
    $x := x.cdr$
    $y.cdr := t$
od
$t$ := nil
```

(a)

```
/* $x$ points to an unshared list */
$y$ := nil
while $x \neq$ nil do
    $t$ := nil
    $t := y$
    $y$ := nil
    $y := x$
    $t_1$ := nil
    $t_1 := x.cdr$
    $x$ := nil
    $x := t_1$
    $y.cdr$ := nil
    $y.cdr := t$
od
$t_1$ := nil
$t$ := nil
```

(b)



(c)

Figure 1: A program, the program in normalized form, and the program's labeled control-flow graph.

is defined as follows:

Definition 2.2 *A* **shape-graph** *is a finite directed graph that consists of nodes, called* **shape-nodes***, and two kinds of edges:* **variable-edges** *and* **selector-edges***. A shape graph is represented by a pair of edge sets, $\langle E_v, E_s \rangle$, where*

- *E_v is the graph's set of variable-edges, each of which is of the form $[x, n]$, where $x \in PVar$ and n is a shape-node.*
- *E_s is the graph's set of selector-edges, each of which is of the form $\langle s, sel, t \rangle$, where s and t are shape-nodes, and $sel \in \{car, cdr\}$.*

We overload the symbol E_v to also mean the function that returns a variable's E_v successors. That is, for $x \in PVar$, we define $E_v(x)$ to be $E_v(x) \stackrel{\text{def}}{=} \{n \mid [x, n] \in E_v\}$. Similarly, for a shape-node s and $sel \in \{car, cdr\}$, we define $E_s(s, sel)$ to be $E_s(s, sel) \stackrel{\text{def}}{=} \{t \mid \langle s, sel, t \rangle \in E_s\}$. (The intended meaning of a use of E_v or E_s will always be clear, according to whether arguments are supplied or not.) Given $SG = \langle E_v, E_s \rangle$, we define $shape\_nodes(SG)$ as follows: $shape\_nodes(SG) \stackrel{\text{def}}{=} \{n \mid [*, n] \in E_v\} \cup \{n \mid \langle *, *, n \rangle \in E_s\} \cup \{n \mid \langle n, *, * \rangle \in E_s\}$. The class of shape-graphs is denoted by $\mathcal{SG}$. □

Note that for a given shape-graph SG, $shape\_nodes(SG)$ is uniquely defined: it consists of the set of non-isolated nodes in SG (i.e., the nodes that are touched by at least one edge). It is for this reason that we do not explicitly list the node set when specifying a shape-graph.

The shape-graphs that arise in the concrete semantics for the language have somewhat different characteristics from the ones that arise in the abstract semantics. However, the fact that both are defined from a shared root concept (namely Definition 2.2) helps in defining the abstraction relation that relates them (see Definitions 4.4 and 4.5).

In the concrete semantics, which is given in Section 3, the result of an execution sequence is a shape-graph that represents the state of heap-allocated storage in memory. In this case, each shape-node represents a unique run-time location, and for each variable x, either $E_v(x)$ is a singleton set (say $\{n\}$) or it is empty. Furthermore, $E_s(n, car)$ and $E_s(n, cdr)$, which represent the run-time locations pointed to by the car and cdr fields of n, are also either singleton sets or empty (depending on whether these fields point to allocated locations or not). Such properties are captured in the following definition:

Definition 2.3 *A shape-graph is* **deterministic** *if (i) for every $x \in PVar$, $|E_v(x)| \leq 1$ and (ii) for every shape-node n and $sel \in \{car, cdr\}$, $|E_s(n, sel)| \leq 1$. The class of deterministic shape-graphs is denoted by $\mathcal{DSG}$.* □

Finally, in several places we make use of a "garbage-collection" operation to eliminate shape-nodes that are not reachable from any of the program variables.

Definition 2.4 *The function $gc\colon \mathcal{SG} \to \mathcal{SG}$ is defined by $gc(\langle E_v, E_s \rangle) \stackrel{\text{def}}{=} \langle E_v, E'_s \rangle$, where $\langle s, sel, t \rangle \in E'_s$ iff $\langle s, sel, t \rangle \in E_s$ and there exists $[x, r] \in E_v$ such that there is a path of selector-edges in E_s from r to s.* □

3 The Concrete Semantics

In this section, we present a concrete semantics in which deterministic shape-graphs are used to represent the memory state (i.e., DSG shape-nodes represent "cons-cells"), and the meaning of an assignment statement is a deterministic shape-graph transformer. This concrete semantics is used to define a concrete collecting semantics that associates a set of possible shape-graphs with each point in the program.

Figure 2 contains the semantic equations of the concrete semantics. The meaning of a statement st is a function $[\![st]\!]_{\mathcal{DSG}}\colon \mathcal{DSG} \to \mathcal{DSG}$. (When examining the last four equations in Figure 2, bear in mind that, because of the Normalization Assumptions of Section 2.1, before each of the statements executes it is known that the left-hand side evaluates to **nil**. Thus, the last four equations need only handle the "gen" aspects of the statements' semantics. The "kill" aspects are handled by the first two equations of Figure 2.) The DSG transformers listed in Figure 2 cover the six kinds of pointer-manipulation statements; all the other DSG transformers — for predicates and for assignment statements that do not perform any pointer manipulations — are the identity function.

By design, the "concrete" semantics is somewhat nonstandard. The only part of the store that the concrete semantics keeps track of is the heap-allocated storage; furthermore, it does not interpret predicates, read statements, and assignment statements that do not perform pointer manipulations. These assumptions build a small amount of abstraction into the "concrete" semantics. The consequence of

$$
\begin{aligned}
&\llbracket x := \mathbf{nil} \rrbracket_{\mathcal{DSG}}(\langle E_v, E_s \rangle) \stackrel{\text{def}}{=} \langle E_v - \{[x, *]\}, E_s \rangle \\
&\llbracket x.sel_0 := \mathbf{nil} \rrbracket_{\mathcal{DSG}}(\langle E_v, E_s \rangle) \\
&\qquad \stackrel{\text{def}}{=} \langle E_v, E_s - \{\langle s, sel_0, * \rangle \mid [x, s] \in E_v\} \rangle \\
&\llbracket x := \mathbf{new} \rrbracket_{\mathcal{DSG}}(\langle E_v, E_s \rangle) \stackrel{\text{def}}{=} \langle E_v \cup \{[x, n_{new}]\}, E_s \rangle \\
&\llbracket x := y \rrbracket_{\mathcal{DSG}}(\langle E_v, E_s \rangle) \stackrel{\text{def}}{=} \langle E_v \cup \{[x, n] \mid [y, n] \in E_v\}, E_s \rangle \\
&\llbracket x := y.sel_0 \rrbracket_{\mathcal{DSG}}(\langle E_v, E_s \rangle) \\
&\qquad \stackrel{\text{def}}{=} \langle E_v \cup \{[x, t] \mid [y, s] \in E_v, \langle s, sel_0, t \rangle \in E_s\}, E_s \rangle \\
&\llbracket x.sel_0 := y \rrbracket_{\mathcal{DSG}}(\langle E_v, E_s \rangle) \\
&\qquad \stackrel{\text{def}}{=} \langle E_v, E_s \cup \{\langle s, sel_0, t \rangle \mid [x, s], [y, t] \in E_v\} \rangle
\end{aligned}
$$

Figure 2: The concrete semantics $\llbracket st \rrbracket_{\mathcal{DSG}} : \mathcal{DSG} \rightarrow \mathcal{DSG}$. The shape-graph transformer associated with all predicates and all assignment statements that do not perform any pointer manipulations is the identity function. The term n_{new} denotes an operation that generates a new shape-node (i.e., a new run-time location).

these assumptions is that the collecting semantics may associate a control-flow-graph vertex with more concrete shape-graphs (i.e., $\mathcal{DSG}$s) than would be the case were we to start with a conventional concrete semantics. (Our assumptions are patently safe, and so we will not take the space here to justify them further.)

For simplicity, we do not introduce any "garbage-collection operations" in the concrete semantics to eliminate nodes from $\mathcal{DSG}$s that are not reachable from any of the program variables. Also, dereferences of nil pointers are ignored. They are handled in [SRW95].

We now turn to the collecting semantics. For a control-flow-graph vertex $v \in V$, let *pathsTo*(v) be the set of paths in the control-flow graph from *start* to predecessors of v.

Definition 3.1 *The* **collecting semantics** $cs : V \rightarrow 2^{\mathcal{DSG}}$ *is defined as follows:*

$$
\begin{aligned}
cs(v) \stackrel{\text{def}}{=} \{\, & \llbracket st(v_k) \rrbracket_{\mathcal{DSG}}(\cdots(\llbracket st(v_1) \rrbracket_{\mathcal{DSG}}(\langle \phi, \phi \rangle))) \mid \\
& [v_1, \ldots, v_k] \in \mathit{pathsTo}(v) \,\}
\end{aligned}
$$

□

4 The Abstract Semantics

In this section, we present a shape-analysis technique that uses a restricted subset of shape-graphs, called *static shape-graphs*, to summarize the possible shapes that heap-allocated storage can take on.

Example 4.1 Suppose x points to a five-element list at the beginning of the list-reversal program. Column two of Figure 3 shows the DSGs that arise at vertex v_2. The abstract (summarized) representations for the six DSGs are shown in column three. The abstract value that is computed by the abstract semantics is the graph shown in the iteration-4 row of column four. (In this example, this graph is exactly the union of the six graphs shown in column three.) □

Static shape-graphs are defined in Section 4.1, the abstraction function is defined in Section 4.2, and the abstract semantics is given in Section 4.3. The reverse program is used as a running example. Section 4.4 explains the reasons for the accuracy of the analysis method, and shows that the method is capable of handling the insertion of an element at an arbitrary point in a linked list.

4.1 Static Shape-Graphs

Unlike the concrete shape-graphs of the collecting semantics, the static shape-graphs of the abstract semantics are non-deterministic: $E_v(x)$, $E_s(n, car)$, and $E_s(n, cdr)$ may each yield a set with more than one shape-node. In addition, static shape-graphs are of bounded size. This is achieved by our naming scheme for shape-nodes: the name of a shape-node is a (possibly empty) set of program variables; in general, the abstraction function clusters multiple concrete shape-nodes into a single static shape-node.

Definition 4.2 *A* **static shape-graph** *is a pair* $\langle SG, \mathit{is\_shared} \rangle$*, where*

- *SG is a shape-graph.*
- *The set* $\mathit{shape\_nodes}(SG)$ *is a subset of* $\{n_X \mid X \subseteq PVar\}$*.*
- *is_shared is a function of type* $\mathit{shape\_nodes}(SG) \rightarrow \{\mathit{false}, \mathit{true}\}$*.*

The class of static shape-graphs is denoted by $\mathcal{SSG}$*.* □

In the following definition, we impose an order on SSGs where $SG \sqsubseteq SG'$ if SG' contains more edges than SG.

Definition 4.3 *Let* $SG = \langle \langle E_v, E_s \rangle, \mathit{is\_shared} \rangle$ *and* $SG' = \langle \langle E'_v, E'_s \rangle, \mathit{is\_shared}' \rangle$*. We define the following ordering on* $\mathcal{SSG}$*:* $SG \sqsubseteq SG'$ *if and only if*

- $E_v \subseteq E'_v$*.*
- $E_s \subseteq E'_s$*.*
- *For every* $n \in \mathit{shape\_nodes}(SG)$*,*
  $\mathit{is\_shared}(n) \Rightarrow \mathit{is\_shared}'(n)$*.*

□

The domain $\mathcal{SSG}$ is a complete join semi-lattice with a join operator $\sqcup$ defined by:

$SG \sqcup SG' \stackrel{\text{def}}{=} \langle \langle E_v \cup E'_v, E_s \cup E'_s \rangle, \mathit{is\_shared} \vee \mathit{is\_shared}' \rangle$.

4.2 The Abstraction Function

Our task in this section is to define the abstraction function that relates the domains $2^{\mathcal{DSG}}$ and $\mathcal{SSG}$. However, before formally defining the abstraction function in Definition 4.5, we first illustrate some of the semantic properties of SSGs.

Column two of Figure 3 shows the DSGs that arise at vertex v_2 for all five iterations of the loop in the list-reversal program when input-list x is a five-element list. Column three shows their corresponding abstract values (i.e., SSGs). We note the following:

(i) In each SSG in column three, a shape-node n_Z, where $Z \neq \phi$, represents a *unique* run-time location in the corresponding DSG in column two — the location pointed to by every one of the variables in Z. However, across the collection of SSGs that are the abstractions of the (several different) DSGs that arise on different loop iterations, n_Z will, in general, denote *different* run-time locations. For example, shape-node $n_{\{x, t_1\}}$ represents the run-time locations l_2, l_3, l_4, and l_5 in the DSGs that arise on iterations 1, 2, 3, and 4, respectively.

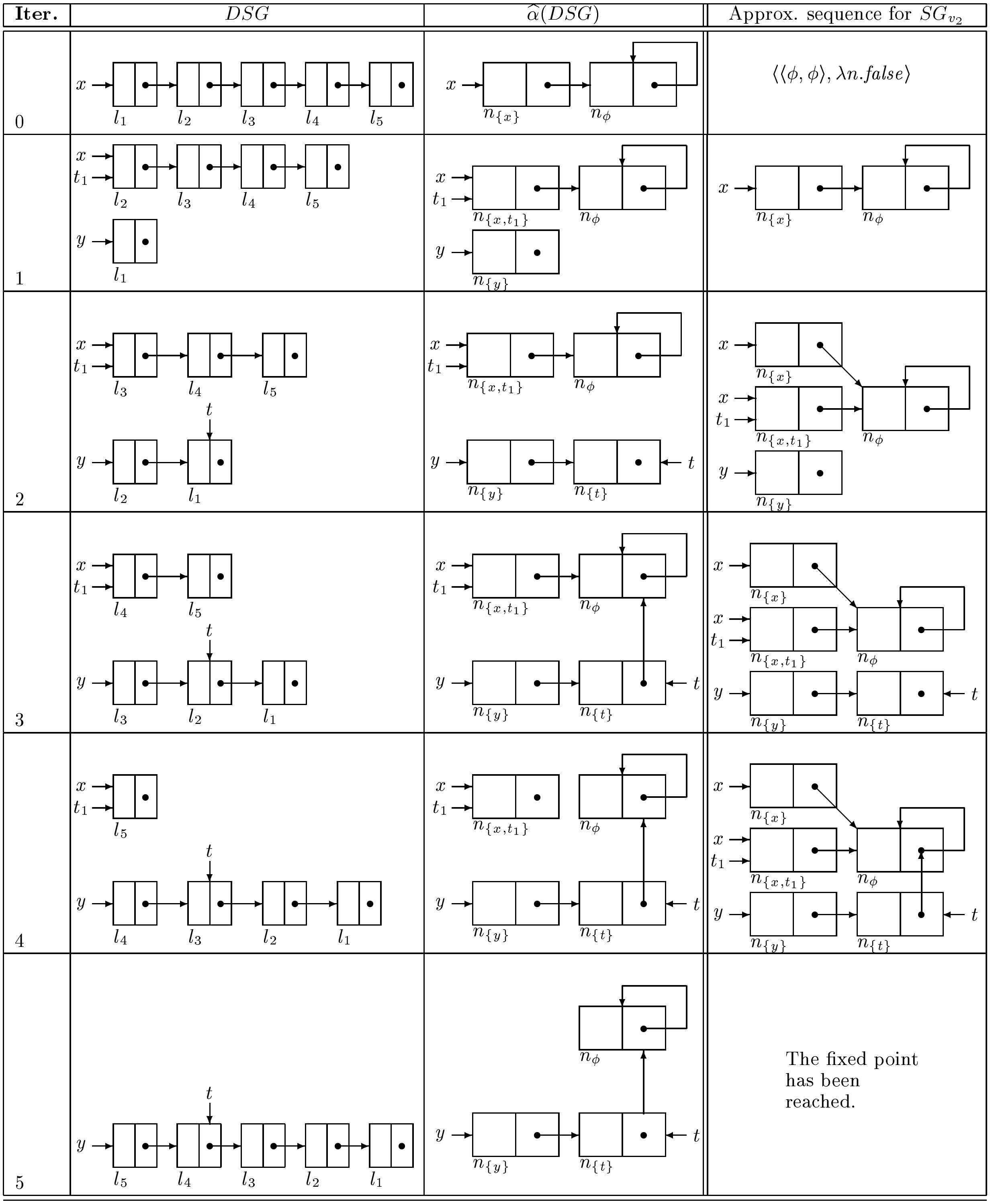


Figure 3: Columns two and three show the DSGs and the corresponding SSGs, as mapped by $\widehat{\alpha}$, that arise at vertex v_2 for the five iterations of the loop in the list-reversal program, when the input x is a five-element list. Column four shows the SSG values that SG_{v_2} takes on during the process of finding the solution to the equations of the abstract semantics. For each of the shape-nodes in all of the SSGs, the value of *is_shared* is false.

(ii) In contrast, shape-node n_ϕ can represent *multiple run-time locations of a* **single** *DSG*. For example, in the

SSG in column three of the iteration-0 row, n_ϕ represents the run-time locations l_2, l_3, l_4, and l_5 of the DSG in column two. In the SSG in column three of the iteration-5 row, n_ϕ represents the run-time locations l_3, l_2, and l_1.

(iii) In different SSGs, the same run-time location may be represented by *different* SSG shape-nodes. For instance, consider the SSGs in column three of Figure 3 in top-to-bottom order. Location l_1 is represented by shape-nodes $n_{\{x\}}$, $n_{\{y\}}$, $n_{\{t\}}$, n_ϕ, n_ϕ, and n_ϕ; location l_3 is represented by n_ϕ, n_ϕ, $n_{\{x,t_1\}}$, $n_{\{y\}}$, $n_{\{t\}}$, and n_ϕ; location l_5 is represented by n_ϕ, n_ϕ, n_ϕ, n_ϕ, $n_{\{x,t_1\}}$, and $n_{\{y\}}$.

There is an important conclusion to draw from these examples: It is *incorrect* to think of a shape-node as representing a fixed partition of memory. Instead, the ideas to keep in mind are the following:

> The variable set of a shape-node in the shape-graph for program-point v consists of variables that, *for some execution sequence ending at* v, must all point to the same run-time location. By going from DSGs to SSGs, we deliberately drop information about the concrete locations, but we keep information that indicates, *for some execution sequence ending at* v, what variables must all point to the same location.

A consequence of this abstraction is that two different shape-nodes n_X and n_Y such that $X \cap Y \neq \phi$ represent *incompatible* configurations of variables; that is, n_X and n_Y cannot possibly represent information from the *same* DSG. This means that the following structural invariants hold for the SSGs that arise in the abstraction process:

Invariant (i) ("Equality-or-disjointness of edge end-points") For all $\langle n_X, sel, n_Y\rangle \in E_s$, either $X = Y$ or $X \cap Y = \phi$. For example, in the SSG in column four of the iteration-4 row, the selector-edge $\langle n_{\{y\}}, cdr, n_{\{t\}}\rangle$ satisfies $\{y\} \cap \{t\} = \phi$. This SSG could not contain a selector-edge such as $\langle n_{\{x\}}, cdr, n_{\{x,t_1\}}\rangle$.

Invariant (ii) If $is\_shared(n) = true$ for a node n, then one or more of the following conditions must hold: (a) There exists a selector edge from n_ϕ to n. Since n_ϕ can represent multiple locations, this single edge can represent two or more selector edges in a given DSG. (b) There exist two selector edges from different shape nodes, say, n_{Z_1} and n_{Z_2} where $Z_1 \cap Z_2 = \phi$ to n. In this case, there may exist a DSG that includes two selector edges: one from the run-time location pointed to by the set of variables Z_1 and one from the run-time location pointed to by the set of variables Z_2. (c) There exist two selector-edges (with *different* selectors) from a single shape-node. (Figure 3 does not illustrate these conditions because none of the shape-nodes are shared.)

Because the converse of Invariant (ii) need not hold, sharing information must be stored explicitly in SSGs. For example, in the SSG in column three of the iteration-0 row, $is\_shared(n_\phi) = false$ even though there exists a selector edge from n_ϕ to itself and a selector edge from $n_{\{x\}}$ to n_ϕ. In this case, the fact that $is\_shared(n_\phi) = false$ indicates that n_ϕ's incoming edges represent DSG edges that can never simultaneously point to the same DSG node.

The abstraction function α is defined in Definition 4.5; α makes use of several auxiliary functions whose definitions are given in Definitions 4.4 and 4.5. Definition 4.4 defines an operation for renaming shape-nodes. (This function will be used both in the abstraction function and in the abstract semantics.)

Definition 4.4 *Let* $SG = \langle E_v, E_s\rangle$ *be a shape-graph, and let* $p\colon shape\_nodes(SG) \rightarrow \{false, true\}$ *and* $f\colon shape\_nodes(SG) \rightarrow \{n_X \mid X \subseteq PVar\}$ *be functions. We define four projection operations with respect to* f *as follows:*

$$(E_v \downarrow f) \stackrel{\text{def}}{=} \{[x, f(n)] \mid [x,n] \in E_v\}$$

$$(E_s \downarrow f) \stackrel{\text{def}}{=} \{\langle f(s), sel, f(t)\rangle \mid \langle s, sel, t\rangle \in E_s\}$$

$$(p \downarrow f)(n_X) \stackrel{\text{def}}{=} \bigvee_{\{n \mid f(n) = n_X\}} p(n)$$

Finally, $\langle\langle E_v, E_s\rangle, p\rangle \downarrow f \stackrel{\text{def}}{=} \langle\langle E_v \downarrow f, E_s \downarrow f\rangle, p \downarrow f\rangle$. □

Definition 4.5 (The Abstraction Function) *The function* $\alpha_s[DSG]\colon shape\_nodes(DSG) \rightarrow \{n_X \mid X \subseteq PVar\}$ *is defined as follows:* $\alpha_s[DSG] \stackrel{\text{def}}{=} \lambda r.n_{\{x \in PVar \mid [x,r] \in E_v\}}$

The function $induced\_is\_shared[DSG]$ *from* $shape\_nodes(DSG)$ *to* $\{false, true\}$ *is defined as follows:*

$$induced\_is\_shared[DSG](t) \stackrel{\text{def}}{=} |\{\langle *, *, t\rangle \in E_s\}| \geq 2$$

The abstraction function $\alpha\colon 2^{\mathcal{DSG}} \rightarrow \mathcal{SSG}$ *is defined by:*

$$\alpha(S) \stackrel{\text{def}}{=} \bigsqcup_{DSG \in S} \widehat{\alpha}(DSG)$$

where

$$\widehat{\alpha}(DSG) \stackrel{\text{def}}{=} let DSG' = gc(DSG) \ in$$
$$\langle DSG', induced\_is\_shared[DSG']\rangle \downarrow \alpha_s[DSG']$$

□

The core of Definition 4.5 is the operation of projection ($\downarrow$) with respect to $\alpha_s[DSG']$. The function $\alpha_s[DSG']$ establishes the relationship between the nodes of a DSG and their corresponding nodes in the SSG. For example, consider the iteration-1 row of Figure 3. In column two, DSG node l_2 is pointed to by program variables x and t_1 and is mapped by $\alpha_s[DSG']$ to SSG node $n_{\{x,t_1\}}$ (see column three). DSG nodes l_3, l_4, and l_5, which are not pointed to (directly) by any variables, are mapped to SSG node n_ϕ. In general, $\alpha_s[DSG']$ generates a finite set of SSG nodes from the *a priori* unbounded number of DSG nodes in DSG'. The projection operation then collapses the DSG onto the smaller set of nodes, while preserving aspects of its structure. We say that a shape-node n_X **represents** a shape-node n in DSG' if $\alpha_s[DSG'](n) = n_X$.

The function $induced\_is\_shared[DSG]$ checks whether a node has 2 or more predecessors in DSG. Because of the projection performed with respect to $\alpha_s[DSG']$, an SSG node's sharing value is true if any of the DSG nodes it represents has 2 or more predecessors in DSG'. (This aspect of $\alpha_s[DSG']$ is not illustrated by the example presented in Figure 3.) On the other hand, if projection-function $\alpha_s[DSG']$ sets the sharing value of SSG-node n_Z to false, this means that the DSG node (or nodes) that n_Z represents all only have at most one predecessor. For example, consider the iteration-0 row of Figure 3. In the SSG in column three, n_ϕ represents the run-time locations l_2, l_3, l_4, and l_5, each of which has exactly one predecessor in the DSG (see column two). Consequently, $is\_shared(n_\phi) = false$.

Note one role of the *gc* operation that appears in the definition of $\widehat{\alpha}$: if DSG contains a garbage shape-node that has 2 or more predecessors, this will be filtered out by the *gc* operation and will not affect the value of $is\_shared(n_\phi)$.

Example 4.6 Definition 4.5 provides a way of identifying a shape-graph with a data type. Figure 5 shows the shape-graphs that represent five kinds of data types. For each DSG SG in one of the five indicated classes, $\widehat{\alpha}(SG)$ approximates ($\sqsubseteq$) the corresponding graph shown in Figure 5. (For the moment, ignore graph (f).)

The reason why approximates ($\sqsubseteq$) is used here is that the shape-analysis algorithm is a conservative algorithm and thus the shape-graphs produced may have superfluous edges. Therefore, when the algorithm reports that a variable points to a circular list, it may actually point to a non-circular list; however, when the algorithm reports that a variable points to a non-circular list, it will never point to a circular list. This kind of conservative approximation is appropriate for use, for example, in parallelizing compilers [HNH92, HG92]. (An extension of our basic technique allows *definitely* circular structures to be identified. See Section 5.4.) □

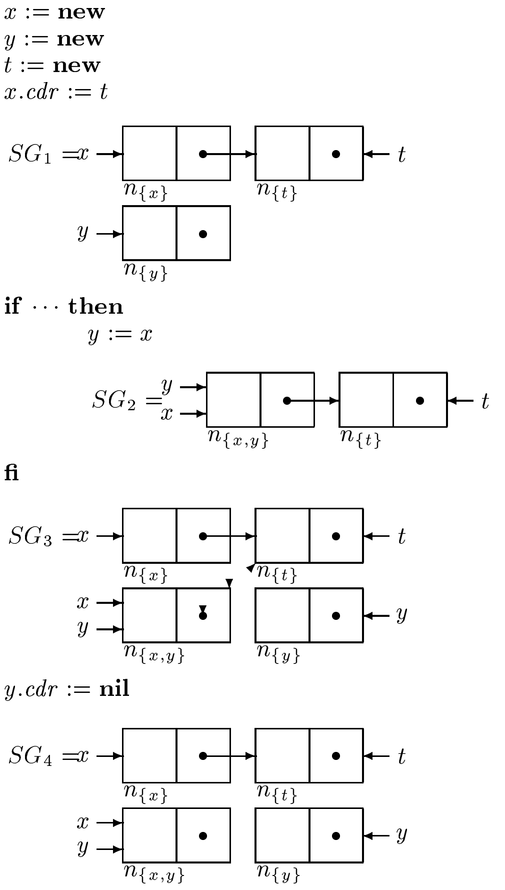
4.3 The Abstract Interpretation

The abstract meaning function $[\![\ ]\!]_{SSG}\colon \mathcal{SSG} \to \mathcal{SSG}$ for the pointer-manipulation statements is given in Figure 6. The operations presented in Figure 6 manipulate variable-edges, selector-edges, and sharing information, as well as the alias information that is maintained in the shape-node names of SSGs. It has been shown that these SSG transformers are conservative with respect to the concrete semantics defined in Figure 2 (see [SRW95]).

The key property of the abstract semantics is that each abstract assignment operation creates an SSG that conservatively covers all the possible new configurations of variable sets whose members all point to the same run-time location (i.e., DSG shape-node). This permits an unusual treatment (for a static-analysis algorithm, that is) of statements of the form $x.sel_0$:= **nil**. When the algorithm processes such a statement, it always *removes* the sel_0 edges emanating from what x points to. We call this operation *strong nullification*.

Example 4.7 Figure 4 shows a simple example that illustrates strong nullification. Note that after statement $y := x$ in the then-branch of the conditional, x and y point to the same run-time location. This is reflected in SG_2 by the fact that x and y point to a single shape-node, $n_{\{x,y\}}$. SG_3 is the union of SG_2 with SG_1; x and y each point to two shape-nodes in SG_3. Because $n_{\{x,y\}}$ in SG_3 represents only run-time locations that are pointed to by *both* x and y (which occurs only on *some* execution sequences), it is safe for the abstract semantics for statement $y.cdr$:= **nil** to eliminate the edge from $n_{\{x,y\}}$ to $n_{\{t\}}$ (see SG_4).

Note that if $n_{\{x\}}$ and $n_{\{x,y\}}$ were merged into one shape-node in SG_3, then it would not be possible to perform a strong nullification because a run-time location pointed to by x alone *does* have a *cdr*-edge emanating from it (i.e., to the node that t points to). □



```
x := new
y := new
t := new
x.cdr := t
if ··· then
    y := x
fi
y.cdr := nil
```

Figure 4: A program that illustrates strong nullification.

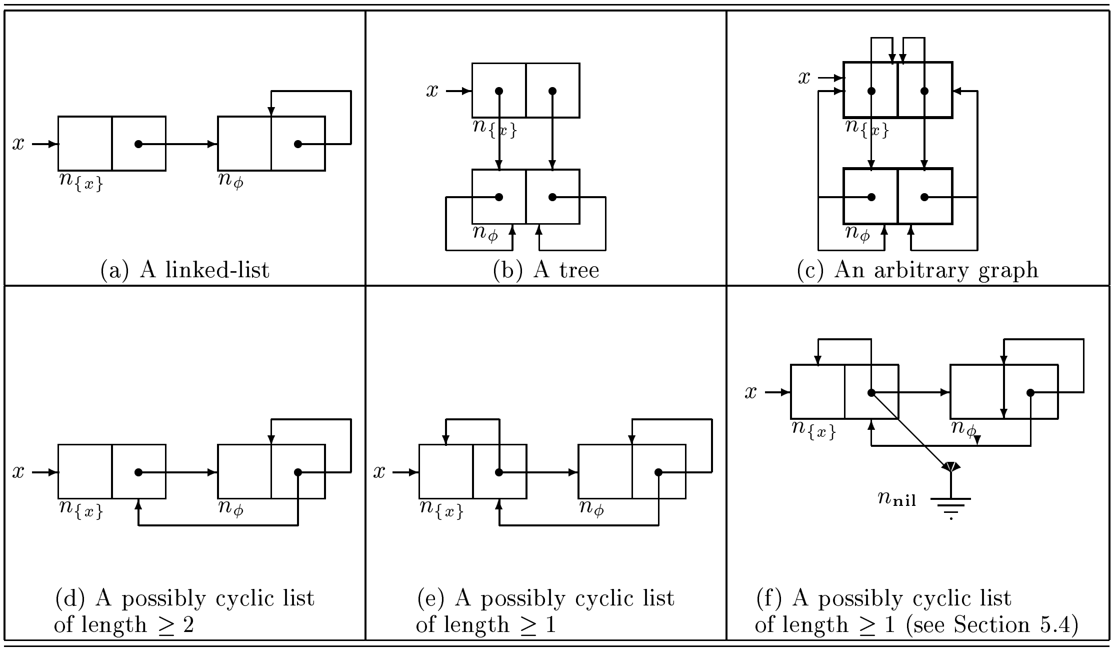
We now discuss the individual cases of the abstract meaning function (Figure 6), illustrating the most important features using Figure 7, which shows the final SSGs computed for each program point by abstract interpretation of the destructive list-reversal program. Each block of Figure 7 indicates the shape of memory just *before* the program-point label that appears at the bottom of the block. The text at the top of a block indicates the preceding program point (or points) in the control-flow graph and the action(s) taken there. For example, v_{15}'s one predecessor is the statement t := **nil** at v_{14}.

(On first reading, it may be helpful to skip the remainder of this section and proceed directly to Section 4.4.)

- For an assignment x := **nil**, the projection operation is used to rename shape-nodes by removing x from their "name". Note that this may cause what were formerly distinct shape-nodes to be merged.

  > **Example**. In the transition between block v_7 and block v_8 of Figure 7 the assignment t_1 := **nil** causes $n_{\{y,x,t_1\}}$ and $n_{\{y,x\}}$ to be merged. □

- For an assignment $x.sel_0$:= **nil**, the SSG transformer given in Figure 6 *removes* all of x's sel_0 selector-edges (what we called "strong nullification" above). The variable set of a shape-node in the SSG for a program-point v consists of variables that, for some execution



(a) A linked-list

(b) A tree

(c) An arbitrary graph

(d) A possibly cyclic list of length ≥ 2

(e) A possibly cyclic list of length ≥ 1

(f) A possibly cyclic list of length ≥ 1 (see Section 5.4)

Figure 5: SSGs that represent five kinds of data types. For each of the shape-nodes in all of the SSGs but (c), the value of *is_shared* is false. In the graph (c) both nodes are shared.

sequence ending at v, all point to the same run-time location; therefore, our method can always remove x's sel_0 selector-edges.[3]

> **Example**. In Figure 7, the transition between v_{11} and v_{12} removes all of y's *cdr* selector-edges. □

The other important aspect of the SSG transformer for $x.sel_0 :=$ **nil** is the way information in shape-node names is used to reset the sharing information. This is based on Invariant (ii) of the abstraction process, as described in Section 4.2. (The resetting of sharing information by the SSG transformer is not illustrated by the list-reversal program since *is_shared* is *false* for all shape-nodes in all shape-graphs that arise. This issue is discussed further at the end of Section 4.4.)

- For an assignment $x :=$ **new**, a new unshared node $n_{\{x\}}$ is created. All other shape-nodes are unaffected.
- For an assignment $x := y$, the shape-node names are changed to reflect the fact that whatever y was pointing to before is now also pointed to by x. In addition, new variable-edges are added to reflect the assignment of y to x.

  > **Example**. See the transition between block v_6 and block v_7 of Figure 7. □

- The SSG transformer for an assignment $x := y.sel_0$ is the most elaborate operation. The reason is that $y.sel_0$ may point to many nodes, and we have to create an SSG that conservatively covers all the possible new configurations of variable sets whose members all point to the same run-time location (i.e., DSG shape-node) after the assignment. That is, if $y.sel_0$ points to n_Z, then we need to "materialize" a copy of n_Z — producing a "new" node $n_{(Z \cup \{x\})}$ from "old" node n_Z. In defining this materialization operation, the goal is to cover conservatively all the possibilities, yet at the same time not introduce too many superfluous edges that prevent the abstract semantics from being able to verify interesting properties, e.g., that a variable points to a list.

  > **Example**. See the transition between block v_8 and block v_9 of Figure 7, in which node $n_{\{t_1\}}$ is materialized from n_ϕ. □

  In what follows, let n_Y be a shape-node that y points to. For every node n_Z pointed to by $y.sel_0$, we materialize a new node $n_{(Z \cup \{x\})}$ and direct the following edges to $n = n_{(Z \cup \{x\})}$:

  - Old variable-edges that point to n_Z before the assignment. (This does not occur in the transition between block v_8 and block v_9 of Figure 7.)
  - A new variable-edge from x. (See variable-edge $[t_1, n_{\{t_1\}}]$ in block v_9 of Figure 7.)
  - A sel_0 selector-edge from n_Y. This edge replaces the old sel_0 selector-edge that emanates from n_Y

[3] Other shape analyses do not handle this statement precisely, or handle it precisely only under certain circumstances. However, we are *not* claiming that our method is somehow "able to treat all statements precisely". In Figure 6, the inevitable loss of precision intrinsic to static-analysis occurs in the treatment of statements of the form $x := y.sel_0$ (when y points to n_ϕ), rather than in statements of the form $x.sel_0 :=$ **nil**. In particular, in the SSG transformer for $x := y.sel_0$, a node-materialization operation is used to create shape-nodes that conservatively cover all the possible new configurations of variable sets whose members all point to the same run-time location.

$$\llbracket x := \mathbf{nil} \rrbracket_{\mathcal{SSG}}(\langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle) \overset{\text{def}}{=} (gc(\langle\langle E_v - [x,*], E_s\rangle, \mathit{is\_shared}\rangle))\downarrow \lambda n_X . n_{(X-\{x\})}$$

$$\begin{array}{l}
\llbracket x.sel_0 := \mathbf{nil} \rrbracket_{\mathcal{SSG}}(\langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle) \overset{\text{def}}{=} \langle\langle E_v, E'_s\rangle, \mathit{is\_shared}'\rangle \\
\text{where } E'_s = (E_s - \{\langle n_X, sel_0, *\rangle \mid x \in X\}) \\
\text{and } \mathit{is\_shared}'(n) = \mathit{is\_shared}(n) \wedge \left(\begin{array}{cll} & \langle n_\phi, *, n\rangle \in E'_s & (a) \\ \vee & \exists n_{Z_1}, n_{Z_2} : Z_1 \cap Z_2 = \phi, \langle n_{Z_1}, *, n\rangle, \langle n_{Z_2}, *, n\rangle \in E'_s & (b) \\ \vee & \exists n_Z : \langle n_Z, car, n\rangle, \langle n_Z, cdr, n\rangle \in E'_s & (c) \end{array} \right)
\end{array}$$

$$\llbracket x := \mathbf{new} \rrbracket_{\mathcal{SSG}}(\langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle) \overset{\text{def}}{=} \langle\langle E_v \cup \{[x, n_{\{x\}}]\}, E_s\rangle, \mathit{is\_shared}[n_{\{x\}} \mapsto \mathit{false}]\rangle$$

$$\llbracket x := y \rrbracket_{\mathcal{SSG}}(\langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle) \overset{\text{def}}{=} \begin{array}{l} \text{let } \langle\langle E'_v, E'_s\rangle, \mathit{is\_shared}'\rangle = \langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle \downarrow \lambda n_Z . \begin{cases} n_{(Z\cup\{x\})} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases} \\ \text{in } \langle\langle E'_v \cup \{[x, n_Y] \mid [y, n_Y] \in E'_v\}, E'_s\rangle, \mathit{is\_shared}'\rangle \end{array}$$

$$\begin{array}{l}
\llbracket x := y.sel_0 \rrbracket_{\mathcal{SSG}}(\langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle) \overset{\text{def}}{=} \langle\langle E'_v, E'_s\rangle, \mathit{is\_shared}'\rangle \\
\text{where} \\
E'_v = E_v \cup \bigcup_{y\in Y, \langle n_Y, sel_0, n_Z\rangle \in E_s} \{[z, n_{(Z\cup\{x\})}] \mid z = x \vee [z, n_Z] \in E_v\} \\
E'_s = (E_s - \{\langle n_Y, sel_0, *\rangle \mid y \in Y\}) \cup \bigcup_{y\in Y, \langle n_Y, sel_0, n_Z\rangle \in E_s} \mathit{disjoint\_or\_equal}(\mathit{assign}(x, n_Y, sel_0, n_Z)) \\
\mathit{assign}(x, n_Y, sel_0, n_Z) = \begin{array}{cll} & \{\langle n_Y, sel_0, n_{(Z\cup\{x\})}\rangle \mid Y \neq Z\} & \text{old} \rightarrow \text{new} \\ \cup & \{\langle n_{(Z\cup\{x\})}, sel, n_W\rangle \mid (Y \neq Z \vee sel \neq sel_0), \langle n_Z, sel, n_W\rangle \in E_s\} & \text{new} \rightarrow \text{old} \\ \cup & \left\{ \begin{array}{ll} \langle n_{(Z\cup\{x\})}, sel, n_{(Z\cup\{x\})}\rangle \mid & ((Z = Y \wedge sel = sel_0) \vee \mathit{is\_shared}(n_Z)), \\ & \langle n_Z, sel, n_Z\rangle \in E_s \end{array} \right\} & \text{new} \rightarrow \text{new} \\ \cup & \left\{ \begin{array}{ll} \langle n_W, sel, n_{(Z\cup\{x\})}\rangle \mid & (y \notin W \vee sel \neq sel_0), \mathit{is\_shared}(n_Z), \\ & \langle n_W, sel, n_Z\rangle \in E_s \end{array} \right\} & \text{old} \rightarrow \text{new} \end{array} \\
\text{and } \mathit{is\_shared}'(n_Z) = \mathit{is\_shared}(n_{(Z-\{x\})})
\end{array}$$

$$\begin{array}{l}
\llbracket x.sel_0 := y \rrbracket_{\mathcal{SSG}}(\langle\langle E_v, E_s\rangle, \mathit{is\_shared}\rangle) \overset{\text{def}}{=} \langle\langle E_v, E'_s\rangle, \mathit{is\_shared}'\rangle \\
\text{where } E'_s = E_s \cup \mathit{disjoint\_or\_equal}(\{\langle n_X, sel_0, n_Y\rangle \mid [x, n_X], [y, n_Y] \in E_v\}) \\
\text{and } \mathit{is\_shared}'(n) = \mathit{is\_shared}(n) \vee |\{\langle n', *, n\rangle \in E_s \mid [y, n] \in E_v\}| \geq \mathit{1}
\end{array}$$

$$\mathit{disjoint\_or\_equal}(E_s) \overset{\text{def}}{=} \{\langle n_X, sel, n_Y\rangle \mid \langle n_X, sel, n_Y\rangle \in E_s, X = Y \vee X \cap Y = \phi\}$$

Figure 6: The SSG meaning function $\llbracket st \rrbracket_{\mathcal{SSG}}: \mathcal{SSG} \rightarrow \mathcal{SSG}$ for a statement *st*.

before the assignment (see the first old → new case in Figure 6 and selector-edge $\langle n_{\{y,x\}}, cdr, n_{\{t_1\}}\rangle$ in block v_9 of Figure 7.) This selector-edge is not added by the old → new case when $Y = Z$, because all directly cyclic selector-edges are handled by the new → new case.

- An edge $\langle n, sel_0, n\rangle$ is materialized when n_Y has a sel_0 selector-edge to itself. (See the new → new case). (This does not occur in the transition between block v_8 and block v_9.)
- Suppose there is a selector-edge $\langle n_Y, sel_0, n_Z\rangle$. When n_Z has a *sel* selector-edge to itself, a selector-edge $\langle n, sel, n\rangle$ is materialized if $\langle n_Z, sel, n_Z\rangle$ and $\langle n_Y, sel_0, n_Z\rangle$ represent edges that can simultaneously co-exist in some DSG. This can happen only if n_Z is a shared node. (See the new → new case). (This also does not occur in the transition between block v_8 and block v_9.)
- Selector-edges from other old predecessors of n_Z need to be connected to n if they can simultaneously coexist with the sel_0 selector-edge from n_Y (see the second old → new case). Here, we take advantage of the variables in shape-node names; in particular, a predecessor of n_Z that has y in its name is incompatible with n_Y. (This does not occur in the transition between block v_8 and block v_9.)

We also connect n to the old successors of n_Z for all the selector-edges where $Y \neq Z$ or $sel \neq sel_0$ (see

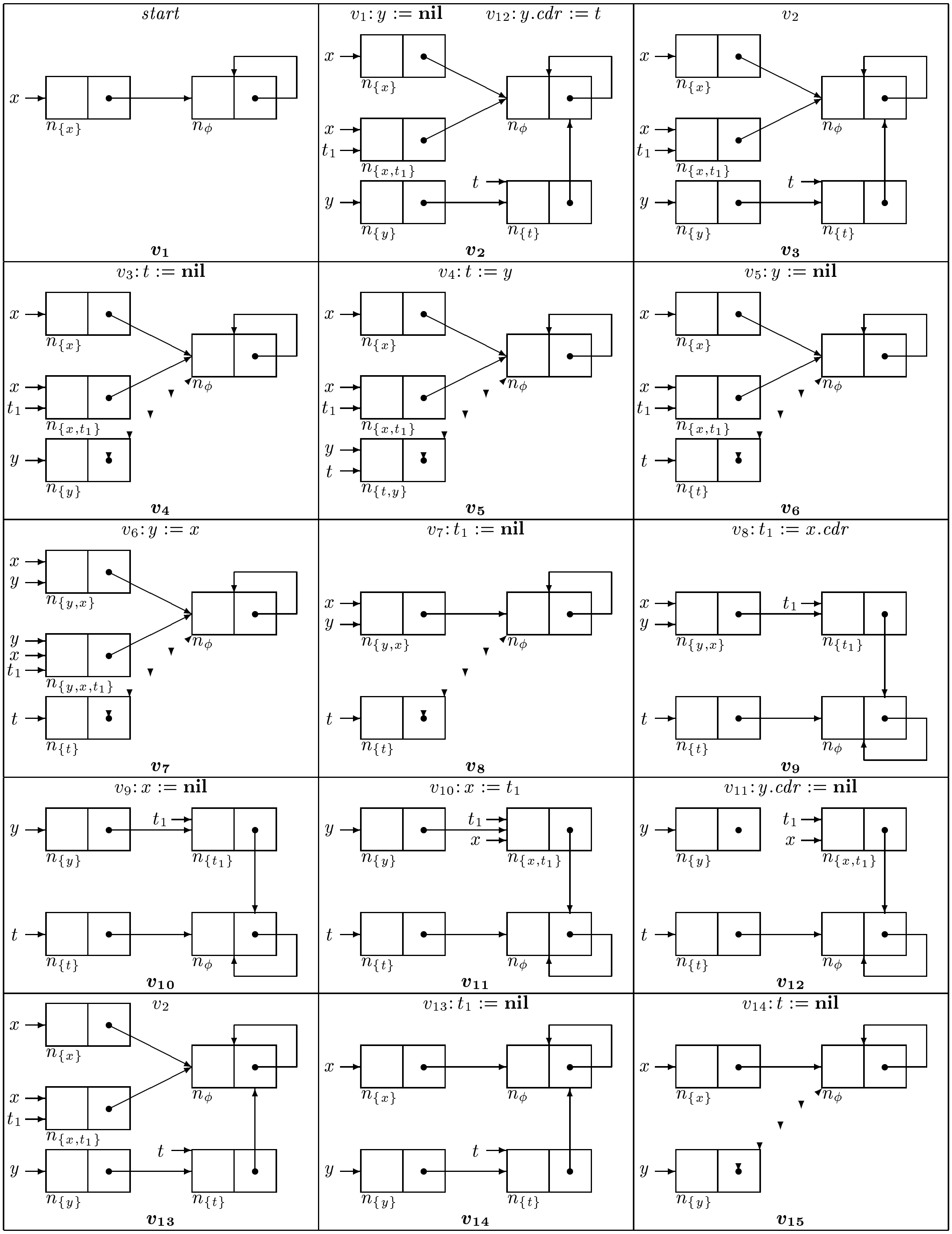


Figure 7: The final SSGs computed for each control-flow-graph vertex by abstract interpretation of the destructive list-reversal program (e.g., block v_2 corresponds to column four of the iteration-4 row of Figure 3). For each of the shape-nodes in all of the SSGs, the value of *is_shared* is false.

the new → old case). (This does not occur in the transition between block v_8 and block v_9.)

The operation *disjoint_or_equal*: $\mathcal{SSG} \to \mathcal{SSG}$ eliminates selector-edges whose end-points do not satisfy Invariant (i) of the abstraction process (the "equality-or-disjointness" property for the variable-set names of selector-edge end-points described in Section 4.2). (This does not filter out any edges in the transition between block v_8 and block v_9.)

- Finally, for an assignment $x.sel_0 := y$, a sel_0 selector-edge is added between shape-nodes pointed to by x and shape-nodes pointed to by y. In addition, all nodes that are pointed to by both y and a selector-edge before the assignment are now considered to be shared nodes.

  Example. See the transition between block v_{12} and block v_2 of Figure 7. □

The abstract semantics associates an SSG, SG_v, with each $v \in V$. Equationally, this can be defined as the least fixed point (under the ordering defined in Definition 4.3) of the following system of equations in SG_v:

$$SG_v = \left\{ \begin{array}{ll} \langle\langle \phi, \phi\rangle, \lambda n.\mathit{false}\rangle & \text{if } v = \mathit{start} \\ \bigsqcup_{\langle u,v\rangle \in A} [\![st(u)]\!]_{SSG}(SG_u) & \text{otherwise} \end{array} \right\} (1)$$

The least-fixed point of these equations can be found by iteration, starting from $\langle\langle \phi, \phi\rangle, \lambda n.\mathit{false}\rangle$.

Example 4.8 The final abstract values for all of the vertices of the list-reversal program's control-flow graph are shown in Figure 7. Among other things, this information tells us is that if x's value is a list at the beginning of the program (see block v_1) then y's value is a list at the end of the program (see block v_{15}).

Column four of Figure 3 shows the SSGs computed for vertex v_2 of the list-reversal program during the successive iterations of the fixed-point-finding procedure. The final abstract value for vertex v_2 (i.e., SG_{v_2}) is the graph shown in the iteration-4 row of column four. The elements of this graph can be interpreted as follows:

- There are two shape-nodes that represent the head of the list that x points to: $n_{\{x\}}$ and $n_{\{x,t_1\}}$. The former represents the situation where x points to the head of the list and t_1 points elsewhere (which only happens before the first iteration of the loop). The latter represents the situation where x and t_1 *both* point to the head of the list. Shape-node $n_{\{y\}}$ represents the head of the reversed list that y points to. Shape-node $n_{\{t\}}$ represents the list that t points to, which is a sublist of the list that y points to. Shape-node n_ϕ represents all the run-time locations in the tails of the lists that x and t point to.
- For each of the shape-nodes in the graph, the value of *is_shared* is false. The fact that *is_shared*$(n_\phi) = \mathit{false}$ tells us a number of interesting things about the memory state (i.e., DSG) produced by any execution sequence that ends at vertex v_2: (1) It implies that selector-edges from x and from t cannot point to the same node (and consequently the tails of x and t cannot have a component in common). (2) Similarly, for every pair of different run-time locations in the tail of x or t, the selector-edges from these run-time locations cannot point to the same node. Consequently, variables x and t must point to acyclic lists that do not share any storage in common.

□

Given how complicated the semantic equations in Figure 6 are, the following theorem, whose proof can be found in [SRW95], is reassuring:

Theorem 4.9 (Correctness Theorem) *For every control-flow-graph vertex v, $\alpha(cs(v)) \sqsubseteq SG_v$.* □

4.4 What the Analysis Algorithm Achieves and Why

The abstract interpretation defined in Section 4.3 yields a new shape-analysis algorithm for finding out information about the possible "shapes" that heap-allocated structures in a program can take on. For certain programs — including ones in which a significant amount of destructive updating takes place — this algorithm is able to verify shape-preservation properties. Examples of such properties include: (i) when the input to the program is a list, the output is (still) a list; (ii) when the input to the program is a tree, the output is (still) a tree; and (iii) when the input to the program is a possibly circular list, the output is a possibly circular list. For instance, we are able to conclude from the information reported by the algorithm about the list-reversal program that "list-ness" is preserved (see Figure 7).

The algorithm is also able to determine that

- "list-ness" is preserved by the list-insert program given in Figure 8 (which searches a list and splices a new element into the list).
- "circular list-ness" is also preserved by the list-insert program. More precisely, if at the beginning of the insert program x is a possibly cyclic list of length ≥ 1 (see Figure 5(e)), then at the end of the program, x is a possibly cyclic list of length ≥ 2 (see Figure 5(d)). (For details, see [SRW95, Appendix B].)

It is instructive to consider the main reasons why the shape-analysis algorithm is able to produce accurate information about the list-reversal program. In analyzing this program, the key issue is: "How does the algorithm keep the y list separate from the x list?" There are two aspects of the algorithm that contribute to the successful handling of this problem.

Cutting the list. The more clear-cut aspect is the removal of y's *cdr* selector-edges by $y.cdr :=$ **nil** via strong nullification in the transition from block v_{11} to block v_{12} in Figure 7. This cuts the y list at the head, separating the first element, $n_{\{y\}}$, from the tail, which x and t_1 point to.

Materialization of $n_{\{t_1\}}$ from summary-node n_ϕ. Equally important is the way the algorithm handles the advancement of t_1 down the x list by $t_1 := x.cdr$ in the transition from v_8 to v_9. At v_8, $x.cdr$ points to n_ϕ; however, the node-materialization operation causes a *new* **non-summary** *shape-node*, $n_{\{t_1\}}$, to be materialized out of n_ϕ.[4]

In the shape analysis of the list-reversal program, there is a crucial interaction between these two aspects. Suppose, for example, that in the transition between blocks v_8 and v_9 shape-node $n_{\{t_1\}}$ was not materialized out of summary-node n_ϕ, but instead variable t_1 was merely set to point to n_ϕ. At v_{11}, variables t_1 and x would then both point to

[4] Jocularly, we refer to n_ϕ as the "primordial soup", and the process of materializing a node such as $n_{\{t_1\}}$ from n_ϕ as "ladling a node out of the soup".

n_ϕ. The removal of y's *cdr* selector-edges in the transition from v_{11} to v_{12} would still separate the node that y points to ($n_{\{y\}}$) from the list pointed to by x and t_1 (which in this case is n_ϕ.). However, the very next transition, from v_{12} to v_2, would set y's *cdr* field to t, whose *cdr* field points to n_ϕ, which is what x points to. At this stage, the two lists are no longer known to be separate lists!

Note how differently things turn out when $n_{\{t_1\}}$ is materialized from n_ϕ in the transition from v_8 to v_9: At v_{12}, t_1 and x point to $n_{\{x,t_1\}}$, and thus in the transition from v_{12} to v_2 when y's *cdr* field is set to t, whose *cdr* field points to n_ϕ, *x does* **not** *point to n_ϕ*. Although n_ϕ occurs in both the tail of x and the tail of y, because $\mathit{is\_shared}(n_\phi) = \mathit{false}$ we know that the two lists cannot share any storage in common; that is, x and y must point to disjoint acyclic lists.

The two operations discussed above — cutting a list and advancing a pointer down a list — are two of the four main operations of most list-manipulation algorithms. The third and fourth common list-manipulation operations — splicing a new element into a list and removing an element from a list — can, in many cases, be handled accurately by our shape-analysis algorithm, *even if shape-nodes temporarily become shared!* (This is not illustrated by the list-reversal program, but is discussed in the next paragraph.) This points up the strength of our approach: Our algorithm handles all four of the basic list-manipulation operations with a remarkable degree of precision — as well as similar tree- and circular-list-manipulation operations.

Let us now turn to the issue of how information in shape-node names can sometimes be used to reset a shape-node's sharing information from *true* to *false*. This ability is the main reason why our algorithm is able to determine that the list-insert program of Figure 8 preserves both "list-ness" and "circular list-ness".

This situation arises in the list-insert program at vertices v_{11}, v_{12}, and v_{13} of the control-flow graph, where the new element is spliced into the list. (We assume that at the beginning of the program shown in Figure 8, x points to an unshared list of length 1 or more and e points to the new element to be inserted.) The key step is the transition from $v_{12}: y.cdr := \mathbf{nil}$ to v_{13}. In the immediately preceding transition, from v_{11} to v_{12} (see Figures 9(a) and 9(b)), $e.cdr$ is assigned the value t, which adds a new selector-edge into $n_{\{t\}}$ and causes $\mathit{is\_shared}(n_{\{t\}})$ to be set to *true* in the shape-graph for v_{12}.

In the SSG transformer given in Figure 6 that covers the case of assignments of the form $y.cdr := \mathbf{nil}$, information in shape-node names is used to reset the sharing information. In particular, $n_{\{t\}}$ meets none of the three conditions for *is_shared* to be *true* at vertex v_{13} and so $\mathit{is\_shared}(n_{\{t\}})$ is reset to *false* at v_{13}. (See Figure 9(c).)

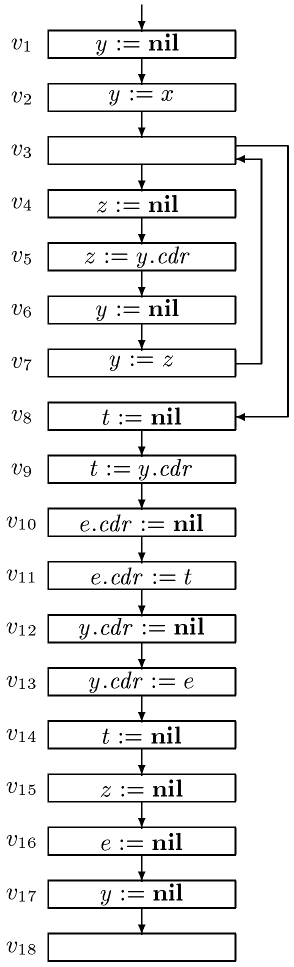
Remark. It is interesting to note that if the assignment at v_{12} were $e.cdr := \mathbf{nil}$, rather than $y.cdr := \mathbf{nil}$, $\mathit{is\_shared}(n_{\{t\}})$ would be reset to *false* at v_{13}, even though there would be *two* incoming selector-edges to $n_{\{t\}}$: $\langle n_{\{z,y\}}, cdr, n_{\{t\}}\rangle$ and $\langle n_{\{x,y\}}, cdr, n_{\{t\}}\rangle$. This is consistent with the concrete semantics because the shape-node names $n_{\{z,y\}}$ and $n_{\{x,y\}}$ tell us that $\langle n_{\{z,y\}}, cdr, n_{\{t\}}\rangle$ and $\langle n_{\{x,y\}}, cdr, n_{\{t\}}\rangle$ are "incompatible". Because $\{z, y\} \cap \{x, y\} \neq \phi$, we know that $n_{\{z,y\}}$ and $n_{\{x,y\}}$ do not represent nodes that co-exist in any DSG. This explains condition (b) in the $y.cdr := \mathbf{nil}$ case of Figure 6. □

```
$y := x$
while $y.cdr \neq$ nil $\wedge \ldots$ do
    $z := y.cdr$
    $y := z$
od
$t := y.cdr$
$e.cdr := t$
$y.cdr := e$
$t :=$ nil
$z :=$ nil
$e :=$ nil
$y :=$ nil
```

(a)

```
$y :=$ nil
$y := x$
while $y.cdr \neq$ nil $\wedge \ldots$ do
    $z :=$ nil
    $z := y.cdr$
    $y :=$ nil
    $y := z$
od
$t :=$ nil
$t := y.cdr$
$e.cdr :=$ nil
$e.cdr := t$
$y.cdr :=$ nil
$y.cdr := e$
$t :=$ nil
$z :=$ nil
$e :=$ nil
$y :=$ nil
```

(b)



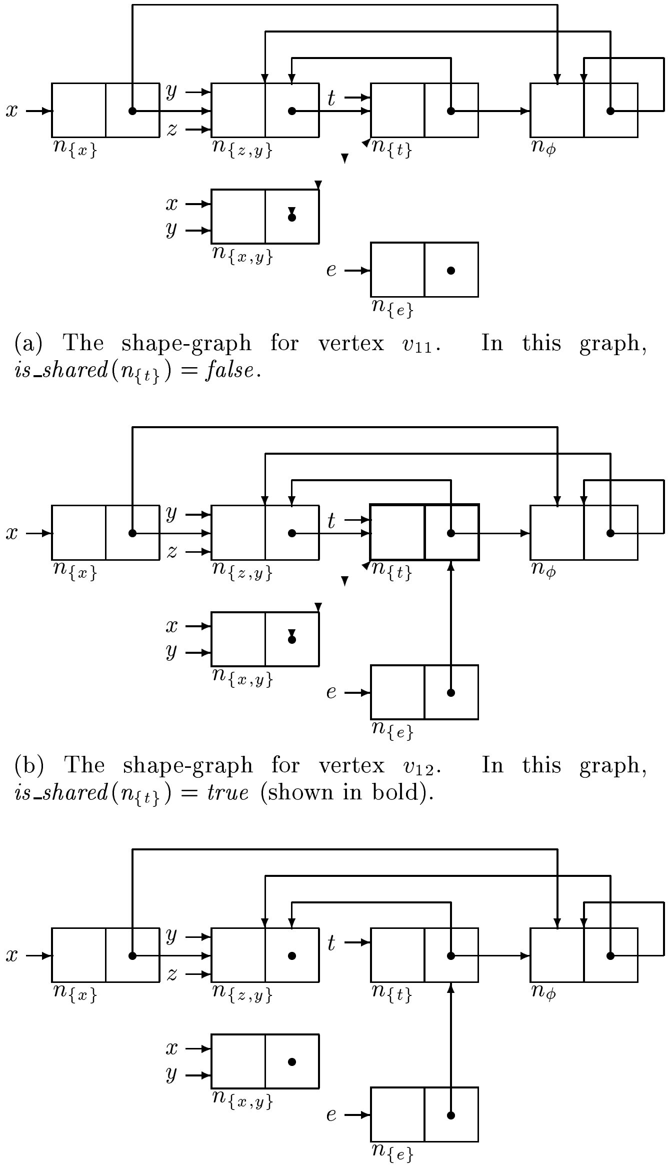
(c)

Figure 8: A program that searches a list and splices a new element into the list.

5 Extensions

5.1 Merging Shape Nodes

The number of shape-nodes in an SSG is bounded by $2^{|PVar|}$. Unfortunately, for some pathological programs the number of shape-nodes can actually grow to be this large (although our limited experience to date suggests that this is unlikely to arise in practice). It is possible to overcome this problem by making use of a widening operator that merges selected shape-nodes. By this means, we can guarantee that a fixed point of Equation (1) of Section 4.3 can be found in polynomial time; the widening operator simply has to be applied whenever necessary to limit the cardinality of shape-node name sets to some chosen constant. (This is similar in spirit to k-limiting [JM81], but is likely to produce more accurate results because limiting the cardinality of name sets still preserves most of the structural information about the graph.) Details can be found in [SRW95].



(a) The shape-graph for vertex v_{11}. In this graph, *is_shared*$(n_{\{t\}}) =$ *false*.

(b) The shape-graph for vertex v_{12}. In this graph, *is_shared*$(n_{\{t\}}) =$ *true* (shown in bold).

(c) The shape-graph for vertex v_{13}. In this graph, *is_shared*$(n_{\{t\}}) =$ *false*.

Figure 9: The shape-graphs at vertices v_{11}, v_{12}, and v_{13} in the list-insert program. These illustrate how *is_shared*$(n_{\{t\}})$ is reset to *false* in the shape-graph for vertex v_{13}.

5.2 Finding Aliases and Sharing

It is possible to use our shape-analysis algorithm to determine whether two pointer variables x and y are possible aliases just before vertex v by testing whether x and y point to a common shape-node in SSG SG_v. If x and y do point to a common node, we (conservatively) conclude that they may be aliases. It is possible to extend this to a test of whether two *access paths* are "may aliases", as follows: First, we instrument the original program with two additional temporary variables (say t_1 and t_2) and code to advance t_1 and t_2 down the two pointer-access paths in question. The code is inserted just before v. Second, we apply our shape-analysis algorithm. Third, we look in the SSG computed for vertex v to see if t_1 and t_2 may be aliases.

We can also use this approach to determine if there is possible sharing between components of two heap-allocated data structures, which is precisely the kind of information needed to be able to compile programs to take advantage of coarse-grained parallelism. (See [SRW95].)

5.3 Interprocedural Analysis

Our method can also be extended to handle procedure calls in a conservative way. Two fundamental problems need to be resolved:

- Representing multiple occurrences of the same local variable in (mutually) recursive procedures.
- Accounting for the different calling contexts in which a procedure can occur.

To approximate the local variables of recursive calls, we introduce an extra variable $\overline{x}$ for every local variable x. Variable $\overline{x}$ is used as a representative for all copies of x in other scopes. Shape-nodes whose name sets contain only barred variables are a new kind of "summary node". Like n_ϕ, they can represent multiple runtime locations from a single DSG. Using these ideas, we have extended the abstract semantics to handle procedure calls and returns.

The second problem can be resolved using one of the known interprocedural techniques of Sharir and Pnueli [SP81]. For example, a simple conservative solution is to consider a procedure call as a goto to the called procedure and a return from a procedure P as a goto to all the statements that follow an invocation of P. A more accurate solution can be determined by tabulating a "shape-graph-transformation" function for each procedure.

5.4 Representing Definitely Circular Structures

In the SSGs defined in Section 4.1, there are no elements that represent the fields of nodes that point to atoms or **nil** (or are uninitialized). One consequence of this is that the shape-analysis algorithm is only able determine rather weak "data-type" properties. As pointed out in Example 4.6, when the algorithm reports that a variable points to a circular list, it may actually point only to a non-circular list. That is, the type "circular list" really means "possibly circular list".

By introducing three additional nodes, $n_{\mathbf{atom}}$, $n_{\mathbf{nil}}$, and $n_{\mathbf{uninit}}$, much more accurate type properties can be obtained in many cases. We impose the invariant on SSGs that all fields of shape-nodes have *at least one* out-going selector-edge (possibly to $n_{\mathbf{atom}}$, $n_{\mathbf{nil}}$, or $n_{\mathbf{uninit}}$). The consequence of this refinement is that this modified domain of SSGs is capable of representing *definitely cyclic* data structures.

For example, with this extension the SSG shown in Figure 5(d) represents a definitely cyclic list of length ≥ 2 (modulo the absence of edges from the *car* fields to $n_{\mathbf{atom}}$ in the two shape-nodes); Figure 5(e) represents a definitely cyclic list of length ≥ 1; and Figure 5(f) represents a *possibly* cyclic list of length ≥ 1.

6 Related Work

The shape-analysis problem was originally investigated by Reynolds for a Lisp-like language with no destructive updating [Rey68]. Reynolds treated the problem as one of simplifying a collection of set equations. A similar shape-analysis problem, but for an imperative language supporting non-destructive manipulation of heap-allocated objects, was formulated independently by Jones and Muchnick, who treated the problem as one of solving (i.e., finding the least fixed-point of) a collection of equations using regular tree grammars [JM81].

In that same paper, Jones and Muchnick also began the study of shape analysis for languages *with* destructive updating. To handle such languages, they formulated an analysis method that associates program points with sets of finite shape-graphs.[5] To guarantee that the analysis terminates for programs containing loops, the Jones-Muchnick approach limits the length of acyclic selector paths by some chosen parameter k. All nodes beyond the "k-horizon" are clustered into a summary node. The k-limiting approach has two inherent drawbacks:

- The analysis yields poor results for programs that manipulate elements beyond the k-horizon. For example, in the list-reversal program of Figure 1, little useful information is obtained. The analysis algorithm must model what happens when the program is applied to lists of length greater than k. However, the tail of such a list will be treated (conservatively) as an arbitrary, and possibly cyclic, data structure.
- The analysis may be extremely costly because the number of possible shape-graphs is doubly exponential in k.

In addition to Jones and Muchnick's work, k-limiting has also been used in a number of subsequent papers (e.g., [HPR89]).

Whereas Jones and Muchnick use *sets* of shape-graphs (in [JM81]), our work follows Jones and Muchnick [JM82], Larus and Hilfinger [LH88, Lar89], Chase, Wegman, and Zadeck [CWZ90], and Stransky [Str92] who developed shape-analysis methods that associate each program point with a *single* shape-graph. The use of a single shape-graph is possibly less accurate than a method based on sets of graphs, but it leads to more compact representations, and thus is more likely to lead to a practical shape-analysis algorithm.

Jones and Muchnick [JM82], Chase, Wegman, and Zadeck [CWZ90], and Stransky [Str92] present similar methods in which the shape-nodes correspond to a program's allocation sites. These methods are more efficient than the methods discussed earlier, both from a theoretical perspective [CWZ90] and from an implementation perspective [AW93].

The algorithm presented by Chase, Wegman, and Zadeck is based on the following ideas:

- Sharing information in the form of heap reference counts (using 0, 1, and ∞) is used to characterize shape-graphs that represent list structures.[6]
- Several heuristics are introduced to allow several shape-nodes to be maintained for each allocation site.
- When $x.sel_0$ is assigned to and $x.sel_0$ points to a shape-node that represents a unique run-time location, the sel_0 selector-edges emanating from the shape-node that x points to are overwritten (a so-called "strong update").

The Chase-Wegman-Zadeck algorithm is able to identify list-preservation properties in some cases; for instance, it can determine that a program that appends a list to a list preserves "list-ness". However, as noted by Chase, Wegman, and Zadeck, allocation-site information alone is insufficient to determine interesting facts in many programs. For example, it cannot determine that "list-ness" is preserved for either the insert program or the reverse program. In particular, in the reverse program, the Chase-Wegman-Zadeck algorithm reports that y points to a possibly cyclic structure and that the structures that x and y point to possibly share elements in common.

There are two major technical differences between our algorithm and the Chase-Wegman-Zadeck algorithm that lead to the improvements in accuracy obtained by our algorithm:

"Strong Nullification" For an assignment $x.sel_0 := y$, the Chase-Wegman-Zadeck method ordinarily performs a "weak update" (i.e., selector-edges emanating from what x points to are *accumulated*). It performs a strong update only under certain specialized conditions.

In our algorithm, because of the Normalization Assumptions of Section 2.1, an assignment statement $x.sel_0 := y$ is transformed into two statements: $x.sel_0 :=$ **nil**, followed immediately by $x.sel_0 := y$. When our algorithm processes the first of these statements, it (always) removes the sel_0 edges emanating from what x points to. We have called this operation "strong nullification", by analogy with "strong update". When the algorithm processes the second statement, it introduces sel_0 edges that emanate from the shape-node that x points to. Taken together, the effect is to *overwrite* the sel_0 edges emanating from the shape-node that x points to — in other words, for a statement in the original program of the form $x.sel_0 := y$, our algorithm *always performs a strong update*.

> **Example**. In SG_3 of Figure 4, $n_{\{x\}}$, $n_{\{y\}}$, and $n_{\{x,y\}}$ are separate nodes. Because $n_{\{x,y\}}$ represents only run-time locations that are pointed to by *both* x and y, it is safe for the abstract semantics to perform a strong nullification to $n_{\{x,y\}}$ (see SG_4). □

The reason why it is possible for our algorithm to perform strong nullifications (and hence strong updates) is because each abstract assignment operation of the abstract semantics creates an SSG that conservatively covers all the possible new configurations of variable sets whose members all point to the same run-time location (i.e., DSG shape-node). If x is in the name of an SSG shape-node n, then n represents a DSG node whose sel_0 field will definitely be overwritten.

Materialization In an assignment statement of the form $x := y.sel_0$, our algorithm materializes new shape-nodes that conservatively cover all the possible new configurations of variable sets whose members all point to the same run-time location. For example, when $y.sel_0$ points to n_ϕ, our algorithm materializes a new node $n_{\{x\}}$ out of n_ϕ. Furthermore, if $\textit{is\_shared}(n_\phi) = \textit{false}$, this information is used to exclude both of the two possible selector-edges from n_ϕ to $n_{\{x\}}$.

[5] In this section, we use the term "shape-graph" in the generic sense, meaning any finite graph structure used to approximate the shapes of run-time data structures.

[6] The idea of augmenting shape-graphs with sharing information also appears in the earlier work of Jones and Muchnick [JM81].

> In programs that use a loop containing an assignment $x := x.cdr$ to traverse an unshared linked list, this technique permits our method to determine that x points to an unshared list element on every iteration. For instance, this occurs in the transition from block v_8 to block v_9 in Figure 7. As explained in Section 4.4, the materialization of $n_{\{t_1\}}$ in block v_9 is one of the key aspects of our algorithm that allows it to determine that the list-reversal program preserves "list-ness".
>
> The Chase-Wegman-Zadeck algorithm lacks a node-materialization operation (although they did recognize that the lack of one was a stumbling block to the accuracy of their method [CWZ90, pp. 309]).

Chase, Wegman, and Zadeck use reference-count values 0, 1, and ∞, whereas we use a Boolean-valued *is_shared* value. However, this does not represent a significant difference because in our SSGs the selector-edges allow recovering the distinction between 0 (no incoming edges) and 1 (at least one incoming selector-edge, but *is_shared* = *false*).

Our method has been presented within the framework of abstract interpretation, which allows us to prove that the algorithm obtained is conservative with respect to the concrete semantics. Chase, Wegman, and Zadeck give only informal arguments about the correctness of their algorithm. Because of several *ad hoc* features of the Chase-Wegman-Zadeck method, several changes would be necessary to reformulate it as an abstract interpretation. For instance, the rules they give for the "join" operation are complicated by the fact that the result of "joining" two shape-graphs depends on the program point at which the operation is applied. (For this reason, "join" is a misnomer in the lattice-theoretic sense.) In contrast, our join operation, which is essentially graph union, is the join operation in the lattice of SSGs defined in Section 4.1.

Larus and Hilfinger [LH88, Lar89] devised a shape-analysis algorithm that is based on somewhat different principles from the aforementioned work. As with our algorithm, shape-nodes are labeled with some auxiliary information. At first glance, their node-labeling scheme appears to be more general than ours: Whereas we use a *set of variables* to label each node, they use a *regular expression* (limited to be no longer than some chosen constant k) representing pointer-access paths that may lead to an instance of the node. However, their shape-node labels do not add any information to their representation because the pointer-access expressions can always be reconstructed from the graph stripped of node labels. In contrast, our labels — which in some sense represent regular expressions of length-1 — *do* contribute essential information to our representation: When x is in the variable-set of shape-node n_X, we know that a strong nullification (and hence a strong update) can be performed on the selector-edges emanating from n_X.

It is possible that it would be worthwhile to extend our technique to use more complicated shape-node names of the kind that Larus and Hilfinger use. However, on many interesting examples, even with our "length-1 labels", our algorithm achieves greater accuracy than the Larus-Hilfinger algorithm does, no matter what value of k is chosen: For example, the Larus-Hilfinger algorithm is not able to determine that programs such as the list-reversal and list-insert programs preserve "list-ness".

There are also several algorithms for finding may aliases that are not based on shape-graphs. The most sophisticated ones are those of Landi and Ryder [LR91] and Deutsch [Deu94]. Deutsch's algorithm is particularly interesting because, for certain programs that manipulate lists, it offers a way of representing the exact (infinite set of) may aliases in a compact way. It can be shown that Deutsch algorithm yields may-alias information for the list-reversal program that is equivalent to that produced by the algorithm of Section 4.1. However, both the Landi-Ryder and Deutsch algorithms do not determine that either "list-ness" or "circular list-ness" is preserved by the insert program of Figure 9. The reason is that due to the lack of a strong-nullification operation, these algorithms cannot infer that the assignment $y.cdr :=$ **nil** in the program shown in Figure 8(b) cuts the list pointed to by x (see Figures 9(b) and (c)). We do not mean to imply that our method dominates the Landi-Ryder and Deutsch algorithms; there exist programs in which the Deutsch algorithm is more accurate than our algorithm.

A different approach was taken by Hendren, who designed an algorithm that handles only acyclic data structures [HN90, Hen90]. Because of the choice to work with programs that only manipulate acyclic structures, the algorithm does not have to have a way of representing cycles conservatively. For this alias-analysis problem, she has given an efficient algorithm that manipulates matrices that record access paths that are aliased.

To the best of our knowledge, Hendren's algorithm is the only algorithm besides ours that can detect that insertion of an element into a list (respectively, tree) preserves the list (tree) structure. However, by design, Hendren's algorithm cannot determine such structure-preservation properties for programs that handle cyclic lists.

Myers presented an algorithm for interprocedural bit-vector problems that accounts for aliasing [Mye81]. Like our shape-analysis algorithm, his algorithm also keeps track of sets of aliased variables. He conjectured that in practice the sizes of the alias sets remain small. However, Myers's work does not handle heap-allocated storage and destructive updating. Therefore, his algorithm is significantly simpler and he is even able to show that it is precise. In contrast, it is undecidable to give a precise solution to our problem, even in the absence of procedure calls [Lan92, Ram94].

Acknowledgments

We are grateful for the helpful comments of Alain Deutsch, Christian Fecht, and Neil Jones. Laurie Hendren provided us with extensive and very helpful information about the capabilities of her analysis technique.

References

[AW93] U. Assmann and M. Weinhardt. Interprocedural Heap Analysis For Parallelizing Imperative Programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82. IEEE Press, September 1993.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1990.

[Deu92] A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1994.

[Hen90] L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, Jan 1990.

[HG92] L. Hendren and G.R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the International Conference on Computer Languages*, pages 242–251, 1992.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[HNH92] L. Hendren, A. Nicolau, and J. Hummel. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 249–260, June 1992.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 28–40, 1989.

[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: Theory and Applications*, 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. In *ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 1992.

[Lar89] J.R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, 1989.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 21–34, 1988.

[LR91] W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *ACM Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[PCK93] J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[Rey68] J.C. Reynolds. Automatic computation of data set definitions. In *Information Processing 68: Proceedings of the IFIP Congress*, pages 456–461, New York, NY, 1968. North-Holland.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[SRW95] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. Technical Report TR-1276, Computer Sciences Department, University of Wisconsin, Madison, WI, July 1995. Available on the WWW from URL http://www.cs.wisc.edu/trs.html.

[Str92] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Information and Computation*, 101(1):70–102, November 1992.