Solving Shape-Analysis Problems in Languages with Destructive Updating

MOOLY SAGIV
Tel-Aviv University
and
THOMAS REPS
University of Wisconsin
and
REINHARD WILHELM
Universität des Saarlandes

This article concerns the static analysis of programs that perform destructive updating on heap-allocated storage. We give an algorithm that uses finite shape graphs to approximate conservatively the possible "shapes" that heap-allocated structures in a program can take on. For certain programs, our technique is able to determine such properties as (1) when the input to the program is a list, the output is also a list and (2) when the input to the program is a tree, the output is also a tree. For example, the method can determine that "listness" is preserved by (1) a program that performs list reversal via destructive updating of the input list and (2) a program that searches a list and splices a new element into the list. None of the previously known methods that use graphs to model the program's store are capable of determining that "listness" is preserved on these examples (or examples of similar complexity). In contrast with most previous work, our shape analysis algorithm is even accurate for certain programs that update cyclic data structures; that is, it is sometimes able to show that when the input to the program is a circular list, the output is also a circular list. For example, the shape-analysis algorithm can determine that an insertion into a circular list preserves "circular listness."

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*; *dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*optimization*; E.1 [**Data**]: Data Structures—*graphs; lists*; *trees*; E.2 [**Data**]: Data Storage Representations—*composite structures*; *linked representations*; F.3.1 [**Logics and Meanings of**

A preliminary version of this article appeared in the Conference Record of the 23 ACM Symposium on Principles of Programming Languages.
Part of this research was done while Sagiv was visiting the University of Chicago; part was done while visiting the Universität des Saarlandes, partially supported by SFB 124-VLSI-Design Methods and Parallelism of the Deutsche Forschungsgemeinschaft; part was done while visiting the University of Wisconsin, supported by a David and Lucile Packard Fellowship for Science and Engineering 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). Reps is supported in part 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).
Sagiv's Address: Dept. of Computer Science; School of Mathematical Sciences; Tel-Aviv University; Tel-Aviv 69978; Israel. E-mail: sagiv@math.tau.ac.il.
Reps's Address: Computer Sciences Department; University of Wisconsin; 1210 West Dayton Street; Madison, WI 53706; USA. E-mail: reps@cs.wisc.edu.
Wilhelm's Address: Fachbereich 14 Informatik; Universität des Saarlandes; 66123 Saarbrücken; Germany. E-mail: wilhelm@cs.uni-sb.de.

Programs]: Specifying and Verifying and Reasoning about Programs—*assertions*; *invariants*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Algorithms, Languages, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, alias analysis, dataflow analysis, destructive updating, pointer analysis, shape analysis, shape graphs, static analysis

1. INTRODUCTION

This article 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, *sharing-analysis* problems, *storage analysis* problems (also known as *shape analysis* problems), or *typechecking* problems. The information obtained is useful, for instance, for generating efficient sequential or parallel code.

Throughout most of the article, we emphasize the application of our work to shape-analysis problems. The goal of shape analysis is to give, for each program point, a conservative, finite characterization of the possible "shapes" that the program's heap-allocated data structures can have at that point. We illustrate our approach by means of a running example in which we apply the shape-analysis technique to a program that uses destructive-updating operations to reverse a list. This example also illustrates the connection between shape analysis and typechecking: 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 7.1.

This article presents a new shape-analysis algorithm. For certain programs—including ones in which a significant amount of destructive updating takes place—the algorithm is able to verify such shape-preservation properties as (1) when the input to the program is a list, the output is also a list and (2) when the input to the program is a tree, the output is also a tree. For instance, the method can determine that "listness" is preserved by (1) a list-reversal program that performs the reversal by destructively updating the input list and (2) a list-insert program that searches a list and splices a new element into the list. Furthermore, the shape-analysis algorithm is even accurate for certain programs that update cyclic data structures; that is, it is sometimes able to show that when the input to the program is a circular list, the output is also a circular list. For example, the shape-analysis algorithm can determine that an insertion into a circular list preserves "circular listness."

These are rather surprising capabilities. None of the previously developed methods that use graphs to solve shape-analysis problems are capable of determining that "listness" is preserved on these examples (or examples of similar complexity) [Chase et al. 1990; Jones and Muchnick 1981; Larus and Hilfinger 1988; Stransky 1992; Plevyak et al. 1993]. Previous to this article, 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/typechecking method (whether based on graphs or other principles [Choi et al. 1993; Deutsch 1992; 1994; Ghiya and Hendren 1996; Hendren 1990; Hendren and Nicolau 1990; Landi and Ryder 1991]) has the ability to determine that "circular listness" 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 8; however, a brief characterization of some of the differences is as follows:

—Several previous methods have used allocation sites to name shape-nodes [Chase et al. 1990; Jones and Muchnick 1982; Plevyak et al. 1993]. 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*. A shape-node named with variable-set X represents run-time locations that are simultaneously pointed to by all (and only) the variables in X.

—Like other shape-analysis methods, our method clusters collections of run-time locations into *summary nodes*. In our approach, run-time locations that are not pointed to by variables are clustered into a single summary node. [Chase et al. 1990, p.309] observed that their shape-analysis method cannot handle programs such as the list-reversal program because it lacks a way to materialize ("un-summarize") summary nodes at certain key points of the analysis. Our method materializes copies of the summary node (as nonsummary 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 [Stransky 1992] or use some heuristics to remove such edges under limited conditions [Chase et al. 1990; Jones and Muchnick 1981; Larus and Hilfinger 1988; Plevyak et al. 1993]. (This unusual characteristic of our method is also a by-product of the node-naming scheme.)

—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 represent cons-cells that may be the target of more than one pointer from fields of cons-cells.[1] (Sharing through variables—e.g., when two variables point to the same cons-cell—is represented directly by shape-graph edges.)

When an unshared list is traversed, say via a loop containing an assignment $x := x.cdr$, the sharing information is used to improve the precision of the ma-

[1]Throughout the article, the presentation is couched in terms of the Lisp primitives for manipulating heap-allocated storage (i.e., **nil**, **cons**, *car*, and *cdr*). However, this is not due to any basic limitation of the work; the algorithm extends readily to the case of pointers to user-defined types that have more than two fields. We assume that 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. (The treatment of procedures is discussed in Section 6.4.) The language provides atomic data (e.g., integer, real, boolean, etc.) and constructor and selector operations (e.g., **nil**, **cons**, *car*, and *cdr*), 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.

terialization operation, which allows the algorithm to determine that x points to an unshared list on every iteration. The limited way in which sharing information is utilized in Chase et al.[1990] and Jones and Muchnick [1981] prevents the methods described in those articles from determining this fact.

—The shape-node names also provide information that sometimes permits the 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 [1990] is sometimes able to determine that a shared node becomes unshared. However, Hendren's method does not handle data structures that contain cycles.

An experimental implementation of the shape-analysis method has been created. The examples presented in this article have been prepared with the aid of this implementation.

The remainder of the article is organized as follows. Section 2 provides an overview of the shape-analysis technique. Section 3 introduces the terminology and notation used in the rest of the article. Section 4 presents a concrete collecting semantics for a language with destructive updating, in terms of "shape-graphs" that represent memory states. Section 5 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 6 presents some elaborations and extensions of our basic approach. Section 7 concerns applications of the shape-analysis method. Section 8 discusses related work. A proof of correctness, showing that the abstract semantics of static shape-graphs is safe with respect to the concrete semantics, is presented in the electronic appendix.

2. AN OVERVIEW OF THE METHOD

The shape-analysis algorithm is presented and proven correct using the framework of abstract interpretation [Cousot and Cousot 1977]. Because pointers, heap-allocated storage, and destructive updating are all mechanisms that introduce aliasing, the formal treatment of shape analysis is notationally somewhat formidable. However, many aspects of the shape-analysis algorithm can be understood at an intuitive level. In this section, we give such an overview of the algorithm, using a program that performs a list reversal via destructive updating as a running example.

The list-reversal program is shown in Figure 1. Assuming that variable x initially points to an unshared list (i.e., a possibly empty, acyclic, singly linked list with no shared cons-cells), after each iteration, y points to the reversal of a successively longer prefix of the original list. The shape-analysis algorithm detects (among other things) that at the beginning of each iteration of the loop the following properties hold:

—*Invariant (1).* Variable x points to an unshared, acyclic, singly linked list.

—*Invariant (2).* Variable y points to an unshared, acyclic, singly linked list, and variable t may point to the second element of the y-list (if such an element exists).

—*Invariant (3).* The lists pointed to by x and y are disjoint.

```
program reverse(x, y)
begin
   /* x points to an unshared, acyclic, singly linked list */
   y := nil
   while x ≠ nil do
      t := y
      y := x
      x := x.cdr
      y.cdr := t
   od
   t := nil
end
```

Fig. 1. A program to reverse a list.

2.1 Static Shape Graphs

The shape-analysis algorithm is based on an abstraction of memory, called a *static shape graph* (SSG). An SSG is a finite, labeled, directed graph that approximates the actual (or "concrete") stores that can arise during program execution. The shape-analysis algorithm itself is an iterative procedure that computes an SSG at every program point.

In contrast to concrete stores, each SSG in a program is *a priori* of **bounded size**. This is achieved by using a single shape-node to represent multiple cons-cells. In general, a shape-node in an SSG has the following properties:

(a) A shape-node n_Z, where $Z \neq \phi$, represents a *unique* cons-cell in any given concrete store: the cons-cell pointed to by exactly the variables in Z. However, across the collection of SSGs that are the abstractions of the (several different) concrete stores that arise on different loop iterations (or during entirely different executions of the program), n_Z will, in general, denote *different* cons-cells. For example, column two of Figure 2 shows the concrete stores that arise at the beginning of the loop in the list-reversal program when input-list x is a five-element list; column three shows the corresponding SSGs. Shape-node $n_{\{x\}}$ represents the cons-cells l_1, l_2, l_3, l_4, and l_5 in the concrete stores that arise on iterations 0, 1, 2, 3, and 4, respectively.
(b) In contrast, shape-node n_ϕ can represent *multiple cons-cells of a* **single** *concrete store*. (Jocularly, we refer to n_ϕ as the "primordial soup.") For example, in the SSG in column three of the iteration-0 row, n_ϕ represents the cons-cells l_2, l_3, l_4, and l_5 of the concrete store in column two. In the SSG in column three of the iteration-5 row, n_ϕ represents the cons-cells l_3, l_2, and l_1.
(c) In different SSGs, the same cons-cell may be represented by *different* shape-nodes. For instance, consider the SSGs in column three of Figure 2 in top-to-bottom order. Cons-cell l_1 is represented by shape-nodes $n_{\{x\}}$, $n_{\{y\}}$, $n_{\{t\}}$, n_ϕ, n_ϕ, and n_ϕ; cons-cell l_3 is represented by n_ϕ, n_ϕ, $n_{\{x\}}$, $n_{\{y\}}$, $n_{\{t\}}$, and n_ϕ; cons-cell l_5 is represented by n_ϕ, n_ϕ, n_ϕ, n_ϕ, $n_{\{x\}}$, and $n_{\{y\}}$.

There is an important conclusion to draw from these properties: it is *incorrect* to think of a shape-node as representing a fixed partition of memory. Instead, the

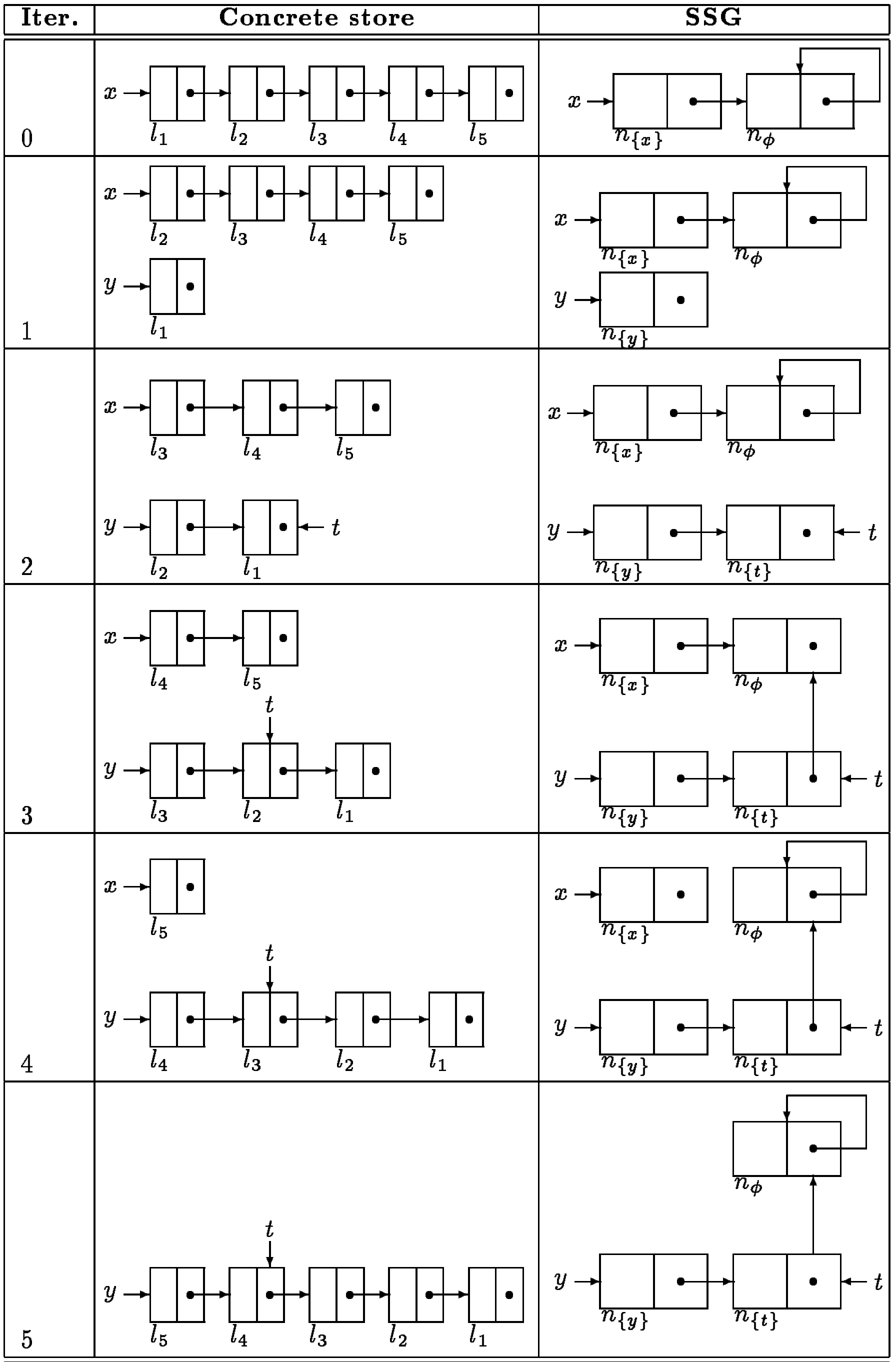


Fig. 2. The concrete stores and the corresponding SSGs that arise at the beginning of the loop in the list-reversal program when input-list x is a five-element list. For each of the shape-nodes in all of the SSGs, the value of $is^\sharp$ is *false*.

ideas to keep in mind are the following:

> By going from stores to SSGs, we deliberately drop information about the concrete locations, but we keep "aliasing-configuration" information that characterizes cons-cells that are simultaneously pointed to by different sets of variables. A shape-node n_X (i.e., with variable-set X) in the shape-graph for program-point p represents the cons-cells that are simultaneously pointed to by all (and only) the variables in X when control reaches p.

(The sets of variable names that represent alias configurations are reminiscent of the alias-configurations tracked by Myers in his algorithm for determining aliasing among (scalar) program variables [Myers 1981].)

2.2 An Explicit Representation of Sharing

An examination of the iteration-0 row of Figure 2 may lead the reader to think that acyclic structures are abstracted to cyclic structures (and hence that the abstraction cannot distinguish between cyclic and acyclic structures). In fact, although SSGs are "cyclic"—in the sense that there are nodes with paths to themselves—there is an additional component of SSGs that distinguishes the abstractions of cyclic concrete structures from the abstractions of acyclic concrete structures. In particular, each shape-node n in an SSG has an associated Boolean flag, denoted by $is^\sharp(n)$, that, when *true*, indicates that the cons-cells represented by n may be the target of pointers emanating from two or more distinct cons-cell fields ("$is^\sharp$" stands for "is-shared"). Edges from variables do *not* contribute to $is^\sharp$ status: $is^\sharp$ captures a notion of "heap shared"; sharing through variables is represented explicitly by edges from variables to shape-nodes.

The SSG in column three of the iteration-0 row of Figure 2 illustrates an important aspect of the abstraction from concrete stores to SSGs: because all nodes in the tail of an acyclic list are represented by summary-node n_ϕ, the abstraction of a list (deliberately) loses information about the length of the list. In this way, we achieve a bounded-size abstract representation and hence a terminating abstract semantics.

The significance of $is^\sharp(n) = false$ is that if several *car* and *cdr* edges in an SSG point to n, they represent concrete edges that never point to the same cons-cell in any concrete store that the SSG represents. For example, in the SSG in column three of the iteration-0 row of Figure 2, because $is^\sharp(n_\phi) = \mathit{false}$, the two *cdr* edges from $n_{\{x\}}$ to n_ϕ and from n_ϕ to n_ϕ cannot represent edges that point to the same cons-cell in any concrete store that this SSG represents. Thus, despite the fact that this SSG contains a cycle, it only represents acyclic concrete stores.

An examination of the iteration-3 row of Figure 2 may lead the reader to think that disjoint lists are abstracted to shared lists (and hence that the abstraction cannot distinguish between disjoint lists and shared lists). However, the $is^\sharp$ values come to the rescue here, too. Because $is^\sharp(n_\phi) = \mathit{false}$, we know that the two *cdr* edges from $n_{\{x\}}$ to n_ϕ and from $n_{\{t\}}$ to n_ϕ cannot point to the same cons-cell in any concrete store. Consequently, the abstraction captures the fact that at the beginning of iteration 3, the lists pointed to by x and t are disjoint. Thus, despite the fact that the tails of the x-list and the t-list are both represented by n_ϕ, the

SSG only represents concrete stores in which the x-list and the t-list do not share any cons-cells in common.

2.3 An Iterative Algorithm

The shape-analysis method is an iterative algorithm that computes an SSG for every point in the program. The algorithm operates over the domain of SSGs, with each statement in the program having an associated SSG-to-SSG transformer. The shape-analysis algorithm is conservative with respect to the collection of stores that can actually arise during any execution:

—The SSG computed for a program point by the algorithm may have more shape-nodes and edges than the SSG obtained by abstracting the collection of stores that can actually arise during execution.

—The SSG for a program point p might have $is^\sharp(n) = \mathit{true}$ even though, in the concrete stores that arise at p, none of the cons-cells that n represents are the target of pointers emanating from two or more distinct cons-cell fields.

For the reverse program from Figure 1, the shape-analysis algorithm uses four iterations over the program to compute the final SSGs. The SSGs that arise at each program point during the analysis are shown in Figure 3. These will be used below, in Section 2.4, to explain how the algorithm is able to establish information about the possible "shapes" that heap-allocated structures in a program can take on. (For space reasons, only the SSGs for statements in the loop body are shown. In all of these SSGs, $is^\sharp$ is *false* for all of the shape-nodes.)

2.4 What the Shape-Analysis Algorithm Achieves and Why

We now consider the main reasons why the shape-analysis algorithm is able to produce accurate information about the list-reversal program. In particular, we wish to give a feeling for why the algorithm is able to establish the three invariants mentioned at the beginning of Section 2.

There are three key aspects of the algorithm that contribute to the successful outcome of the analysis of the list-reversal program. Each of them can be illustrated by the SSG transformations carried out by the algorithm during its third iteration over the program (see the iteration-3 column of Figure 3.)

2.4.1 *Tracking of Aliasing Configurations.* One aspect involves the tracking of aliasing configurations via the "names" attached to shape-nodes. This is illustrated by the SSG transformation carried out at the statements "$t := y$" and "$y := x$" during the third iteration. In particular, when the statement "$t := y$" is analyzed—producing the second SSG in column 3 of Figure 3 from the first SSG. There are two issues: (1) the "liquidization" of $n_{\{t\}}$ and (2) the "renaming" of $n_{\{y\}}$.

When "$t := y$" is encountered in the third iteration, t points to $n_{\{t\}}$, which is also pointed to by $n_{\{y\}}.cdr$. This represents a concrete store in which y is the only variable pointing to a cons-cell l_y, and $l_y.cdr$ points to a cons-cell l_t, which is pointed to by t (and no other variable). After the assignment "$t := y$", l_t is not pointed to by any variable, and both t and y point to l_y. The appropriate SSG to represent this store is obtained by "liquidizing" $n_{\{t\}}$: to model the fact that variable t no longer points to l_t, we remove t from the "name" of $n_{\{t\}}$; because

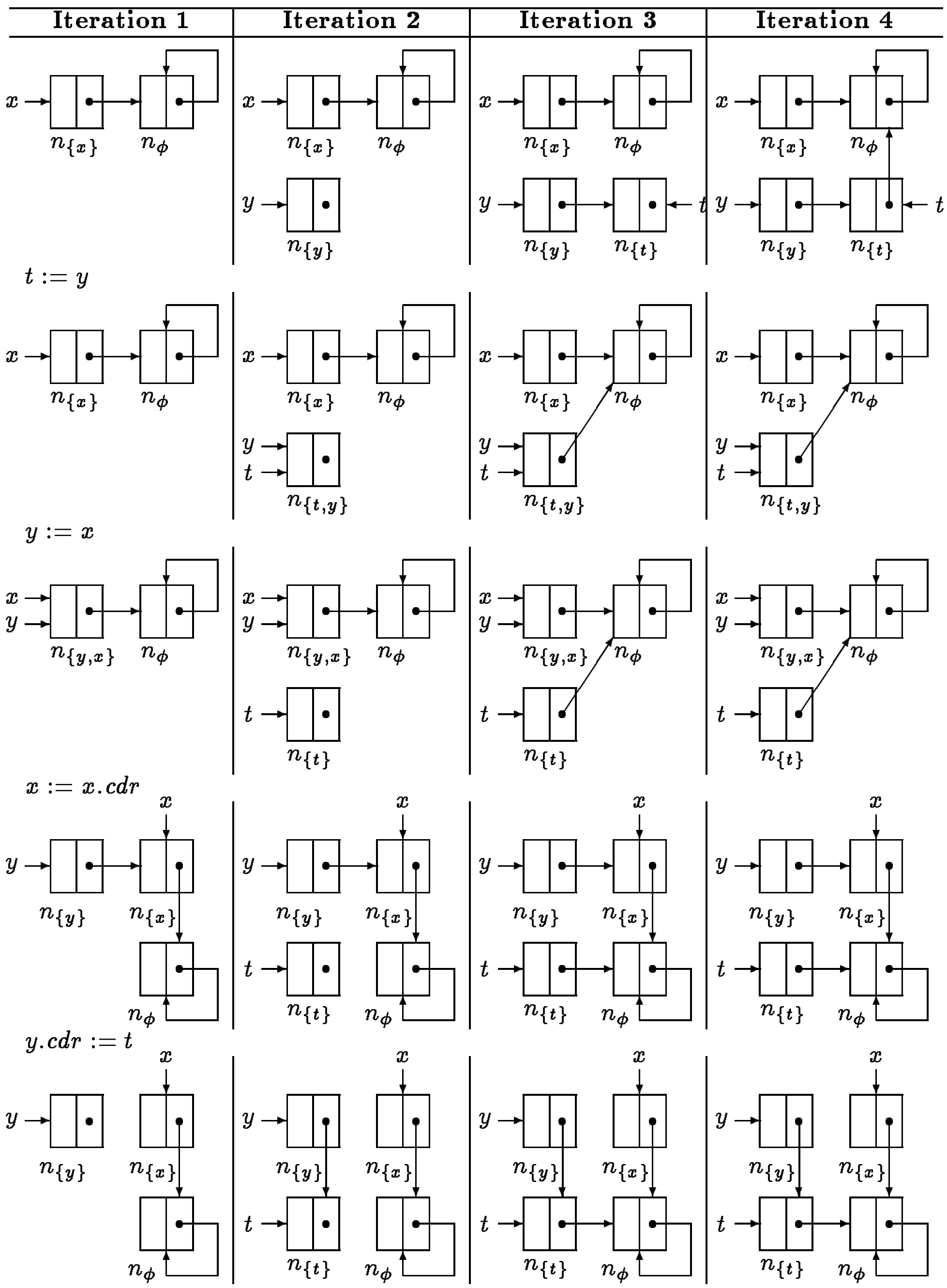


Fig. 3. The SSGs that arise when the shape-analysis algorithm is applied to the list-reversal program.

this turns the "name" of the shape-node into ϕ, it is merged with shape-node n_ϕ, which already has the "name" ϕ. (By this, $n_{\{t\}}$ is "liquidized" and falls into the primordial soup.)

In addition, to model the fact that variable t now points to l_y, we add t to the "name" of $n_{\{y\}}$, renaming $n_{\{y\}}$ to $n_{\{t,y\}}$. Therefore, after statement "$t := y$" there is a *cdr* edge from $n_{\{t,y\}}$ to n_ϕ (and there is no longer any shape-node known as $n_{\{y\}}$).

Executing the statement "$t := y$" cannot increase the amount of "sharing" in any concrete store. That is, it cannot increase the number of cons-cells that are the target of pointers emanating from two or more distinct cons-cell fields. This is reflected in the SSG by the fact that $is^\sharp$ is still *false* for all shape-nodes in the new SSG. Thus, despite the fact that the tails of the x-list and y-list are both represented by n_ϕ, the SSG captures the fact that the lists pointed to by y and x remain disjoint. (See Invariant (3) and the discussion in Section 2.2)

Similarly, when the statement "$y := x$" is processed (to produce the third SSG from the second SSG), $n_{\{t,y\}}$ is renamed to $n_{\{t\}}$, and $n_{\{x\}}$ is renamed to $n_{\{y,x\}}$. (The use of sets of variables to name the nodes in SSGs can result in an exponential number of shape-nodes. Techniques to sidestep this problem are discussed in Section 6.2.)

2.4.2 *"Materialization" of $n_{\{x\}}$ from n_ϕ.* Equally important is the way the algorithm handles the advancement of x down the x-list by "$x := x.cdr$" (to produce the fourth SSG from the third SSG).

When "$x := x.cdr$" is encountered in the third iteration, x and y point to shape-node $n_{\{y,x\}}$, and the *cdr* field of $n_{\{y,x\}}$ points to summary-node n_ϕ. Because $is^\sharp(n_\phi) = \mathit{false}$, this represents a concrete store in which x and y point to a cons-cell l_0, and the *cdr* field of l_0 points to an unshared cons-cell l_1, which may, in turn, point to an unshared, acyclic, singly linked list (made up of l_2, l_3, etc.). After the assignment, only y points to l_0; x points to l_1 (and l_2, l_3, etc. are still not pointed to by any variable). The appropriate SSG to represent this store has shape-nodes $n_{\{y\}}$, $n_{\{x\}}$, $n_{\{t\}}$, and n_ϕ as shown in the fourth SSG in column 3 of Figure 3.

The effect has been to "materialize" a *new* **nonsummary** *shape-node* $n_{\{x\}}$ from summary-node n_ϕ. (We say that "the operation '$x := x.cdr$' ladles a node out of the primordial soup.")

Materialization creates an SSG that conservatively covers all the possible new configurations of storage. For example, had $is^\sharp(n_\phi)$ been *true* in the third SSG, as it is in column two of Figure 4, then there would have been three additional *cdr* edges: from $n_{\{x\}}$ to $n_{\{x\}}$, from $n_{\{t\}}$ to $n_{\{x\}}$, and from n_ϕ to $n_{\{x\}}$. (When we ladle a node from the primordial soup, and $is^\sharp(n_\phi) = \mathit{true}$, we refer to edges like these as "bits of algae" attached to $n_{\{x\}}$.)

2.4.3 *Cutting the List.* In the analysis of "$y.cdr := t$" (which produces the fifth SSG from the fourth SSG), the *cdr* edge of shape-node $n_{\{y\}}$ (which points to $n_{\{x\}}$ in the fourth SSG) is first removed. This cuts the y list at the head, separating the first element, $n_{\{y\}}$, from the tail, which x points to. A *cdr* edge from $n_{\{y\}}$ to $n_{\{t\}}$ is then added, which concatenates shape-node $n_{\{y\}}$ at the head of the list that t points to.

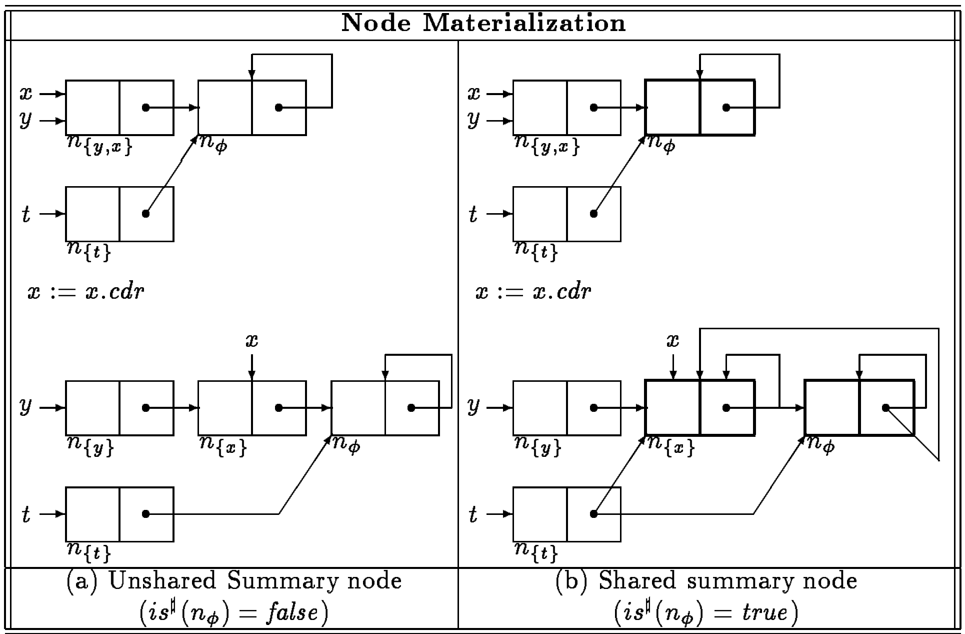


Fig. 4. Materialization of n_t from n_ϕ.

Other shape-analysis algorithms handle a statement of the form "$y.cdr := t$" much more conservatively: they do not, in general, remove the *cdr* edges emanating from the shape-nodes that y points to.[2] Instead, they *retain* the old edges and *add* *cdr* edges from the shape-node that y points to, to the shape-node that t points to.

The reason our shape-analysis algorithm is able to do a better job is because it conservatively tracks all the possible aliasing configurations via the "names" attached to shape-nodes: a shape-node n_Z in the shape-graph for program-point p represents the cons-cells that are simultaneously pointed to by exactly the variables in Z when control reaches p. If y is in the name of shape-node n_Z (i.e., if $y \in Z$), then n_Z represents only concrete cons-cells whose *cdr* field will definitely be overwritten. Therefore, in the interpretation of "$y.cdr := t$," our method can always replace the *cdr* edges of all shape-nodes that y points to by edges to the shape-nodes that t points to. (In the fourth SSG in column 3 of Figure 3, there is only a single shape-node that y points to (namely $n_{\{y\}}$), and a single shape-node that t points to (namely $n_{\{t\}}$).)

In the shape analysis of the list-reversal program, there is a crucial interaction between these three aspects. Suppose, for example, that in the SSG transformation for "$x := x.cdr$," shape-node $n_{\{x\}}$ was not materialized out of n_ϕ, but instead variable x was merely set to point to n_ϕ. (This is essentially what other shape-analysis algorithms do, but expressed in our terminology.) At "$y.cdr := t$," the removal of y's *cdr* edge would still cut the y-list at the head, separating the node

[2] In some algorithms, *cdr* edges emanating from a shape-node that y points to are removed in very limited circumstances.

that y points to (i.e., $n_{\{y\}}$) from the list pointed to by x (which in this case would be represented by n_ϕ.). However, when the *cdr* edge from $n_{\{y\}}$ to $n_{\{t\}}$ is added to the SSG, this sets y's *cdr* field to t, whose *cdr* field points to n_ϕ, which is what x points to. At this stage, the information that the x-list and the y-list are disjoint has been lost!

Note how differently things turn out when $n_{\{x\}}$ is materialized from n_ϕ at "$x := x.cdr$." At "$y.cdr := t$," x points to $n_{\{x\}}$, and thus when y's *cdr* field is set to $n_{\{t\}}$ (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 $is^\sharp(n_\phi) = \mathit{false}$ we know that the two lists do not share any cons-cells in common; that is, x and y must point to disjoint acyclic lists.

The operations discussed above—assigning a pointer to a pointer, advancing a pointer down a list, and cutting a list—are three of the five main operations of list-manipulation algorithms. The fourth and fifth 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 Section 5.5.) This points up the strength of our approach: our algorithm handles all five of the basic list-manipulation operations with a remarkable degree of precision, as well as similar tree- and circular-list-manipulation operations.

3. TERMINOLOGY AND NOTATION

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.

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 $PVar$.

3.1 Normalization Assumptions

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

—Only one constructor or selector is applied per assignment statement.

—An expression $\mathbf{cons}(x, y)$ is executed in three steps: (1) an uninitialized **cons** cell is allocated, and its address is assigned into a new temporary variable (e.g., "$temp := \mathbf{new}$"); (2) the *car* component of *temp* is initialized with the value of x ("$temp.car := x$"); (3) the *cdr* component of *temp* is initialized with the value of y ("$temp.cdr := y$").

—All allocation statements are of the form $x := \mathbf{new}$, (as opposed to $x.sel := \mathbf{new}$).

—In each assignment statement, the same variable does not occur on both the left-hand and right-hand side.

—Each assignment statement of the form $lhs := rhs$ in which $rhs \not\equiv \mathbf{nil}$ is immediately preceded by an assignment statement of the form $lhs := \mathbf{nil}$.

—An assignment statement of the form $temp$:= **nil** is placed at the end of the program for each temporary variable $temp$ introduced as part of normalization.

Thus, for every vertex $v \in V$ in which a pointer manipulation is performed, $st(v)$ has one of the following forms: $x := \mathbf{nil}$, $x.sel := \mathbf{nil}$, $x := \mathbf{new}$, $x := y$, $x := y.sel$, 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 next-to-last assumption allows the semantics to treat the "kill" aspects of a statement (e.g., $x := \mathbf{nil}$) separately from the "gen" aspects (e.g., $x := y.sel$, assuming that x's value is **nil**). (See Figures 6 , 8, and 9)

Example 3.1.1. Figure 5 shows (a) the normalized version of the list-reversal program and (b) the control-flow graph of the program in normalized form.

3.2 Shape-Graphs

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

Definition 3.2.1. A **shape-graph** is a finite directed graph that consists of two kinds of nodes—**variables** (i.e., $PVar$) and **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 denoted by a pair of the form $[x, n]$, where $x \in PVar$ and n is a shape-node.

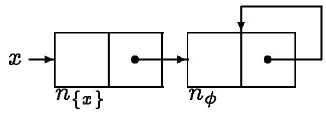
—E_s is the graph's set of selector-edges, each of which is denoted by a triple of the form $\langle s, sel, t\rangle$, where s and t are shape-nodes, and $sel \in \{car, cdr\}$.

We overload the symbol E_v to also mean the function that, when applied to a variable x, returns x'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 $\mathcal{SG}$.

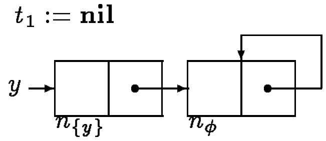
Note that for a given shape-graph SG, $shape\_nodes(SG)$ is uniquely defined: it consists of the set of nonisolated nodes in SG (i.e., the nodes that are touched by at least one edge). It is for this reason that we do not explicitly list the node set when specifying a shape-graph. (For the sake of simplicity, we choose not to work with a definition of $shape\_nodes(SG)$ that removes unreachable nodes.)

Remark. We will systematically use the terms "nodes" and "edges" when referring to elements of shape-graphs, and "vertices" and "arcs" when referring to

program reverse(x, y)
begin
/* x points to an unshared, acyclic, singly linked list */

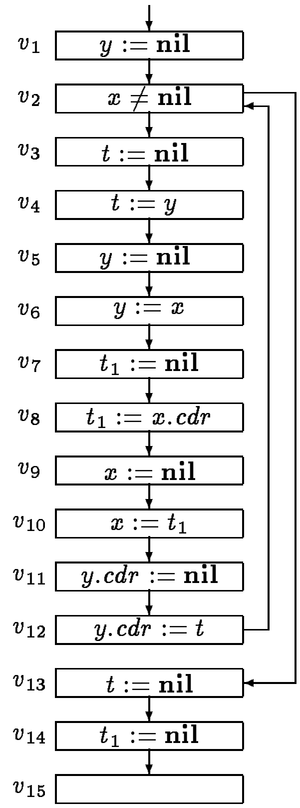


$y :=$ **nil**
while $x \neq$ **nil do**
  $t :=$ **nil**
  $t := y$
  $y :=$ **nil**
  $y := x$
  $t_1 :=$ **nil**
  $t_1 := x.cdr$
  $x :=$ **nil**
  $x := t_1$
  $y.cdr :=$ **nil**
  $y.cdr := t$
od
$t :=$ **nil**
$t_1 :=$ **nil**



end

(a)



(b)

Fig. 5. The list-reversal program in normalized form and the normalized program's control-flow graph.

elements of control-flow graphs. In general, properties of (or operations on) the shape-graphs used to define the abstract semantics will be superscripted with $\sharp$; those used to define the concrete semantics have no superscript.

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 3.2.1) helps in defining the abstraction relation that relates them (see Definition 5.2.1).

In the concrete semantics, which is given in Section 4, 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 cons-cell, 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 cons-cells pointed to by the *car* and

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

Fig. 6. The concrete semantics $[\![st]\!]: \mathcal{DSG} \to \mathcal{DSG}$. 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 cons-cell).

cdr fields of n, are also either singleton sets or empty (depending on whether these fields point to allocated cons-cells or not). Such properties are captured in the following definition:

Definition 3.2.2. *(Deterministic Shape-Graphs).* A shape-graph is **deterministic** if (1) for every $x \in PVar$, $|E_v(x)| \leq 1$, and (2) 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 $\mathcal{DSG}$.

The concrete semantics will treat statements as "deterministic-shape-graph transformers." In contrast, in Section 5.3, the abstract semantics will use *nondeterministic* shape-graphs to model (conservatively) the state of heap-allocated storage. In nondeterministic shape-graphs, quantities such as $E_v(x)$, $E_s(n, car)$, and $E_s(n, cdr)$ may each yield a set with more than one shape-node.

4. THE CONCRETE SEMANTICS

In this section, we present a concrete semantics in which deterministic shape-graphs are used to represent the state of memory and in which 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 concrete stores with each point in the program.

Figure 6 contains the semantic equations of the concrete semantics. The meaning of a statement *st* is a function $[\![st]\!]: \mathcal{DSG} \to \mathcal{DSG}$. (When examining the last four equations in Figure 6, bear in mind that, because of the Normalization Assumptions of Section 3.1, before each of the statements executes it is known that the value of the left-hand side is **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 6.) The DSG transformers listed in Figure 6 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 in the following ways:

—The only parts of the store that the concrete semantics keeps track of are the pointer variables and the cons-cells of heap-allocated storage.

—Rather than causing an "abnormal termination" of the program, dereferences of nil pointers and uninitialized pointers are treated as no-ops.[3]

—The concrete semantics 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 these assumptions is that the collecting semantics may associate a control-flow-graph vertex with more concrete stores (i.e., DSGs) than would be the case were we to start with a conventional concrete semantics. (These assumptions are patently safe, so we will not take the space here to justify them further.)

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 4.1. The **collecting semantics** $cs\colon V \to 2^{\mathcal{DSG}}$ is defined as follows:

$$cs(v) \stackrel{\text{def}}{=} \{\, [\![st(v_k)]\!](\cdots([\![st(v_1)]\!](\langle\phi,\phi\rangle))) \mid [v_1,\ldots,v_k] \in pathsTo(v) \,\}$$

The value of $cs(v)$ represents (a superset of) the concrete stores that could arise just before vertex v is executed.

Equationally, the collecting semantics can be defined as the least fixed point (under set inclusion) of the following system of equations in CS_v, for $v \in V$:

$$CS_v = \begin{cases} \{\langle\phi,\phi\rangle\} & \text{if } v = start \\ \{[\![st(u)]\!](SG) \mid \langle u,v\rangle \in A, SG \in CS_u\} & \text{otherwise} \end{cases} \tag{1}$$

5. THE ABSTRACT SEMANTICS

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

Static shape-graphs (or *SSGs*, for short) are defined in Section 5.1; the abstraction function is defined in Section 5.2; and the abstract semantics is given in Section 5.3. The reverse program is used as a running example in these sections. Section 5.4 discusses an interesting aspect of how the abstract semantics treats statements of the form "$x.sel$:= **nil**." Section 5.5 considers a second example program—a list-insertion program that may insert a cons-cell at an arbitrary point in a linked list—and shows that the shape-analysis method is capable of determining that when the argument is an unshared acyclic list, the result is also an unshared acyclic list.

5.1 Static Shape-Graphs

Unlike the concrete stores of the collecting semantics (i.e., DSGs), the SSGs of the abstract semantics are nondeterministic: $E_v(x)$, $E_s(n, car)$, and $E_s(n, cdr)$ may

[3] An alternative semantics that returns a special value $\perp$ if a nil pointer or uninitialized pointer is dereferenced was used in Sagiv et al. [1995]. The present formulation has the advantage of being simpler.

each yield a set with more than one shape-node. In addition, and again in contrast with DSGs, the SSGs for a given program are a priori 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 infinitely many concrete cons-cells (from an infinite set of finite DSGs) into a single SSG shape-node.

Definition 5.1.1. A **static shape-graph** is a pair $\langle SG^\sharp, is^\sharp \rangle$, where

—$SG^\sharp$ is a shape-graph,
—the set $\mathit{shape\_nodes}(SG^\sharp)$ is a subset of $\{n_X \mid X \subseteq PVar\}$, and
—$is^\sharp$ is a function of type $\mathit{shape\_nodes}(SG^\sharp) \to \{\mathit{false}, \mathit{true}\}$.

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

In the following definition, we impose an order on SSGs where $SG_1^\sharp \sqsubseteq SG_2^\sharp$ if $SG_2^\sharp$ contains at least the edges of $SG_1^\sharp$:

Definition 5.1.2. Let $SG_1^\sharp = \langle \langle E_{v1}^\sharp, E_{s1}^\sharp \rangle, is_1^\sharp \rangle$ and $SG_2^\sharp = \langle \langle E_{v2}^\sharp, E_{s2}^\sharp \rangle, is_2^\sharp \rangle$. We define the following ordering on $\mathcal{SSG}$: $SG_1^\sharp \sqsubseteq SG_2^\sharp$ if and only if all of the following conditions hold:

—$E_{v1}^\sharp \subseteq E_{v2}^\sharp$
—$E_{s1}^\sharp \subseteq E_{s2}^\sharp$
—For every $n \in \mathit{shape\_nodes}(SG^\sharp)$, $is_1^\sharp(n) \Rightarrow is_2^\sharp(n)$.

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

$$SG_1^\sharp \sqcup SG_2^\sharp \stackrel{\text{def}}{=} \langle \langle E_{v1}^\sharp \cup E_{v2}^\sharp, E_{s1}^\sharp \cup E_{s2}^\sharp \rangle, is_1^\sharp \vee is_2^\sharp \rangle.$$

5.2 The Abstraction Function

Our task in this section is to define the abstraction function that relates the domains $2^{\mathcal{DSG}}$ and $\mathcal{SSG}$. The abstraction function α is defined in Definition 5.2.1; α is defined in terms of the auxiliary functions π, which establishes the relationship between the nodes of a DSG and their corresponding nodes in the SSG, and β, which is an overloaded symbol denoting a family of functions defined inductively on the structure of elements that make up a DSG.

Definition 5.2.1. *(The Abstraction Function).* Let $SG = \langle E_v, E_s \rangle$ be a shape-graph in $\mathcal{DSG}$, and let l, l_1, and l_2 be shape-nodes in $\mathit{shape\_nodes}(SG)$. The function $\pi[E_v](l)$, from $\mathit{shape\_nodes}(SG)$ to 2^{PVar}, identifies the set of variables that point to a given cons-cell l. It is defined as follows:

$$\pi[E_v](l) \stackrel{\text{def}}{=} \{x \in PVar \mid [x, l] \in E_v\}.$$

(When E_v is understood, we will write $\pi[E_v](l)$ as $\pi(l)$.)

The function $iis[E_s](l)$, from $\mathit{shape\_nodes}(SG)$ to $\{\mathit{false}, \mathit{true}\}$, checks whether a cons-cell l is the target of pointers emanating from two or more distinct cons-cell fields ("*iis*" stands for "induced-is-shared"). It is defined as follows:

$$iis[E_s](l) \stackrel{\text{def}}{=} |\{\langle *, *, l \rangle \in E_s\}| \geq 2.$$

(When E_s is understood, we will write $iis[E_s](l)$ as $iis(l)$.)

The collection of functions $\beta[E_v]$ (abbreviated as β in all but the last case below) is defined as follows:

$$\begin{aligned}
\beta(l) &\stackrel{\text{def}}{=} n_{\pi[E_v](l)}\\
\beta([x,l]) &\stackrel{\text{def}}{=} [x, \beta(l)]\\
\beta(E_v) &\stackrel{\text{def}}{=} \{\beta([x,l]) \mid [x,l] \in E_v\}\\
\beta(\langle l_1, sel, l_2\rangle) &\stackrel{\text{def}}{=} \langle \beta(l_1), sel, \beta(l_2)\rangle\\
\beta(E_s) &\stackrel{\text{def}}{=} \{\beta(\langle l_1, sel, l_2\rangle) \mid \langle l_1, sel, l_2\rangle \in E_s\}\\
\beta(\langle E_v, E_s\rangle) &\stackrel{\text{def}}{=} \langle\langle \beta[E_v](E_v), \beta[E_v](E_s)\rangle,\ \lambda n. \bigvee_{\{l \mid \beta[E_v](l)=n\}} iis[E_s](l)\ \rangle
\end{aligned}$$

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

$$\alpha(S) \stackrel{\text{def}}{=} \bigsqcup_{\langle E_v, E_s\rangle \in S} \beta[E_v](\langle E_v, E_s\rangle).$$

The core components of Definition 5.2.1 are the operations $\pi[E_v](l)$ and $\beta(l) = n_{\pi[E_v](l)}$. The function $\pi[E_v](l)$ establishes the relationship between a DSG shape-node l and the name of the SSG shape-node that represents l. For example, consider the iteration-0 row of Figure 2. In column two, DSG node l_1 is pointed to by variable x and is mapped by β to SSG node $\beta(l_1) = n_{\pi[E_v](l_1)} = n_{\{x\}}$ (see column three). DSG nodes l_2, l_3, l_4, and l_5, which are not pointed to (directly) by any variables, are mapped by β to SSG node n_ϕ. In general, $\pi[E_v]$ generates a finite set of SSG node names from the a priori unbounded number of DSG nodes in the DSGs in S. Auxiliary function β then collapses SG onto the smaller set of nodes, while preserving many aspects of SG's structure. We say that an SSG shape-node n **represents** a DSG shape-node l in $SG = \langle E_v, E_s\rangle$ if $\beta[E_v](l) = n$.

The function $iis[E_s](l)$ checks whether a cons-cell l is the target of pointers emanating from two or more distinct cons-cell fields. Because of the indexed-or performed with respect to the set of cons-cells that β maps to n_X, $is^\sharp(n_X)$ is *true* if any of the cons-cells in SG that n_X represents is the target of pointers emanating from two or more distinct cons-cell fields in SG. (This aspect of β and $\pi[E_v]$ is not illustrated by the SSGs that appear in Figure 2.) On the other hand, if β sets $is^\sharp(n_X)$ to *false*, this means that the cons-cell (or cells) that n_X represents all have at most one predecessor. For example, consider the iteration-0 row of Figure 2. In the SSG in column three, n_ϕ represents the cons-cells l_2, l_3, l_4, and l_5, each of which has exactly one predecessor in the DSG shown in column two. Consequently, $is^\sharp(n_\phi) = \mathit{false}$.

In Section 5.3 and the electronic appendix, it is convenient to work with an alternative, but equivalent, definition of $iis[E_s]$:

Definition 5.2.2.

$$\begin{aligned}
iis[E_s](l) \stackrel{\text{def}}{=}\ & \exists l_1, l_2, \exists sel_1, sel_2 :\\
& \langle l_1, sel_1, l\rangle, \langle l_2, sel_2, l\rangle \in E_s \wedge (l_1 \neq l_2 \vee sel_1 \neq sel_2)
\end{aligned}$$

It may not be immediately apparent why sharing information is represented explicitly in SSGs. The reason is that only very conservative information about sharing could be inferred directly from an SSG if there was no explicit $is^\sharp$ function. For example, without the $is^\sharp$ function, it would not be possible to distinguish acyclic lists from cyclic lists. Suppose that $SG^\sharp$ is an SSG that arises from an application of abstraction-function α to DSG SG, and suppose that n is a shape-node of $SG^\sharp$ that represents a shared shape-node l of SG. Not only is $is^\sharp(n) = true$, but one or more of the following conditions must hold in $SG^\sharp$:

—There exists a selector-edge from n_ϕ to n. This reflects the fact that n_ϕ can represent multiple cons-cells, and thus the single SSG edge $\langle n_\phi, sel, n\rangle$ can represent two or more selector-edges in SG.

—There exist two selector-edges to n from different SSG shape-nodes, say, n_{Z_1} and n_{Z_2}, where $Z_1 \cap Z_2 = \phi$. In this case, the two SSG edges represent two different selector-edges to l in SG—one from the cons-cell pointed to by the set of variables Z_1 and one from the cons-cell pointed to by the set of variables Z_2.

—There exist two selector-edges to n, with *different* selectors, from a single shape-node in $SG^\sharp$. In this case, the two SSG edges represent two different selector-edges in SG.

The reason why explicit sharing information is maintained in SSGs is that the converse of the above observation need not hold. For example, in the SSG in column three of the iteration-0 row of Figure 2, there exist two selector-edges to n_ϕ from different shape-nodes, n_ϕ (i.e., n_{Z_1}) and $n_{\{x\}}$ (i.e., n_{Z_2}) such that $Z_1 \cap Z_2 = \phi \cap \{x\} = \phi$. However, the value of $is^\sharp(n_\phi)$ is *false*. (The fact that $is^\sharp(n_\phi) = false$ is what indicates that n_ϕ's incoming edges represent edges that never point to the same cons-cell in any concrete store.)

As defined in Definition 5.2.1, the abstraction of a set of DSGs S results in a single SSG, $\alpha(S)$. Even though abstraction function α combines information from several applications of β, we can sometimes recognize that particular pairs of elements in an SSG represent features that can only come from *different* DSGs. In particular, a shape-node n_Z represents cons-cells that are simultaneously pointed to by all (and only) the variables in Z. In a given DSG, each program variable points to at most one cons-cell. Therefore, two different shape-nodes n_X and n_Y, such that $X \neq Y$ and $X \cap Y \neq \phi$, represent *incompatible* configurations of variables: they cannot possibly represent cons-cells that are in the same DSG. This means that the following structural property holds for an SSG $\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle$ that arises from an application of abstraction function α:

Compatibility of Edge End-Points: For all $\langle n_X, sel, n_Y\rangle \in E_s^\sharp$, either $X = Y$ or $X \cap Y = \phi$.

Example 5.2.3. For example, in the SSG at the top of column two in Figure 11, the selector-edge $\langle n_{\{y\}}, cdr, n_{\{t\}}\rangle$ satisfies $\{y\} \cap \{t\} = \phi$. This SSG could not contain any of the following selector-edges: $\langle n_{\{x\}}, car, n_{\{x,t_1\}}\rangle$, $\langle n_{\{x\}}, cdr, n_{\{x,t_1\}}\rangle$, $\langle n_{\{x,t_1\}}, car, n_{\{x\}}\rangle$, and $\langle n_{\{x,t_1\}}, cdr, n_{\{x\}}\rangle$.

The SSG transformers of the abstract semantics make use of several properties similar to the "compatibility-of-edge-end-points" property to determine that certain combinations of shape-node elements cannot possibly coexist in the same concrete store (see Figures 10 and 12). This is one of the key reasons why our shape-analysis method is able to carry out accurate "node materialization" on many programs

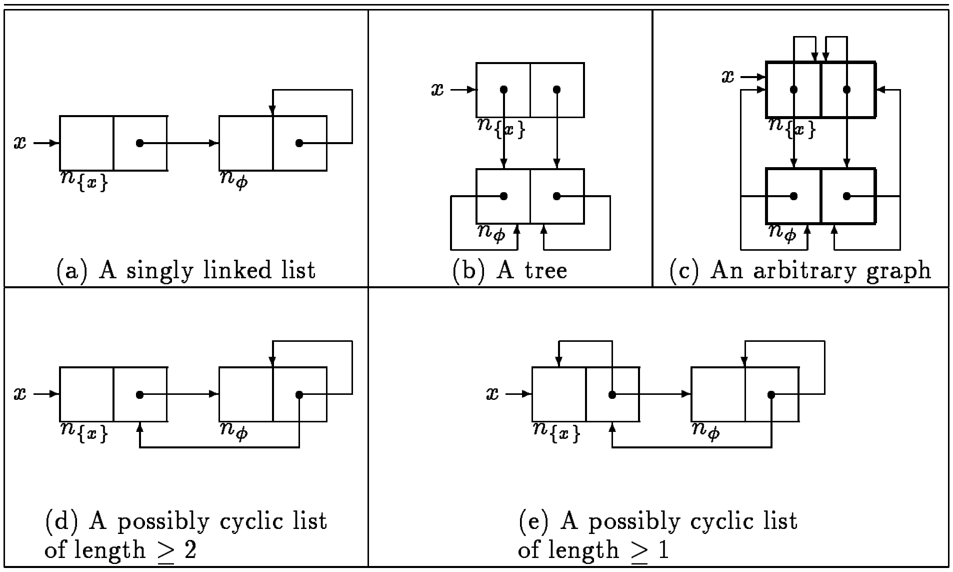


Fig. 7. SSGs that characterize five kinds of data types. For each of the shape-nodes in all of the SSGs but (c), the value of $is^\sharp$ is *false*. In (c), the value of $is^\sharp$ is *true* for both shape-nodes.

(and consequently why it is more precise than competing methods on many of these programs).

Because abstraction function α distributes over $\cup$ (i.e., union of sets of DSGs), the unique concretization function γ such that α and γ form a Galois connection can be defined as follows:

Definition 5.2.4. *(The Concretization Function).* Let $SG^\sharp$ be a shape-graph in $\mathcal{SSG}$. Concretization function $\gamma \colon \mathcal{SSG} \to 2^{\mathcal{DSG}}$ is defined as follows:

$$\gamma(SG^\sharp) \stackrel{\text{def}}{=} \{SG \in \mathcal{DSG} \mid \beta(SG) \sqsubseteq SG^\sharp\}.$$

A *data type* is a collection of DSGs. Definition 5.2.4 provides a way for certain data types, including linked lists, trees, and arbitrary graphs, to be characterized by SSGs:

Definition 5.2.5. Let $SG^\sharp$ be a shape-graph in $\mathcal{SSG}$. $SG^\sharp$ **characterizes** the data type $\gamma(SG^\sharp)$.

Example 5.2.6. Figures 7(a)–(e) show the shape-graphs that characterize five kinds of data types.

The shape-analysis algorithm is conservative with respect to the concrete semantics, and thus the shape-graphs produced may have superfluous edges. Therefore, when the shape-analysis algorithm reports that a variable points to a circular list, it may actually point to a noncircular list; however, when the algorithm reports that a variable points to a noncircular list, it will never point to a circular list. This kind of conservative approximation is appropriate for use, for example, in parallelizing compilers [Hendren and Gao 1992; Hendren et al. 1992]. (An extension of our basic technique allows SSGs to characterize some kinds of *definitely* circular data types, including definitely circular lists. See Section 6.3.)

Remark. The reader may wonder why we do not use an abstraction function that uses a *set* of SSGs to represent the set of stores that can arise at a control-flow-graph vertex, such as

$$\widetilde{\alpha}(S) \stackrel{\text{def}}{=} \{\beta[E_v](\langle E_v, E_s\rangle) \mid \langle E_v, E_s\rangle \in S\}.$$

Using such an abstraction function would have certain advantages:

—In general, it would lead to a shape-analysis algorithm that is more accurate than the method described in this article.

—It would allow us to give simpler definitions for the transfer functions of the abstract semantics (cf. Figures 8 and 9). In particular, there would be no need to use the compatibility-of-edge-end-points property.

However, our belief is that an approach based on a set of SSGs per control-flow-graph vertex is not likely to be feasible in practice. The number of shape-nodes associated with a single control-flow-graph vertex can grow to be very large—in the worst case, doubly exponential in the number of program variables (i.e., $2^{2^{|PVar|}}$). Using a single SSG per control-flow-graph vertex avoids the space blow-up.[4] In addition, the operations needed by a fixed-point-finding algorithm—join of SSGs, equality of SSGs, and applications of the transfer functions of the abstract semantics—can usually be carried out more efficiently for a method based on one SSG per vertex. For these reasons, we believe that the use of a single SSG per vertex is more likely to provide a practical shape-analysis algorithm and that the additional notational complexity required to define the transformers of the abstract semantics is warranted.

5.3 The Abstract Interpretation

The abstract meaning function $[\![\,]\!]^\sharp : \mathcal{SSG} \to \mathcal{SSG}$ for the pointer-manipulation statements is given in Figures 8 and 9. The operations presented in Figures 8 and 9 manipulate variable-edges, selector-edges, and sharing information, as well as the alias information that is maintained in the shape-node names of SSGs. As we shall see, this meaning function is conservative with respect to the concrete semantics defined in Figure 6 (see Theorems 5.3.6 and 5.3.7).

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 cons-cell (i.e., DSG shape-node). The formal definition of the abstract semantics, given in Figures 8 and 9, uses two basic mechanisms:

[4]The number of shape-nodes in a single SSG is bounded by $2^{|PVar|}$. Although with our shape-analysis algorithm, the number of shape-nodes can actually grow to be this large for some pathological programs, our limited experience to date suggests that this is unlikely to arise in practice. The blow-up problem can also be mitigated by using widening (see Section 6.2).

—In many of the cases, the "names" of SSG shape-nodes are adjusted by performing operations on the variable sets that "name" SSG shape-nodes, e.g., n_Z becomes $n_{Z \cup \{x\}}$ or $n_{Z-\{x\}}$.

—The cases of the abstract semantics use "abstract predicates" over SSG shape-graph elements. These provide safe tests for corresponding "concrete predicates" on DSG shape-graphs.

Figure 10 lists four of the abstract predicates that are used in the abstract semantics and the corresponding concrete predicates.

Each of the abstract predicates $p^\sharp$ in Figure 10 has the property that if the concrete property p holds on the elements of a given DSG SG, then $p^\sharp$ holds on the corresponding elements of $\beta(SG)$. Therefore, the abstract semantics can use $p^\sharp = \mathit{false}$ as a safe test of whether p holds on the corresponding elements in any of the DSGs in $\gamma(SG^\sharp)$: if $p^\sharp$ equals *false* (on specific elements of an SSG $SG^\sharp$), then p does not hold on the corresponding elements in any of the DSGs in $\gamma(SG^\sharp)$. For example, when $iis^\sharp[E_s^\sharp](n_X)$ does not hold, we conclude that $iis[E_s]$ does not hold on any of the cons-cells represented by n_X (i.e., none of the cons-cells represented by n_X are shared). In the SSG transformer for a statement of the form "$x.sel := \mathbf{nil}$," which removes the selector-edges emanating from all shape-nodes n_X with x in their name, the abstract semantics can determine whether it is safe to set $is^\sharp(n_X)$ to *false* by testing the value of $iis^\sharp(n_X)$.

This relationship is captured by the following lemma about the properties listed in Figure 10:

LEMMA 5.3.1. *Let $SG = \langle E_v, E_s \rangle$ be some DSG in $\mathcal{DSG}$; let l, l_1, l_2, $\ldots$, l_n be shape-nodes in shape_nodes$(\langle E_v, E_s \rangle)$; and let β denote $\beta[E_v]$.*

(1) $\mathit{compatible}(l_1, \ldots, l_n) \Rightarrow \mathit{compatible}^\sharp(\beta(l_1), \ldots, \beta(l_n))$
(2) $l_1 = l_2 \Rightarrow \beta(l_1) =^\sharp \beta(l_2)$
(3) $l_1 \neq l_2 \Rightarrow \beta(l_1) \neq^\sharp \beta(l_2)$
(4) $iis[E_s](l) \Rightarrow iis^\sharp[\beta(E_s)](\beta(l))$

PROOF. The abstract properties are derived from the concrete ones using the following observations:

(1) A shape-node n_Z represents cons-cells that are simultaneously pointed to by all (and only) the variables in Z. In a given DSG, each program variable points to at most one cons-cell. Consequently, two different shape-nodes n_{Z_i} and n_{Z_j}, such that $Z_i \cap Z_j \neq \phi$, represent incompatible configurations of variables: they cannot possibly represent cons-cells that are in the same DSG. Therefore, two different SSG shape-nodes n_{Z_i} and n_{Z_j} can represent cons-cells in the same DSG only if $Z_i \cap Z_j = \phi$.

(2) A given cons-cell in SG is represented by a unique SSG shape-node in $\beta(SG)$. Therefore, predicate $n_{Z_1} =^\sharp n_{Z_2}$ tests whether n_{Z_1} and n_{Z_2} are the same.

(3) Different cons-cells in SG are either represented in $\beta(SG)$ by different SSG shape-nodes, or else both are represented by summary node n_ϕ.

(4) Let l_1, l_2, sel_1, and sel_2 be elements of SG that satisfy the conditions of the existential quantifiers in column two of the case for $iis[E_s](l)$ in Figure 10. We

$$[\![x := \mathbf{nil}]\!]^\sharp(\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle) \stackrel{\text{def}}{=} \langle\langle E_v^{\sharp\prime}, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle, \text{where}$$

$$\begin{aligned}
f_x(n_W) &= n_{W-\{x\}} \\
E_v^{\sharp\prime} &= \{[y, f_x(n_W)] \mid [y, n_W] \in E_v^\sharp \wedge y \not\equiv x\} \\
E_s^{\sharp\prime} &= \{\langle f_x(n_V), sel, f_x(n_W)\rangle \mid \langle n_V, sel, n_W\rangle \in E_s^\sharp\} \\
is^{\sharp\prime}(n_Z) &= is^\sharp(n_Z) \vee is^\sharp(n_{Z\cup\{x\}})
\end{aligned}$$

$$[\![x.sel := \mathbf{nil}]\!]^\sharp(\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle) \stackrel{\text{def}}{=} \langle\langle E_v^\sharp, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle, \text{where}$$

$$\begin{aligned}
E_s^{\sharp\prime} &= E_s^\sharp - \{\langle n_X, sel, *\rangle \mid x \in X\} \\
is^{\sharp\prime}(n) &= \begin{cases} is^\sharp(n) \wedge iis^\sharp[E_s^{\sharp\prime}](n) & \text{if } \exists n_X \,.\, [x, n_X] \in E_v^\sharp \wedge \langle n_X, sel, n\rangle \in E_s^\sharp \\ is^\sharp(n) & \text{otherwise} \end{cases}
\end{aligned}$$

$$[\![x := \mathbf{new}]\!]^\sharp(\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle) \stackrel{\text{def}}{=} \langle\langle E_v^\sharp \cup \{[x, n_{\{x\}}]\}, E_s^\sharp\rangle, is^\sharp[n_{\{x\}} \mapsto \mathit{false}]\rangle$$

$$[\![x := y]\!]^\sharp(\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle) \stackrel{\text{def}}{=} \langle\langle E_v^{\sharp\prime}, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle, \text{where}$$

$$\begin{aligned}
g_{x,y}(n_Z) &= \begin{cases} n_{Z\cup\{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases} \\
E_v^{\sharp\prime} &= \{[z, g_{x,y}(n_Z)] \mid [z, n_Z] \in E_v^\sharp\} \cup \{[x, g_{x,y}(n_Z)] \mid [y, n_Z] \in E_v^\sharp\} \\
E_s^{\sharp\prime} &= \{\langle g_{x,y}(n_{Z_1}), sel, g_{x,y}(n_{Z_2})\rangle \mid \langle n_{Z_1}, sel, n_{Z_2}\rangle \in E_s^\sharp\} \\
is^{\sharp\prime}(n_Z) &= is^\sharp(n_{Z-\{x\}})
\end{aligned}$$

$$[\![x.sel := y]\!]^\sharp(\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle) \stackrel{\text{def}}{=} \langle\langle E_v^\sharp, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle, \text{where}$$

$$\begin{aligned}
E_s^{\sharp\prime} &= E_s^\sharp \cup \{\langle n_X, sel, n_Y\rangle \mid [x, n_X], [y, n_Y] \in E_v^\sharp \wedge \mathit{compatible}^\sharp(n_X, n_Y)\} \\
is^{\sharp\prime}(n) &= \begin{cases} is^\sharp(n) \vee iis^\sharp[E_s^{\sharp\prime}](n) & \text{if } [y, n] \in E_v^\sharp \\ is^\sharp(n) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 8. The abstract semantics $[\![\,]\!]^\sharp : \mathcal{SSG} \to \mathcal{SSG}$ for the five of the six kinds of statements that manipulate pointer variables. (By convention, $is^\sharp(n) = \mathit{false}$ if $n \notin \mathit{shape\_nodes}(SG^\sharp)$.)

have

$$\begin{array}{rcll}
\mathit{true} & \Rightarrow & \mathit{compatible}(l_1, l_2, l) & \text{Figure 10} \\
& \Rightarrow & \mathit{compatible}^\sharp(\beta(l_1), \beta(l_2), \beta(l)) & \text{Case (1) above} \\
\begin{array}{l} \langle l_1, sel_1, l\rangle \in E_s \\ \wedge\ \langle l_2, sel_2, l\rangle \in E_s \\ \wedge\ (l_1 \neq l_2 \vee sel_1 \neq sel_2) \end{array} & \Rightarrow & \begin{array}{l} \langle\beta(l_1), sel_1, \beta(l)\rangle \in \beta(E_s) \\ \wedge\ \langle\beta(l_2), sel_2, \beta(l)\rangle \in \beta(E_s) \\ \wedge\ (l_1 \neq l_2 \vee sel_1 \neq sel_2) \end{array} & \text{Definition 5.2.1} \\
& \Rightarrow & \begin{array}{l} \langle\beta(l_1), sel_1, \beta(l)\rangle \in \beta(E_s) \\ \wedge\ \langle\beta(l_2), sel_2, \beta(l)\rangle \in \beta(E_s) \\ \wedge\ (\beta(l_1) \neq^\sharp \beta(l_2) \vee sel_1 \neq sel_2) \end{array} & \text{Case (3) above}
\end{array}$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{[\![x := y.sel]\!]^\sharp(\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle) \stackrel{\text{def}}{=} \langle\langle E_v^{\sharp\prime}, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle, \text{where}} \\
h_x(n_Z) & = & n_{Z\cup\{x\}} \\
E_v^{\sharp\prime} & = & \begin{array}{l} E_v^\sharp \;\cup\; \bigcup_{[y,n_Y]\in E_v^\sharp, \langle n_Y, sel, n_Z\rangle \in E_s^\sharp} \{[x, h_x(n_Z)]\} \\ \qquad \cup\; \bigcup_{[y,n_Y]\in E_v^\sharp, \langle n_Y, sel, n_Z\rangle \in E_s^\sharp, [z,n_Z]\in E_v^\sharp} \{[z, h_x(n_Z)]\} \end{array} \\
E_s^{\sharp\prime} & = & \begin{array}{l} \quad (E_s^\sharp - \{\langle n_Y, sel, *\rangle \mid y \in Y\}) \\ \cup\; \bigcup_{[y,n_Y]\in E_v^\sharp, \langle n_Y, sel, n_Z\rangle \in E_s^\sharp} as(h_x, y, n_Y, sel, n_Z) \end{array} \\
is^{\sharp\prime}(n_Z) & = & is^\sharp(n_{Z-\{x\}}) \\
as(h_x, y, n_Y, sel, n_Z) & = & \begin{array}{l} \quad \left\{ \begin{array}{l} \langle n_W, sel', h_x(n_Z)\rangle \mid \\ compat\_in^\sharp([y, n_Y], \langle n_Y, sel, n_Z\rangle, \langle n_W, sel', n_Z\rangle) \end{array} \right\} \\ \cup \left\{ \begin{array}{l} \langle h_x(n_Z), sel', h_x(n_Z)\rangle \mid \\ compat\_self^\sharp([y, n_Y], \langle n_Y, sel, n_Z\rangle, \langle n_Z, sel', n_Z\rangle) \end{array} \right\} \\ \cup \left\{ \begin{array}{l} \langle h_x(n_Z), sel', n_W\rangle \mid \\ compat\_out^\sharp([y, n_Y], \langle n_Y, sel, n_Z\rangle, \langle n_Z, sel', n_W\rangle) \end{array} \right\} \end{array}
\end{array}
$$

Fig. 9. The abstract semantics for statements of the form $x := y.sel$. This case implements node materialization.

Concrete Predicates	
Usage	**Meaning**
$compatible(l_1, \ldots, l_k)$	$true$
$l_1 = l_2$	$l_1 = l_2$
$l_1 \neq l_2$	$l_1 \neq l_2$
$iis[E_s](l)$	$\exists l_1, l_2, \exists sel_1, sel_2 :$ $\langle l_1, sel_1, l\rangle, \langle l_2, sel_2, l\rangle \in E_s$ $\wedge\ (l_1 \neq l_2 \vee sel_1 \neq sel_2)$
Abstract Predicates	
Usage	**Meaning**
$compatible^\sharp(n_{Z_1}, \ldots, n_{Z_k})$	$\forall i, j : Z_i = Z_j \vee Z_i \cap Z_j = \phi$
$n_{Z_1} =^\sharp n_{Z_2}$	$Z_1 = Z_2$
$n_{Z_1} \neq^\sharp n_{Z_2}$	$Z_1 \neq Z_2 \vee Z_1 = Z_2 = \phi$
$iis^\sharp[E_s^\sharp](n_Z)$	$\exists n_{Z_1}, n_{Z_2}, \exists sel_1, sel_2 :$ $compatible^\sharp(n_{Z_1}, n_{Z_2}, n_Z)$ $\wedge\ \langle n_{Z_1}, sel_1, n_Z\rangle, \langle n_{Z_2}, sel_2, n_Z\rangle \in E_s^\sharp$ $\wedge\ (n_{Z_1} \neq^\sharp n_{Z_2} \vee sel_1 \neq sel_2)$

Fig. 10. The basic concrete and abstract properties used in the abstract semantics.

This shows that there exist elements $\beta(l_1)$, $\beta(l_2)$, sel_1, and sel_2 in $\beta(SG)$ such that $compatible^\sharp(\beta(l_1), \beta(l_2), \beta(l)) \wedge \langle \beta(l_1), sel_1, \beta(l)\rangle \in \beta(E_s) \wedge \langle \beta(l_2), sel_2, \beta(l)\rangle \in \beta(E_s) \wedge (\beta(l_1) \neq^\sharp \beta(l_2) \vee sel_1 \neq sel_2)$. Therefore, $iis^\sharp[E_s^\sharp](\beta(l))$ holds. □

The above four predicates are examples of a more general principle:

Definition 5.3.2. Let p be any predicate on various DSG components (i.e., shape-nodes, variable-edges, selector-edges, etc.). Similarly, let $p^\sharp$ be a predicate on the corresponding kinds of SSG components. We say that $p^\sharp$ is a *safe approximation* of p (denoted by $p \Rightarrow_\beta p^\sharp$) if we have for every $\langle E_v, E_s\rangle \in \mathcal{DSG}$ and components A,

B, ... of $\langle E_v, E_s \rangle$

$$p(A, B, \ldots) \Rightarrow p^\sharp(\beta[E_v](A), \beta[E_v](B), \ldots).$$

The case of the abstract semantics that handles statements of the form "$x := y.sel$" make use of three additional abstract predicates: $compat\_in^\sharp$, $compat\_self^\sharp$, and $compat\_out^\sharp$. These will be discussed in detail later in the section.

We now discuss the individual cases of the abstract meaning function (Figures 8 and 9), illustrating the most important features using Figure 11, which shows the final SSGs computed for each program point by abstract interpretation of the destructive list-reversal program. Each block of Figure 11 indicates the shape of memory just *before* the program-point label that appears at the bottom of the block. The set listed at the top of each block indicates the vertex (or vertices) in the control-flow graph and the action(s) taken there. For example, the block labeled v_{15} (see the lower right-hand corner of Figure 11) indicates that vertex v_{15}'s one predecessor is the statement $t_1 :=$ **nil** at v_{14}.

—For an assignment statement of the form $x :=$ **nil**, x is "liquidized" from all shape-nodes that have variable x in their "name." That is, x is removed from the name of all such shape-node names, which 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 11, the assignment $t_1 :=$ **nil** causes t_1 to be removed from the "name" of shape-node $n_{\{y,x,t_1\}}$. Shape-nodes $n_{\{y,x,t_1\}}$ and $n_{\{y,x\}}$ are then merged.

—For an assignment statement of the form $x.sel :=$ **nil**, the SSG transformer given in Figure 8 removes all of the *sel* selector-edges in shape-nodes that have x in their "name." This is safe because the variable set of a shape-node in the SSG for a program-point consists of variables that all point to the same cons-cell; therefore, all shape-nodes that have x in their name represent cons-cells whose *sel* field will be overwritten. (Conversely, if it is possible for a concrete cons-cell l that is not pointed to by variable x to arise at this statement, then l's *sel* field will not be overwritten; l will be represented in the SSG by a shape-node that does not have x in its "name.") See also the discussion of "strong nullification" in Section 5.4.

Example. In Figure 11, the transition between v_{11} and v_{12} removes selector-edge $\langle n_{\{y\}}, cdr, n_{\{x,t_1\}} \rangle$.

The other important aspect of the SSG transformer for $x.sel :=$ **nil** is the way information in shape-node names is used to reset sharing information. This is based on the observation that it is safe to reset $is^\sharp(n)$ to *false* whenever $iis^\sharp(n)$ is *false*. (The resetting of sharing information by the SSG transformer is not illustrated by the list-reversal program, since $is^\sharp$ is *false* for all shape-nodes in all shape-graphs that arise. The issue of resetting sharing information to *false* is discussed in detail in Section 5.5.)

—For an assignment statement of the form $x :=$ **new**, a new unshared node $n_{\{x\}}$ is created. All other shape-nodes are unaffected.

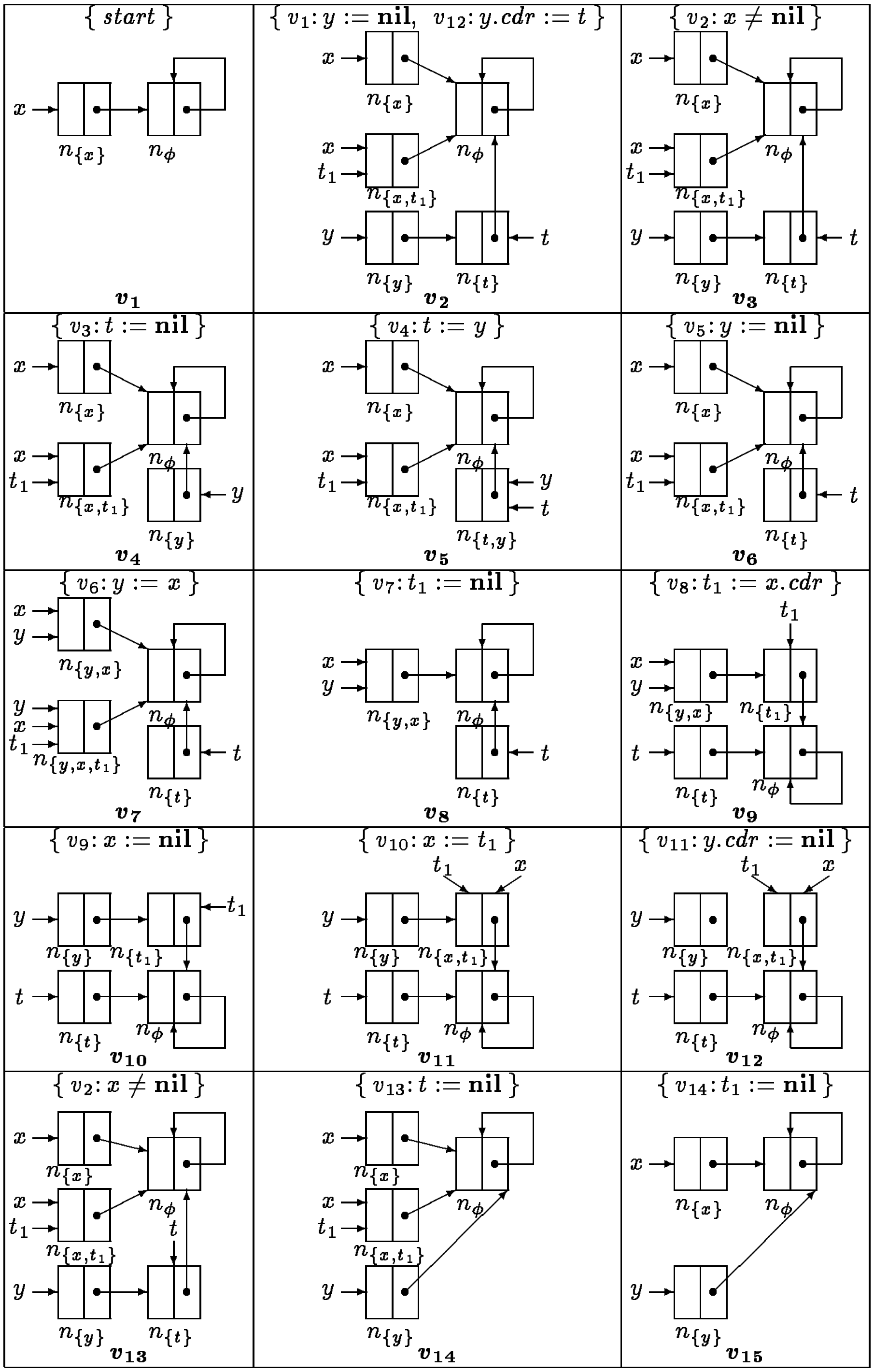


Fig. 11. The final SSGs computed for the destructive list-reversal program. For each of the shape-nodes in all of the SSGs, the value of $is^\sharp$ is *false*.

—For an assignment statement of the form $x := y$, 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 11.

—For an assignment statement of the form $x.sel := y$, *sel* selector-edges are added between shape-nodes pointed to by x and compatible shape-nodes pointed to by y.

Example. See the transition between block v_{12} and block v_2 of Figure 11.

In addition, shape-nodes that are pointed to by y may have their $is^\sharp$ values adjusted if the concrete cons-cells they represent could have become shared.

—The SSG transformer for an assignment statement of the form $x := y.sel$ is the most elaborate operation (see Figure 9). The reason for this is that the SSG transformer has to create an SSG that conservatively covers all the possible new configurations of variable sets whose members all point to the same cons-cell after the assignment: in particular, if $y.sel$ points to n_Z, then a copy of n_Z is "materialized"—producing a "new" node $n_{Z\cup\{x\}}$ from "old" node n_Z. In defining the 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.

Example. See the transition between block v_8 and block v_9 of Figure 11, in which the assignment $t_1 := x.cdr$ causes node $n_{\{t_1\}}$ to be materialized from n_ϕ.

The other important aspect of the SSG transformer for $x.sel :=$ **nil** is the way information in shape-node names is used to reset sharing information. This is based on the observation that it is safe to reset $is^\sharp(n)$ to *false* whenever $iis^\sharp(n)$ is *false*. (The resetting of sharing information by the SSG transformer is not illustrated by the list-reversal program, since $is^\sharp$ is *false* for all shape-nodes in all shape-graphs that arise. The issue of resetting sharing information to *false* is discussed in detail in Section 5.5.)

—For an assignment statement of the form $x :=$ **new**, a new unshared node $n_{\{x\}}$ is created. All other shape-nodes are unaffected.

—For an assignment statement of the form $x := y$, 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 11.

—For an assignment statement of the form $x.sel := y$, *sel* selector-edges are added between shape-nodes pointed to by x and compatible shape-nodes pointed to by y.

Example. See the transition between block v_{12} and block v_2 of Figure 11.

In what follows, let n_Y be a shape-node that y points to, and let n_Z be one of the shape-nodes that $y.sel$ points to (i.e., there is a selector-edge $\langle n_Y, sel, n_Z\rangle$). Bear in mind that

—$y.sel$ may have selector-edges to many shape-nodes and
—$y.sel$ may have a selector-edge to n_ϕ, which in general represents many concrete cons-cells in a single DSG.

For every node n_Z pointed to by $y.sel$, we materialize a new node $n_{Z\cup\{x\}}$ and direct the following variable-edges to $n_{Z\cup\{x\}}$:
—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 11.)
—A new variable-edge from x. (See variable-edge $[t_1, n_{\{t_1\}}]$ in block v_9 of Figure 11.)

In Figure 9, the process of determining the selector-edges that are to be directed to and from $n_{Z\cup\{x\}}$ is divided into three cases, based on three additional abstract predicates, $compat\_in^\sharp$, $compat\_self^\sharp$, and $compat\_out^\sharp$, defined in columns three and four of Figure 12.
—The property $compat\_in^\sharp$ describes when two selector-edges whose targets are both n_Z can possibly represent edges that coexist in the same concrete store. In particular, if an edge $\langle n_W, sel', n_Z\rangle$ is compatible with $\langle n_Y, sel, n_Z\rangle$, the abstract semantics for $x := y.sel$ generates an old $\rightarrow$ new selector-edge from the old node n_W into the new node $n_{Z\cup\{x\}}$.

Example. Selector-edge $\langle n_{\{y,x\}}, cdr, n_\phi\rangle$ in block v_8 of Figure 11 is a compatible incoming edge with respect to itself. This generates edge $\langle n_{\{y,x\}}, cdr, n_{\{t_1\}}\rangle$ in block v_9.

Note that if $is^\sharp(n_Z) = false$, all of n_Z's incoming edges—other than $\langle n_Y, sel, n_Z\rangle$ itself—are incompatible with $\langle n_Y, sel, n_Z\rangle$. In this case, the *only* old $\rightarrow$ new edge generated is $\langle n_Y, sel, n_{Z\cup\{x\}}\rangle$. (This situation is illustrated by the above example.)

When $y.sel$ points to n_ϕ, if n_ϕ has a direct cycle of the form $\langle n_\phi, sel', n_\phi\rangle$, this also counts as an "incoming" edge of n_ϕ. If $is^\sharp(n_\phi) = true$, such an edge will be "materialized" in two ways: as an edge $\langle n_\phi, sel', n_{\{x\}}\rangle$ and as a direct cycle $\langle n_{\{x\}}, sel', n_{\{x\}}\rangle$. (The latter is handled by the set former that uses predicate $compat\_self^\sharp$; see the discussion below.)

Example. In block v_8 of Figure 11, $is^\sharp(n_\phi) = false$, so the direct cycle $\langle n_\phi, cdr, n_\phi\rangle$ is incompatible with $\langle n_{\{y,x\}}, cdr, n_\phi\rangle$. Consequently, a selector-edge $\langle n_\phi, cdr, n_{\{t_1\}}\rangle$ is *not* generated when $n_{\{t_1\}}$ is materialized from n_ϕ in the transition from block v_8 to block v_9.

—The property $compat\_self^\sharp$ is used to define when a direct cycle $\langle n_Z, sel', n_Z\rangle$ is materialized as a direct cycle $\langle n_{Z\cup\{x\}}, sel', n_{Z\cup\{x\}}\rangle$. If the edges $\langle n_Z, sel', n_Z\rangle$ and $\langle n_Y, sel, n_Z\rangle$ represent a direct cycle and an edge that can possibly coexist in the same concrete store, the abstract semantics for $x := y.sel$ generates a new $\rightarrow$ new selector-edge from the new node $n_{Z\cup\{x\}}$ to $n_{Z\cup\{x\}}$. Note

Concrete Predicates	
Usage	**Meaning**
$compat\_in\left(\begin{array}{l}[y,l_1],\\ \langle l_1,sel,l_2\rangle,\\ \langle l_3,sel',l_2\rangle\end{array}\right)$	$[y,l_1] \in E_v$ $\wedge\ \langle l_1,sel,l_2\rangle \in E_s$ $\wedge\ \langle l_3,sel',l_2\rangle \in E_s$ $\wedge\ l_3 \neq l_2$
$compat\_self\left(\begin{array}{l}[y,l_1],\\ \langle l_1,sel,l_2\rangle,\\ \langle l_2,sel',l_2\rangle\end{array}\right)$	$[y,l_1] \in E_v$ $\wedge\ \langle l_1,sel,l_2\rangle \in E_s$ $\wedge\ \langle l_2,sel',l_2\rangle \in E_s$
$compat\_out\left(\begin{array}{l}[y,l_1],\\ \langle l_1,sel,l_2\rangle,\\ \langle l_2,sel',l_3\rangle\end{array}\right)$	$[y,l_1] \in E_v$ $\wedge\ \langle l_1,sel,l_2\rangle \in E_s$ $\wedge\ \langle l_2,sel',l_3\rangle \in E_s$ $\wedge\ l_3 \neq l_2$

Abstract Predicates	
Usage	**Meaning**
$compat\_in^\sharp\left(\begin{array}{l}[y,n_Y],\\ \langle n_Y,sel,n_Z\rangle,\\ \langle n_W,sel',n_Z\rangle\end{array}\right)$	$compatible^\sharp(n_Y,n_Z,n_W)$ $\wedge\ [y,n_Y] \in E_v^\sharp$ $\wedge\ \langle n_Y,sel,n_Z\rangle, \langle n_W,sel',n_Z\rangle \in E_s^\sharp$ $\wedge\ n_Z \neq^\sharp n_W$ $\wedge\ ((n_Y =^\sharp n_W \wedge sel = sel')$ $\vee is^\sharp(n_Z))$
$compat\_self^\sharp\left(\begin{array}{l}[y,n_Y],\\ \langle n_Y,sel,n_Z\rangle,\\ \langle n_Z,sel',n_Z\rangle\end{array}\right)$	$compatible^\sharp(n_Y,n_Z)$ $\wedge\ [y,n_Y] \in E_v^\sharp$ $\wedge\ \langle n_Y,sel,n_Z\rangle, \langle n_Z,sel',n_Z\rangle \in E_s^\sharp$ $\wedge\ ((n_Y =^\sharp n_Z \wedge sel = sel')$ $\vee is^\sharp(n_Z))$
$compat\_out^\sharp\left(\begin{array}{l}[y,n_Y],\\ \langle n_Y,sel,n_Z\rangle,\\ \langle n_Z,sel',n_W\rangle\end{array}\right)$	$compatible^\sharp(n_Y,n_Z,n_W)$ $\wedge\ [y,n_Y] \in E_v^\sharp$ $\wedge\ \langle n_Y,sel,n_Z\rangle \langle n_Z,sel',n_W\rangle \in E_s^\sharp$ $\wedge\ n_Z \neq^\sharp n_W$ $\wedge\ (n_Y \neq^\sharp n_Z \vee sel \neq sel')$

Fig. 12. Properties used in the "$x := y.sel$" case of the abstract semantics.

that if $is^\sharp(n_Z) = false$, all of n_Z's incoming self edges are *incompatible* with $\langle n_Y, sel, n_Z\rangle$, unless n_Y and n_Z are the *same* shape-node and $sel' = sel$.

Example. Selector-edge $\langle n_\phi, cdr, n_\phi\rangle$ in block v_8 of Figure 11 is an incompatible self edge with respect to edge $\langle n_{\{y,x\}}, cdr, n_\phi\rangle$, and hence a selector-edge $\langle n_{\{t_1\}}, cdr, n_{\{t_1\}}\rangle$ is *not* generated when $n_{\{t_1\}}$ is materialized from n_ϕ in the transition from block v_8 to block v_9.

—The property $compat\_out^\sharp$ describes when an outgoing selector-edge $\langle n_Z, sel', n_W\rangle$ of n_Z represents an edge that can possibly coexist in the same concrete store with an edge represented by selector-edge $\langle n_Y, sel, n_Z\rangle$. If $\langle n_Z, sel', n_W\rangle$ and $\langle n_Y, sel, n_Z\rangle$ are compatible, the abstract semantics for $x := y.sel$ generates a new $\rightarrow$ old selector-edge from the new node $n_{Z\cup\{x\}}$ to the old node n_W.

Example. Selector-edge $\langle n_\phi, cdr, n_\phi\rangle$ in block v_8 of Figure 11 is a compatible outgoing edge of n_ϕ with respect to edge $\langle n_{\{y,x\}}, cdr, n_\phi\rangle$, and hence a selector-edge $\langle n_{\{t_1\}}, cdr, n_\phi\rangle$ is generated when $n_{\{t_1\}}$ is materialized from n_ϕ in the transition from block v_8 to block v_9.

> Because each field of a concrete cons-cell can have only a single outgoing edge, if $n_Y = n_Z$, then $sel' \neq sel$ (or equivalently, $n_Y \neq n_Z \vee sel' \neq sel$).

Note that all of the operations of the abstract semantics preserve the "compatibility" property for the variable-set names of selector-edge end-points described in Section 5.2.

The shape-analysis algorithm itself is an iterative procedure that computes an SSG, $SG_v^\sharp$, for each control-flow graph v, as the least fixed point (under the ordering defined in Definition 5.1.2) of the following system of equations in $SG_v^\sharp$:

$$SG_v^\sharp = \begin{cases} \langle\langle \phi, \phi\rangle, \lambda n.\mathit{false}\rangle & \text{if } v = \mathit{start} \\ \bigsqcup_{\langle u,v\rangle \in A} [\![st(u)]\!]^\sharp(SG_u^\sharp) & \text{otherwise} \end{cases} \tag{2}$$

The iteration starts from the initial assignment $SG_w^\sharp = \langle\langle \phi, \phi\rangle, \lambda n.\mathit{false}\rangle$ for each control-flow-graph vertex w.

Example 5.3.3. The final abstract values for all of the control-flow-graph vertices of the normalized list-reversal program are shown in Figure 11. Among other things, this information tells us 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}).

Block v_2 of Figure 11 shows the final SSG computed for vertex v_2 of the list-reversal program. 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 $n_{\{x,t_1\}}$. Shape-node $n_{\{x\}}$ represents the situation where x is the only variable pointing to the head of the list (which only happens before the first iteration of the loop). Shape-node $n_{\{x,t_1\}}$ 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 head of the list that t points to, which is a sublist of the list that y points to. Shape-node n_ϕ represents all of the cons-cells in the tails of the lists that x, t_1, and t point to.

—For each of the shape-nodes in the graph, the value of $is^\sharp$ is false. The fact that $is^\sharp(n_\phi) = \mathit{false}$ tells us a number of interesting things about the memory state produced by any execution sequence that ends at vertex v_2: (1) it implies that the *cdr*-edges from the cons-cells that x and t point to cannot point to the same cons-cell (and consequently that the tails of the x-list and the t-list cannot share any cons-cells in common) and (2) similarly, for every pair of different cons-cells in the tail of x (respectively, t), the *cdr*-edges from these cons-cells cannot point to the same cons-cell. Consequently, the x-list (respectively, t-list) is an acyclic list.

Example 5.3.4. It is instructive to consider the algorithm's behavior when working with a program that uses the convention that the end of a linked list is represented by an element whose *cdr* field points to itself (e.g., see Sedgewick [1988, p.17–20]). This termination node would be represented in the SSG by a shape-node for which $is^\sharp$ is *true*. However, if the convention is that there is a single copy of this node—shared by multiple lists—that is pointed to by a global variable, say *nil*, then the termination node $n_{\{nil\}}$ is distinct from summary-node n_ϕ in the SSG, and list-manipulation programs are analyzed as precisely as before.

Alternatively, suppose there are multiple termination nodes—one for each list—and that each linked-list object is represented using a pair of pointer variables, one pointing to the head of the list and one to the termination node. Again, the termination node will be distinct from summary-node n_ϕ in the SSG, and list-manipulation programs are analyzed as precisely as before.

If no pointer to the termination node is maintained, then $is^\sharp(n_\phi)$ will be *true*. Therefore, techniques such as the ones discussed in Section 6.1 need to be used, or else much of the precision of the analysis will be lost.

Termination and safety of the shape-analysis algorithm are argued in the standard manner [Cousot and Cousot 1977]. For a given program P, we work with the domain of SSGs in which $PVar$ consists of the program variables in P. This domain is finite and hence of finite height. Termination is assured because the semantic equations of Figures 8 and 9 are monotonic:

PROPOSITION 5.3.5. (MONOTONICITY OF $[\![\,]\!]^\sharp$). *For all assignment statements st, and for each pair of SSGs $SG_1^\sharp$ and $SG_2^\sharp$ such that $SG_1^\sharp \sqsubseteq SG_2^\sharp$, we have $[\![st]\!]^\sharp(SG_1^\sharp) \sqsubseteq [\![st]\!]^\sharp(SG_2^\sharp)$.*

The heart of the safety argument involves showing that each semantic equation of the abstract semantics is conservative with respect to the corresponding equation of the concrete semantics:

THEOREM 5.3.6. (LOCAL SAFETY THEOREM). *For all assignment statements st, and for every $SG \in \mathcal{DSG}$, we have $\beta([\![st]\!](SG)) \sqsubseteq [\![st]\!]^\sharp(\beta(SG))$.*

PROOF. See the electronic appendix. □

Finally, we have the following

THEOREM 5.3.7. (GLOBAL SAFETY THEOREM). *For every control-flow-graph vertex v, we have $\alpha(cs(v)) \sqsubseteq SG_v^\sharp$.*

PROOF. Immediate from Proposition 5.3.5 (Monotonicity) and Theorem 5.3.6 (Local Safety), using Theorem T2 of Cousot and Cuosot [1977, p.252]. □

5.4 Strong Nullification

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 cons-cell. This permits statements of the form $x.sel :=$ **nil** to be treated in an unusual manner—unusual for a static-analysis algorithm, that is. When the algorithm processes such a statement, it always *removes* the *sel* edges emanating from the shape-nodes that x points to (i.e., the shape nodes n_X such that $x \in X$). We call this operation *strong nullification*.

Example 5.4.1. Figure 13 shows a simple example that illustrates why strong nullification is possible. After the statement $x := y$ in the then-branch of the conditional, x and y point to the same cons-cell, and z points to a different cons-cell. Similarly, after the statement $x := z$ in the else-branch of the conditional, x

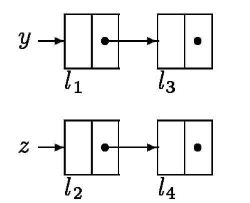
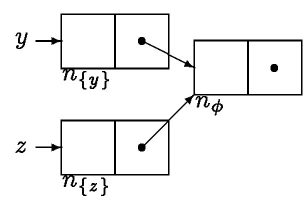
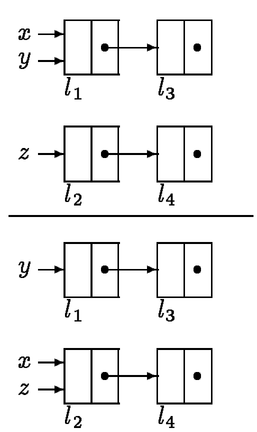
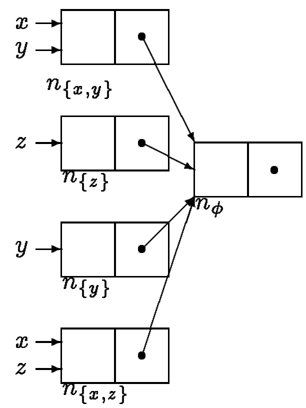
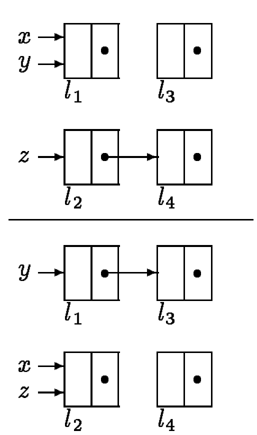
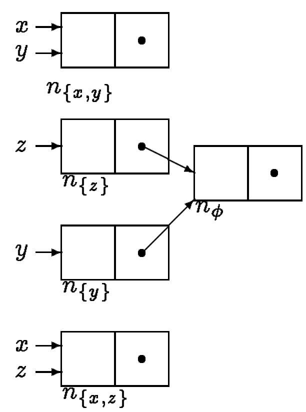
Program	Collecting semantics	Abstract semantics
		
if $\cdots$ **then** $x := y$ **else** $x := z$ **fi**		
		
$x.cdr :=$ **nil**		
		

Fig. 13. A program that illustrates strong nullification.

and z point to the same cons-cell, and y points to a different cons-cell. Thus, in the SSG after the conditional statement, x and y both point to shape-node $n_{\{x,y\}}$, and z points to $n_{\{z\}}$ (reflecting the state of memory after the then-branch is executed); in addition, x and z both point to shape-node $n_{\{x,z\}}$, and y points to $n_{\{y\}}$ (reflecting the state of memory after the else-branch is executed).

The abstract semantics for the statement $x.cdr$:= **nil** eliminates the edges $\langle n_{\{x,y\}}, cdr, n_\phi\rangle$ and $\langle n_{\{x,z\}}, cdr, n_\phi\rangle$. This is safe because $n_{\{x,y\}}$ represents only cons-cells that are pointed to by *both* x and y (which occurs only on *some* execution sequences), and $n_{\{x,z\}}$ represents only cons-cells that are pointed to by both x and z (which also occurs only on some execution sequences).

The abstract semantics for $x.cdr$:= **nil** retains the edges $\langle n_{\{z\}}, cdr, n_\phi\rangle$ and $\langle n_{\{y\}}, cdr, n_\phi\rangle$. This correctly captures the fact that in the collecting semantics after $x.cdr$:= **nil**, there is a DSG that contains a cons-cell pointed to by y alone with an outgoing *cdr*-edge (to l_3), as well as a DSG that contains a cons-cell pointed to by z alone with an outgoing *cdr*-edge (to l_4).

Other shape-analysis algorithms do not perform strong nullification for a statement of the form "$x.sel$:= **nil**," except under very limited circumstances [Chase et al. 1990; Jones and Muchnick 1981; Larus 1989; Larus and Hilfinger 1988; Plevyak et al. 1993]. The reason for this is that, after the conditional statement in Example 5.4.1, they perform actions that (in our terminology) are equivalent to merging $n_{\{y\}}$ and $n_{\{x,y\}}$ into one shape-node. When this is done, it is not safe to perform strong nullification—i.e., to remove the shape-node's outgoing *cdr* edge—because it would lose the information that y can, in fact, point to a cons-cell that has an outgoing *cdr*-edge.

In contrast, our shape-analysis algorithm always performs strong nullification. However, we are *not* claiming that our method is somehow "able to treat all statements precisely." With our method, the place where precision is lost (i.e., when it is lost) occurs in the treatment of statements of the form "$x := y.sel$," rather than in statements of the form "$x.sel$:= **nil**." In particular, this is a problem when y's *sel* field points to n_ϕ and $is^\sharp(n_\phi) = true$. In the SSG transformer for $x := y.sel$, node materialization creates shape-nodes that conservatively cover *all* the possible new configurations of variable sets whose members could all point to the same cons-cell. (See the treatment of the statement "$x := x.cdr$" in Figure 4(b).)

On the other hand, in comparing the capabilities of our method with those of other graph-based shape-analysis algorithms, it would be wrong to think that we have just shifted the place where imprecision is introduced from the treatment of statements of the form "$x.sel$:= **nil**" to statements of the form "$x := y.sel$." Not only do other shape-analysis algorithms use less precise SSG transformers for $x.sel$:= **nil** (i.e., performing strong nullification only under very limited circumstances), they also use less precise SSG transformers for statements of the form $x := y.sel$; namely, they merely advance x to point to whatever shape-node (or shape-nodes) $y.sel$ points to. In the case where $y.sel$ points to n_ϕ, this advances x into the primordial soup!

5.5 Insertion into a List

Whereas the previous sections have all considered the actions of the shape-analysis algorithm on the list-reversal program, this section considers a second example program, the list-insertion program shown in Figure 14, which may insert a cons-cell at an arbitrary point in a linked list. For this program, the shape-analysis algorithm is able to establish the following properties:

—"Listness" is preserved by the list-insert program. That is, when the initial value of variable x is an unshared acyclic list, the value of y at the end of the program is also an unshared acyclic list.

—"Circular listness" is also preserved by the list-insert program. More precisely, if at the beginning of the list-insert program x is a possibly cyclic list of length ≥ 1 (see Figure 7(e)), then at the end of the program, x is a possibly cyclic list of length ≥ 2 (see Figure 7(d)). (For details, see Sagiv et al.[1995, Appendix B].)

The list-insert program also illustrates an interesting capability of the shape-analysis algorithm that does not arise with the list-reversal program: in certain circumstances, information in shape-node names can be used to reset a shape-node's sharing information from *true* to *false*. In the case of the list-insert program, this feature plays a crucial role in the ability of the shape-analysis algorithm to determine that the program preserves both "listness" and "circular listness."

Assume that at the beginning of the list-insert program, x points to an unshared list of length 1 or more and that e points to the new element to be inserted. The key SSGs are those that arise at vertices v_{11}, v_{12}, and v_{13} of the control-flow graph, where the new element is spliced into the list. The crucial step is the transition from v_{12} : $y.cdr$:= **nil** to v_{13} In the immediately preceding transition, from v_{11} to v_{12} $e.cdr$ is assigned the value t, which adds a new selector-edge into $n_{\{t\}}$ and causes $is^\sharp(n_{\{t\}})$ to be set to *true* in the shape-graph for v_{12}.

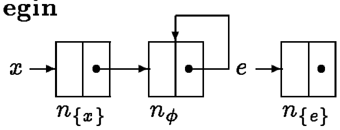
The strong nullification performed in the transition from v_{12} to v_{13} removes the selector-edges $\langle n_{\{z,y\}}, cdr, n_{\{t\}}\rangle$ and $\langle n_{\{x,y\}}, cdr, n_{\{t\}}\rangle$. Thus, in the SSG for vertex v_{13}, shape-node $n_{\{t\}}$ retains only a single incoming selector-edge, namely $\langle n_{\{e\}}, cdr, n_{\{t\}}\rangle$. In the SSG transformer given in Figure 8 that covers the assignment $y.cdr := \mathbf{nil}$, the fact that predicate $iis^\sharp$ is a safe approximation to iis is used to reset sharing information. In this case, because the value of $iis^\sharp[E_s^{\sharp\prime}](n_{\{t\}})$ at v_{12} is *false*, the SSG transformer for the control-flow graph arc from v_{12} to v_{13} determines that it is safe to set $is^\sharp(n_{\{t\}})$ to *false* in the SSG for vertex v_{13}. (See the third row of Figure 15.)

Remark. It is interesting to note that if the assignment at v_{12} were $e.cdr := \mathbf{nil}$, rather than $y.cdr := \mathbf{nil}$, $is^\sharp(n_{\{t\}})$ would still be set to *false* at v_{13}, even though there would be *two* incoming selector-edges to shape-node $n_{\{t\}}$: $\langle n_{\{z,y\}}, cdr, n_{\{t\}}\rangle$ and $\langle n_{\{x,y\}}, cdr, n_{\{t\}}\rangle$. Because these two edges are incompatible—they do not represent edges that can simultaneously coexist in a single concrete store—the value of $iis^\sharp[E_s^{\sharp\prime}](n_{\{t\}})$ would again be *false* at vertex v_{12}, and so $is^\sharp(n_{\{t\}})$ would be *false* at vertex v_{13}.

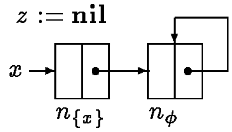
program insert(x, e)
begin
  $y := x$
  while $y.cdr \neq$ **nil** $\wedge \ldots$ **do**
    $y := y.cdr$ **od**
  $t := y.cdr$
  $e.cdr := t$
  $y.cdr := e$
  $t :=$ **nil**
  $e :=$ **nil**
  $y :=$ **nil**
end

(a)

program insert(x, e)
begin

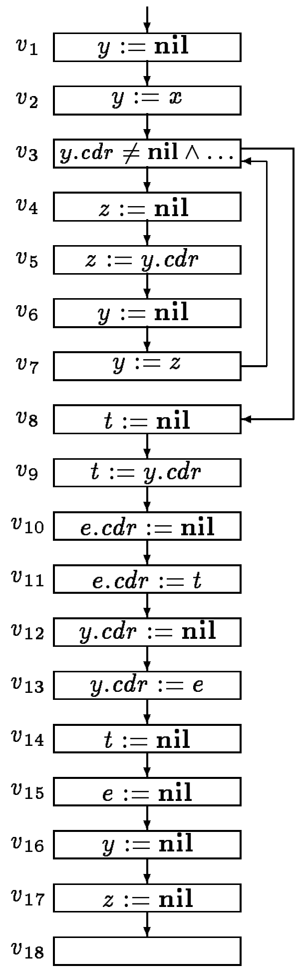


  $y :=$ **nil**
  $y := x$
  while $y.cdr \neq$ **nil** $\wedge \ldots$ **do**
    $z :=$ **nil**
    $z := y.cdr$
    $y :=$ **nil**
    $y := z$ **od**
  $t :=$ **nil**
  $t := y.cdr$
  $e.cdr :=$ **nil**
  $e.cdr := t$
  $y.cdr :=$ **nil**
  $y.cdr := e$
  $t :=$ **nil**
  $e :=$ **nil**
  $y :=$ **nil**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



end

(b)



(c)

Fig. 14. (a) A program that searches a list and splices a new element into the list. (b) The normalized program. (c) The normalized program's control-flow graph.

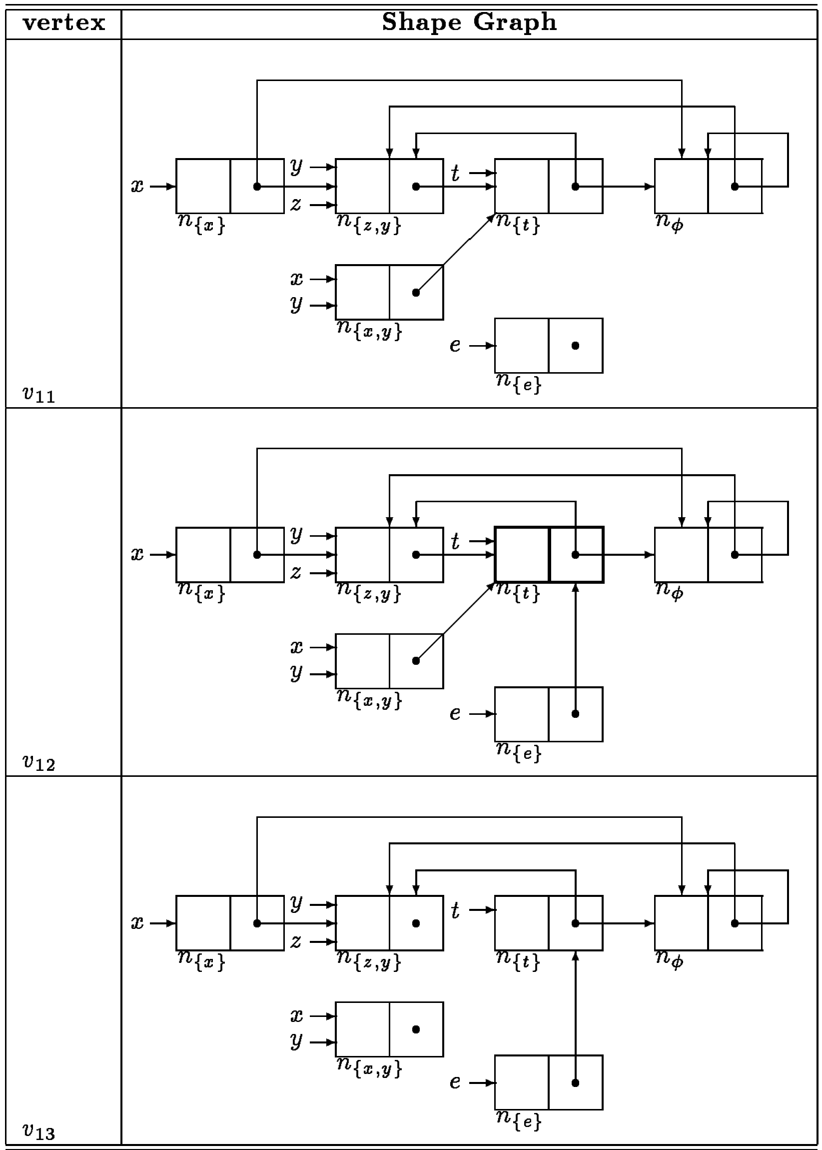


Fig. 15. The SSGs at vertices v_{11}, v_{12}, and v_{13} of the list-insert program. These illustrate how $is^\sharp(n_{\{t\}})$ is reset to *false* in the transition from v_{12} to v_{13}. First, the shape-graph for vertex v_{11} is shown. In this graph, $is^\sharp(n_{\{t\}}) = \mathit{false}$. Then, in the shape-graph for vertex v_{12}, $is^\sharp(n_{\{t\}}) = \mathit{true}$ (shown in bold). Finally, in the shape-graph for vertex v_{13}, $is^\sharp(n_{\{t\}}) = \mathit{false}$.

6. EXTENSIONS

This section discusses several variations on the basic method that has been presented above. Due to space limitations, they will only be sketched out below.

6.1 More Summary Shape-Nodes

The major source of inaccuracy in our method is attributable to the fact that, in general, summary shape-node n_ϕ represents a number of unrelated cons-cells. This is particularly a problem when $is^\sharp(n_\phi) = true$. For example, in a program that uses both a list and a graph, the tail of the list is abstracted to the same summary node as (most of) the graph's nodes. Consequently, the shape-analysis algorithm imprecisely identifies both structures as graphs: variables that actually point into the list appear to point to some kind of shared graph structure.

There are several simple ways to improve the accuracy of shape analysis by introducing more summary nodes, including the following:

— *Using two separate summary nodes*: One in which $is^\sharp(n_\phi) = false$, representing all nodes not pointed to by any variable and pointed to by at most one distinct cons-cell field, and one in which $is^\sharp(n_\phi) = true$, representing all nodes not pointed to by any variable and pointed to by two or more distinct cons-cell fields.

— *Using allocation-sites to identify shape-nodes [Chase et al. 1990; Jones and Muchnick 1982]*: This can be incorporated into our method as an "orthogonal dimension" of shape-node names—e.g., shape-nodes would have names such as $n_{s,X}$, where s is an allocation site, and X is a set of program variables. n_ϕ

— *Using some type information*: For example having one n_ϕ node for every declared data type.

However, even these extensions do not help solve the following kind of accuracy problem:

Example 6.1.1. At vertex v_{11} of the list-insert program, the shape-graph computed by our shape-analysis algorithm indicates that variables y and z can point to a *cyclic list* (see Figure 15, the v_{11} row). Note that during an execution of the program, at v_{11} variables y and z point into the middle of the (acyclic) list that x points to. The reason for the inaccuracy in the structure reported at v_{11} by the shape-analysis algorithm is that n_ϕ does double duty: (1) it represents the segment of the list in between x and y (cf. the selector-edges $\langle n_{\{x\}}, cdr, n_\phi\rangle$ and $\langle n_\phi, cdr, n_{\{z,y\}}\rangle$); (2) it also represents the segment of the list beyond what y points to (cf. the selector-edges $\langle n_{\{z,y\}}, cdr, n_{\{t\}}\rangle$ and $\langle n_{\{t\}}, cdr, n_\phi\rangle$). This, combined with the fact that $is^\sharp(n_{\{t\}}) = true$, causes it to appear that y and z may be pointing to a cyclic list.

In the case of the insert program, this temporary inaccuracy does not cause our algorithm any problems. In going from the second to the third row of Figure 15, the (apparent) cycle is broken, and thus the algorithm is still able to determine that at the end of the program x points to an acyclic list. However, for other programs we are not so fortunate. For example, in Lindstrom scanning of a tree [Lindstrom 1973] (also known as the Deutsch-Schorr-Waite algorithm for traversing a tree without a stack [Knuth 1973, p.417]) this kind of inaccuracy prevents us from finding that, after traversing a tree, we still have a tree.

It is possible to avoid this sort of inaccuracy by introducing additional summary nodes (with an appropriate naming scheme) to discriminate between cons-cells that are transitively pointed to by different collections of variables or via different se-

$x :=$ **new**
if $\cdots$ **then** $y_1 := x$ **fi**
if $\cdots$ **then** $y_2 := x$ **fi**
$\vdots$
if $\cdots$ **then** $y_n := x$ **fi**

Fig. 16. An artificial program for which an SSG with 2^n shape-nodes arises.

lectors. We have developed several alternative abstract semantics based on this idea.

6.2 Reducing the Number of 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. For example, the number of shape-nodes in the SSG that arises at the end of the program shown in Figure 16 is 2^n. Our limited experience to date suggests that this is unlikely to arise in practice, the main reason being that the number of possible aliasing configurations is normally small.

It is possible to change the shape-analysis algorithm to use *widening* to eliminate the possibility of exponential blow-up and to guarantee that a conservative solution to Eq. (2) of Section 5.3 can be found in polynomial time. The basic idea is to reduce the number of shape-nodes that can arise, by discarding an arbitrary amount of "simultaneously-points-to" information at certain shape-nodes, thereby trading off accuracy for efficiency. For instance, at various points in the shape-analysis algorithm (e.g., at loops) we can widen an SSG into a less precise, but usually more compact, SSG by merging shape-nodes, say n_{Z_1} and n_{Z_2}, into $n_{Z_1 \cap Z_2}$ and giving $n_{Z_1 \cap Z_2}$ all the variable- and selector-edges that were incident on n_{Z_1} and n_{Z_2}.

Formalizing this notion involves changing the $\mathcal{SSG}$ domain by weakening what has (up to now) been a fundamental condition on variable-sets in shape-node names. In particular, we now allow the name of a shape-node to be any *subset* of the variables pointing to it:

> As before, the SSG for program point p represents, in general, a number of DSGs. An SSG shape node n_Z, where $Z \neq \phi$, represents the (at most one) cons-cell in each DSG that is simultaneously pointed to by all the variables in Z. In addition, if w is a variable not in Z, but the SSG has a variable-edge $[w, n_Z]$, then variable w *may or may not* point to that same cons-cell.

For SSGs that are in the image of abstraction-function α (Definition 5.2.1), we have the following property:

> For every $x \in PVar$ and $n_X \in shape\_nodes(SG^\sharp)$, $x \in X$ if and only if $[x, n_X] \in E_v$.

For SSGs on which widening has been performed, we have the weaker property:

For every $x \in PVar$ and $n_X \in shape\_nodes(SG^\sharp)$, if $x \in X$ then $[x, n_X] \in E_v$.

The relation $\sqsubseteq'$ on the extended domain of SSGs captures the fact that the widened SSGs are less accurate than the original SSGs, i.e., a widened SSG denotes more DSGs than the original SSG. Formally, $\sqsubseteq'$ is a preordering on the extended domain of SSGs, defined as follows:

Definition 6.2.1. Let $SG_1^\sharp = \langle\langle E_{v1}^\sharp, E_{s1}^\sharp\rangle, is_1^\sharp\rangle$ and $SG_2^\sharp = \langle\langle E_{v2}^\sharp, E_{s2}^\sharp\rangle, is_2^\sharp\rangle$. $SG_1^\sharp \sqsubseteq' SG_2^\sharp$ if and only if there exists a mapping $r: 2^{PVar} \to 2^{PVar}$ such that for all $X \subseteq PVar$, $r(X) \subseteq X$, and

—for every $[x, n_X] \in E_{v1}^\sharp$, $[x, n_{r(X)}] \in E_{v2}^\sharp$,

—for every $\langle n_S, sel, n_T\rangle \in E_{s1}^\sharp$, $\langle n_{r(S)}, sel, n_{r(T)}\rangle \in E_{s2}^\sharp$, and

—for every $n_X \in shape\_nodes(SG_1^\sharp)$, $is_1^\sharp(n_X) \Rightarrow is_2^\sharp(n_{r(X)})$.

In Definition 6.2.1, function r has the ability to discard an arbitrary amount of "simultaneously-points-to" information at any shape-node. Note that we can still use graph union as a confluence operator.

We have been careful to write the abstract semantics given in Figures 8and 9 so that it does not have to be changed when widening is employed. For example, $x.sel := \mathbf{nil}$ removes x's *sel* selector-edges from a shape-node n_Z only when $x \in Z$. Thus, if $[x, n_Z] \in E_v$ but $x \notin Z$, the abstract semantics does not remove n_Z's outgoing *sel* edges: this would not be safe because we do not know that n_Z represents a cons-cell that x *must* point to. Note that we still *do* perform a strong nullification of n_Z's outgoing *sel* edges for assignments of the form $z.sel := \mathbf{nil}$, where $z \in Z$, because n_Z represents only cons-cells whose *sel* field will definitely be overwritten. (In Section 6.2.2, we define an operation that can be used to materialize shape-nodes in order to guarantee that if $[x, n_Z] \in E_v$ then $x \in Z$. By applying this operation prior to nullification, we can still *always* perform strong nullification, even if the shape-analysis algorithm performs widening.)

This observation is captured in the following proposition:

PROPOSITION 6.2.2. (GENERALIZED MONOTONICITY OF $[\![\,]\!]^\sharp$). *For all assignment statements st, and for each pair of SSGs $SG_1^\sharp$ and $SG_2^\sharp$ such that $SG_1^\sharp \sqsubseteq' SG_2^\sharp$, $[\![st]\!]^\sharp(SG_1^\sharp) \sqsubseteq' [\![st]\!]^\sharp(SG_2^\sharp)$.*

The following generalization of Theorem 5.3.6 is an immediate corollary:

COROLLARY 6.2.3. *For all assignment statements st, for every $SG \in \mathcal{DSG}$, and for every $SG^\sharp$ such that $\beta(SG) \sqsubseteq' SG^\sharp$, $\beta([\![st]\!](SG)) \sqsubseteq' [\![st]\!]^\sharp(SG^\sharp)$.*

PROOF. $\beta([\![st]\!](SG)) \sqsubseteq [\![st]\!]^\sharp(\beta(SG)) \sqsubseteq' [\![st]\!]^\sharp(SG^\sharp)$. $\square$

6.2.1 *Strategies for Merging Nodes.* There are many possible strategies for reducing the number of shape-nodes through widening. Different widening policies may lead to shape-analysis algorithms that differ in accuracy and efficiency. For example, we may decide to forget an arbitrary variable $z \in PVar$ by widening

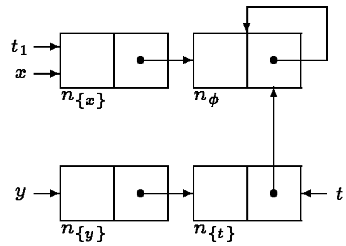


Fig. 17. The SSG obtained by eliminating t_1 from the name sets of the SSG shown in Figure 11.

$\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle$ into $\langle\langle E_v^{\sharp\prime}, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle$ where

$$\begin{aligned} f_z(n_W) &= n_{W-\{z\}} \\ E_v^{\sharp\prime} &= \{[y, f_z(n_W)] \mid [y, n_W] \in E_v^\sharp\} \\ E_s^{\sharp\prime} &= \{\langle f_z(n_V), sel, f_z(n_W)\rangle \mid \langle n_V, sel, n_W\rangle \in E_s^\sharp\} \\ is^{\sharp\prime}(n_W) &= is^\sharp(n_W) \vee is^\sharp(n_{W\cup\{z\}}) \end{aligned}$$

By definition, $\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle \sqsubseteq' \langle\langle E_v^{\sharp\prime}, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle$. It is possible to use this widening operator to guarantee that a conservative solution to Eq. (2) of Section 5.3 can be found in polynomial time. It 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 [Jones and Muchnick 1981], but it is likely to produce much better results because limiting the cardinality of name sets still preserves most of the structural information about the graph.)

Example 6.2.1.1. In the SSG shown in the v_2 box of Figure 11, we can eliminate t_1 from the name sets to get the SSG shown in Figure 17. In this case, widening amounts to not distinguishing between the store that arises just before the first iteration and the stores that arise on each succeeding iteration. Because t_1 is not live at v_2, and because there are no structural differences between the store that arises just before the first iteration and the ones that arise subsequently, widening does not lead to a loss of accuracy in this case.

As mentioned earlier, another possible widening strategy is to merge shape-nodes n_Y and n_Z into a shape-node $n_{(Y\cap Z)}$. This would seem to make the most sense when Y and Z have many variables in common.

Experimentation is necessary to determine what kind of widening works best in practice.

6.2.2 *Materializing Shape Nodes via Narrowing.* It is interesting to note that if widening has been performed we can also narrow an SSG into one that denotes the same DSGs but in some sense is more "precise." This operation may be employed gainfully just before the interpretation of statements of the form $x.sel$:= **nil** to allow the abstract semantics to always be able to perform strong nullification (i.e., to always remove x's sel selector-edges), even if the shape-analysis algorithm has widened the SSG with respect to variable x. The narrowing operation converts an

SSG $\langle\langle E_v^\sharp, E_s^\sharp\rangle, is^\sharp\rangle$ into an SSG $\langle\langle E_v^{\sharp\prime}, E_s^{\sharp\prime}\rangle, is^{\sharp\prime}\rangle$ defined by the following:

$$
\begin{aligned}
f[E_v^\sharp](n_W) &= \begin{cases} n_{W\cup\{x\}} & [x, n_W] \in E_v^\sharp \\ n_W & \text{otherwise} \end{cases} \\
E_v^{\sharp\prime} &= \{[z, n_Z] \mid [z, n_Z] \in E_v^\sharp \wedge z \neq x\} \cup \{[z, f[E_v^\sharp](n_Z)] \mid [z, n_Z] \in E_v^\sharp\} \\
E_s^{\sharp\prime} &= \begin{array}{l} E_s^\sharp \;\cup\; \{\langle f[E_v^\sharp](n_V), sel', n_W\rangle \mid \langle n_V, sel', n_W\rangle \in E_s^\sharp\} \\ \quad\;\cup\; \{\langle n_V, sel', f[E_v^\sharp](n_W)\rangle \mid \langle n_V, sel', n_W\rangle \in E_s^\sharp\} \\ \quad\;\cup\; \{\langle f[E_v^\sharp](n_V), sel', f[E_v^\sharp](n_W)\rangle \mid \langle n_V, sel', n_W\rangle \in E_s^\sharp\} \end{array} \\
is^{\sharp\prime}(n_W) &= \begin{cases} is^\sharp(n_W) \vee is^\sharp(n_{W-\{x\}}) & x \in W \wedge [x, n_{W-\{x\}}] \in E_v^\sharp \\ is^\sharp(n_W) & \text{otherwise} \end{cases}
\end{aligned}
$$

The narrowing operation materializes at most $|E_v(x)|$ shape-nodes and guarantees that x is in the name of all the shape-nodes that x points to. This permits the interpretation of $x.sel :=$ **nil** to nullify the *sel* field of all shape-nodes that x points to.

Example 6.2.2.1. Applying this narrowing operation to the SSG in Example 6.2.1.1 yields back the SSG shown in the v_2 box of Figure 11. In general, however, narrowing a widened graph may yield an SSG that is less precise than the original SSG.

6.3 Refining the Concrete Semantics

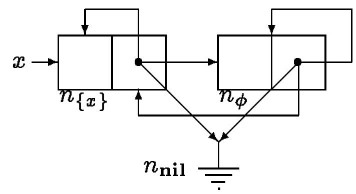
In this article, we have tried to simplify the presentation of our ideas both by choosing a small programming language and by using a "concrete semantics" that has a small amount of abstraction built into it already. The following are some possible refinements of the concrete semantics that would lead to slightly different abstract semantics (with somewhat different powers):

—Shape-nodes that are not reachable from variables are not removed by the operational semantics shown in Figure 6. For certain programs, this may lead to loss of accuracy. This can be overcome by working with concrete and abstract semantics that incorporate garbage-collection operations (see Sagiv et al. [1995]).

—In the way shape graphs were defined in Section 3.2, there are no shape-graph elements that represent uninitialized fields of cons-cells or fields whose value is either an atom or **nil**. One consequence of this is that the shape-analysis algorithm is only able determine rather weak data-type properties. As pointed out in Example 5.2.6, when the algorithm reports that a variable points to a circular list, it may actually point only to a noncircular 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 shape-graphs that all fields of shape-nodes have *at least one* outgoing selector-edge (possibly to $n_{\mathbf{atom}}$, $n_{\mathbf{nil}}$, or $n_{\mathbf{uninit}}$). This modified domain of SSGs is capable of characterizing some *definitely cyclic* data structures.

For example, with this extended definition of shape-graphs, the SSG shown in Figure 7(d) characterizes the definitely cyclic lists of length ≥ 2 (modulo the absence of edges from the *car* fields to $n_{\mathbf{atom}}$ in the two shape-nodes); Figure 7(e) characterizes the definitely cyclic lists of length ≥ 1. The *possibly* cyclic lists of length ≥ 1 are characterized by the following SSG:



6.4 Interprocedural Analysis

The shape-analysis algorithm can also be extended to handle procedure calls. Two fundamental problems need to be resolved:

(1) Representing multiple occurrences of the same local variable in (mutually) recursive procedures.

(2) 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 cons-cells of a single concrete store. 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 [1981]. 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.

An alternative is to use Hendren's tabulation method for interprocedural analysis [Hendren 1990].

7. APPLICATIONS

The algorithm developed in Section 5 produces an SSG $SG_v^\natural$ for every program point v. This SSG provides an approximation to the set of stores (DSGs) that can possibly occur in any execution of the program that ends at v. Therefore, many interesting questions about the stores at v can be answered (conservatively) by investigating $SG_v^\natural$. In this section, we present two such applications of SSGs. The information that these techniques provide is useful both in optimizing compilers and in software engineering tools.

7.1 Finding May-Aliases

We say that two access paths, e_1 and e_2, are *may-aliases* at a program point v if there exists an execution sequence leading to v that produces a store in which both e_1 and e_2 point to the same cons-cell. The may-alias problem is a fundamental problem in optimizing compilers generating code for scalar, superscalar, and parallel architectures. It is also useful in software engineering tools.

A special case of the may-alias problem concerns whether two pointer *variables* x and y are may-aliases just before a given program point v. It is possible to use the results of our shape-analysis algorithm to give a conservative answer to this question by testing whether x and y point to a common shape-node in the SSG $SG^\sharp_v$. If x and y do not point to a common shape-node, we conclude that they cannot be may-aliases; otherwise, we conservatively conclude that there may exist an execution sequence in which they point to the same cons-cell.[5]

The more general complex question is whether two *access paths* e_1 and e_2 are may-aliases. There are several different possible tests:

—A simple test is to check if there are common shape-nodes accessible from both e_1 and e_2. If there are no such nodes, we can conclude that e_1 and e_2 are definitely not may-aliases. Otherwise, we conservatively conclude that e_1 and e_2 are may-aliases.

Example. In the v_2 box of Figure 11, this test yields that $x.cdr$ and $y.cdr$ are may-aliases. This is obviously a superfluous may-aliases because n_ϕ is not shared.

—Sharing information can be used to reduce the number of superfluous may-aliases reported for two given access paths. The main idea is that directed paths in $SG^\sharp_v$ that lead to unshared nodes do not create aliases. To test whether e_1 and e_2 are may-aliases just before program-point v, we test whether there exist two (possibly empty) directed selector-paths π_1 and π_2 in $SG^\sharp_v$ leading to a common SSG node such that the following conditions hold:

(1) For every node n_X that appears more than once along π_1 (resp., π_2), either $X = \phi$ or $is(n_X)$.
(2) There exists a possibly empty common suffix Δ of e_1 and e_2 (i.e., $e_1 = e'_1\Delta$ and $e_2 = e'_2\Delta$) such that e'_1 and e'_2 lead to a common node n and either (1) e'_1 and e'_2 is a simple variable or (2) $is(n) = true$.

Example. In the v_2 box of Figure 11, none of the shape nodes are shared. Therefore, the only detected may-aliases are the ones induced by variables, e.g., $y.cdr.cdr$ and $t.cdr$ are may-aliases.

—The node-compatibility conditions can be used to trim out some additional superfluous aliases. The main idea is that directed paths that go through different shape-nodes that have common variables in their name can only come from different DSGs. Therefore, we can safely say that these kind of paths do not indicate may-aliases.

[5] Similarly, it is possible to use the results of our shape-analysis algorithm to give a conservative answer to the question of whether two pointer variables x and y are must-aliases just before a given program point v: if every shape-node with x in its name also has y in its name, and vice versa, then we conclude that they are must-aliases.

7.2 Detecting Shared Data Structures

SSGs can also be used 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. The sharing problem is a natural generalization of the may-aliases problem in which we quantify over access paths. Therefore, the three aforementioned approaches can be adapted to provide a solution to the sharing problem. For example, the second approach implies that if all shape-nodes n accessible from both x and y have the value $is^\sharp(n) = \mathit{false}$, there cannot possibly be any sharing between the data structures pointed to by x and y.

8. RELATED WORK

The shape-analysis problem was first investigated by Reynolds [1968], who studied it in the context of a Lisp-like language with no destructive updating. Reynolds treated the problem as one of simplifying a collection of set equations. A similar shape-analysis problem, but for an imperative language supporting nondestructive 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 [Jones and Muchnick 1981]. Follow-on work on this kind of shape-analysis problem includes Mogensen [1988; 1989], Heintze [1992], and Reps [1995].

Jones and Muchnick [1981], 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.[6] 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 Jones-Muchnick formulation has two drawbacks:

(1) The analysis yields poor results for programs that manipulate cons-cells beyond the k-horizon. For example, in the list-reversal program of Figure 5, 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 is treated conservatively, as an arbitrary, and possibly cyclic, data structure.

(2) 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., Horwitz et al.[1989]).

Whereas Jones and Muchnick [1981] use *sets* of shape-graphs, our work follows Jones and Muchnick [1982], Larus [1989], Larus and Hilfinger [1988], Chase et al. [1990], and Stransky [1992] who developed shape-analysis methods that associate each program point with a *single* shape-graph. The use of a single shape-graph is

[6] 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.

possibly less accurate than a method based on sets of graphs, and, compared with a method that uses sets of graphs, working with a single graph complicates the abstract semantics. However, for reasons mentioned in Section 5.2, the use of a single graph seems necessary for the pragmatic reason that it is more likely to lead to a practical shape-analysis algorithm.

Jones and Muchnick [1982], Chase at el. [1990], and Stransky [1992] 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 [Chase et al. 1990] and from an implementation perspective [Assmann and Weinhardt 1993].

The algorithm presented by Chase et al. [1990] is based on the following ideas:

— Sharing information, in the form of abstract heap reference counts (0, 1, and ∞), is used to characterize shape-graphs that represent list structures.[7]
— Several heuristics are introduced to allow several shape-nodes to be maintained for each allocation site.
— For an assignment to $x.sel$, when the shape-node that x points to represents only concrete cons-cells that will definitely be overwritten, the *sel* field of the shape-node that x points to can be 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 "listness." However, as noted by Chase et al. [1990], allocation-site information alone is insufficient to determine interesting facts in many programs. For example, it cannot determine that "listness" is preserved for either the list-reversal program or the list-insert program. In particular, in the list-reversal 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 might share cons-cells in common.

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

(1) *Tracking of aliasing configurations.* The sets of variable names attached to shape-nodes track possible aliasing configurations. A shape-node n_Z represents cons-cells that are simultaneously pointed to by exactly the variables in Z. The abstract semantics tracks possible aliasing configurations by performing operations on the variable sets that name SSG shape-nodes.

(2) *"Strong nullification."* For an assignment of the form $x.sel := y$, the Chase-Wegman-Zadeck method ordinarily performs a "weak update" (i.e., selector-edges emanating from the shape-nodes that 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 3.1, an assignment statement $x.sel := y$ is transformed into two statements: $x.sel :=$ **nil**, followed immediately by $x.sel := y$. When our algorithm processes the first of these statements, it (always) removes the *sel* edges emanating from the shape-nodes that x points to. We have called this operation "strong nullifi-

[7]The idea of augmenting shape-graphs with sharing information also appears in the earlier work of Jones and Muchnick [1981].

cation," by analogy with "strong update." When the algorithm processes the second statement, it introduces *sel* edges that emanate from the shape-nodes that x points to. With these taken together, the effect is to *overwrite* the *sel* edges emanating from the shape-nodes that x points to. In other words, for a statement in the original program of the form $x.sel := y$, our algorithm *always performs a strong update*, even when x does not point to a unique SSG shape-node.

(3) *Materialization.* In an assignment statement of the form $x := y.sel$, our algorithm materializes new shape-nodes that conservatively cover all the possible new configurations of variable sets whose members all point to the same cons-cell. For example, when $y.sel$ points to n_ϕ, our algorithm materializes a new node $n_{\{x\}}$ out of n_ϕ. Furthermore, if $is^\sharp(n_\phi) = \mathit{false}$, this information is used to exclude both of the two possible selector-edges from n_ϕ to $n_{\{x\}}$, as well as both of the two possible selector-edges from $n_{\{x\}}$ to $n_{\{x\}}$. 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.

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 [Chase et al. 1990, p. 309]).

Chase, Wegman, and Zadeck use reference-count values 0, 1, and ∞, whereas we use a Boolean-valued function $is^\sharp$. 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 $is^\sharp(n) = \mathit{false}$).

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 5.1.

Plevyak et al. [1993] describe a shape-analysis algorithm that is similar to the Chase-Wegman-Zadeck method. Their algorithm inherits most of the drawbacks of the original Chase-Wegman-Zadeck algorithm.

Larus and Hilfinger [Larus 1989; Larus and Hilfinger 1988] 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 degenerate 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 "listness."

There are also several algorithms that are not based on shape-graphs for finding may-alias information for pointer variables. The most sophisticated ones are those of Landi and Ryder [1991] and Deutsch [1994]. 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, for the list-reversal program, Deutsch's algorithm yields may-alias information that is equivalent to that produced by the algorithm of Section 5.1. However, both the Landi-Ryder and Deutsch algorithms do not determine that either "listness" or "circular listness" is preserved by the insert program of Figure 15. 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 14(b) cuts the list pointed to by x (see Figures 15, the second and the third rows)). (We do not mean to imply that our method always dominates the Landi-Ryder and Deutsch algorithms; we know of at least one program for which Deutsch's algorithm is more accurate than our algorithm.)

A different approach was taken by Hendren and Nicolau, who designed an algorithm that handles only acyclic data structures [Hendren 1990; Hendren and Nicolau 1990]. Because of the decision to work with programs that only manipulate acyclic structures, the algorithm does not have to have a way of representing cycles, even conservatively. For this alias-analysis problem, they have given an efficient algorithm that manipulates matrices that record access paths that are aliases.

To the best of our knowledge, the Hendren-Nicolau 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, their algorithm cannot determine such structure-preservation properties for programs that handle cyclic lists.

Myers [1981] presented an algorithm for interprocedural bit-vector problems that accounts for aliasing. 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 [Landi 1992;

Ramalingam 1994].

ACKNOWLEDGEMENTS

We are grateful for the helpful comments of Martin Alt, Alain Deutsch, Christian Fecht, John Field, Neil Jones, Yishay Mansour, Florian Martin, Mike O'Donnell, G. Ramalingam, and Michael Rodeh. Laurie Hendren provided us with extensive and very helpful information about the capabilities of her shape-analysis technique.

We used two logic-programming systems—Coral [Ramakrishnan et al. 1993] and XSB [Warren 1992]—to experiment with implementations of the shape-analysis algorithm. We are indebted to Raghu Ramakrishnan and Praveen Seshadri for help with Coral and to C.R. Ramakrishnan for porting the Coral implementation to XSB.

REFERENCES

ASSMANN, U. AND WEINHARDT, M. 1993. Interprocedural heap analysis for parallelizing imperative programs. In *Programming Models For Massively Parallel Computers*, W. K. Giloi, S. Jähnichen, and B. D. Shriver, Eds. IEEE Computer Society, Washington, D.C., 74–82.

CHASE, D., WEGMAN, M., AND ZADECK, F. 1990. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation.* ACM Press, New York, 296–310.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM Press, New York, 232–245.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM Press, New York, 238–252.

DEUTSCH, A. 1992. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE International Conference on Computer Languages.* IEEE Computer Society, Washington, D.C., 2–13.

DEUTSCH, A. 1994. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation.* ACM Press, New York, 230–241.

GHIYA, R. AND HENDREN, L. 1996. Is it a tree, a dag, or a cyclic graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM Press, New York. Available via ftp://ftp-acaps.cs.mcgill.ca/pub/doc/papers/POPL96.ps.gz.

HEINTZE, N. 1992. Set-based program analysis. Ph.D. thesis, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

HENDREN, L. 1990. Parallelizing programs with recursive data structures. Ph.D. thesis, Cornell Univ., Ithaca, N.Y.

HENDREN, L. AND GAO, G. 1992. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the International Conference on Computer Languages.* IEEE Computer Society, Washington, D.C., 242–251.

HENDREN, L., HUMMEL, J., AND NICOLAU, A. 1992. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conference on Programming Languages Design and Implementation.* ACM Press, New York, 249–260.

HENDREN, L. AND NICOLAU, A. 1990. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst. 1,* 1 (Jan.), 35–47.

HORWITZ, S., PFEIFFER, P., AND REPS, T. 1989. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation.* ACM Press, New York, 28–40.

JONES, N. AND MUCHNICK, S. 1981. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., Chapter 4, 102–131.

JONES, N. AND MUCHNICK, S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 66–74.

KNUTH, D. E. 1973. *The Art of Computer Programming*, 2nd. ed. Vol. 1: *Fundamental Algorithms*. Addison-Wesley, Reading, Mass.

LANDI, W. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1,* 4, 323–337.

LANDI, W. AND RYDER, B. 1991. Pointer induced aliasing: A problem classification. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 93–103.

LARUS, J. 1989. Restructuring symbolic programs for concurrent execution on multiprocessors. Ph.D. thesis, Univ. of California, Berkeley, Calif.

LARUS, J. AND HILFINGER, P. 1988. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*. ACM Press, New York, 21–34.

LINDSTROM, G. 1973. Scanning list structures without stacks or tag bits. *Inf. Process. Lett. 2,* 2 (June), 47–51.

MOGENSEN, T. 1988. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation* (Gammel Avernaes, Denmark, Oct 18-24, 1987). North-Holland, Amsterdam, 325–347.

MOGENSEN, T. 1989. Separating binding times in language specifications. In *Proceedings of the 4th International Conference on Functional Programming and Computer Architecture*. ACM Press, New York, 12–25.

MYERS, E. 1981. A precise inter-procedural data flow algorithm. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 219–230.

PLEVYAK, J., CHIEN, A., AND KARAMCHETI, V. 1993. Analysis of dynamic structures for efficient parallel execution. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 768. Springer-Verlag, Berlin, 37–57.

RAMAKRISHNAN, R., SESHADRI, P., SRIVASTAVA, D., AND SUDARSHAN, S. 1993. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD 93 Conference*. ACM Press, New York, 167–176.

RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16,* 5, 1467–1471.

REPS, T. 1995. Shape analysis as a generalized path problem. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*. ACM Press, New York, 1–11.

REYNOLDS, J. 1968. Automatic computation of data set definitions. In *Information Processing 68: Proceedings of the IFIP Congress*. North-Holland, Amsterdam, 456–461.

SAGIV, M., REPS, T., AND WILHELM, R. 1995. Solving shape-analysis problems in languages with destructive updating. Tech. Rep. TR-1276, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wisc. July. Available via http://www.cs.wisc.edu/trs.html.

SEDGEWICK, R. 1988. *Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J., 189–234.

STRANSKY, J. 1992. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. Comput. 101,* 1 (Nov.), 70–102.

WARREN, D. 1992. Memoing for logic programs. *Commun. ACM 35,* 3 (Mar.), 93–111.

Received August 1996; accepted April 1997

APPENDIX

An appendix to this article is available in electronic form (PostScript™). Any of the following methods may be used to obtain it; or see the inside back cover of a current issue for up-to-date instructions.

—By anonymous ftp from **acm.org**, file **[pubs.journals.toplas.append]p1835.ps**

—Send electronic mail to **mailserve@acm.org** containing the line
send [anonymous.pubs.journals.toplas.append]p1835.ps

—By *Gopher* from **acm.org**

—By anonymous ftp from **ftp.cs.princeton.edu**, file **pub/toplas/append/p1835.ps**

—Hardcopy from *Article Express*, for a fee: phone 800-238-3458, fax +1-516-997-0890, or write 469 Union Avenue, Westbury NY 11550; and request ACM-TOPLAS-APPENDIX-1835.

1997

THIS DOCUMENT IS THE APPENDIX TO THE FOLLOWING PAPER:

Solving Shape-Analysis Problems in Languages with Destructive Updating

MOOLY SAGIV
Tel-Aviv University
and
THOMAS REPS
University of Wisconsin
and
REINHARD WILHELM
Universität des Saarlandes

A. PROOF OF THREE SAFE-APPROXIMATION PROPERTIES

In the proof of Theorem 5.3.6, the case for statements of the form "$x := y.sel$" (Lemma B.1) relies on the fact that the abstract predicates $compat\_in^\sharp$, $compat\_self^\sharp$, and $compat\_out^\sharp$ are a safe approximation of the corresponding concrete properties *compat_in*, *compat_self*, and *compat_out* (see Figure 12). In the following lemma, we prove that.

Lemma A.1.

(1) $compat\_in \Rightarrow_\beta compat\_in^\sharp$
(2) $compat\_self \Rightarrow_\beta compat\_self^\sharp$
(3) $compat\_out \Rightarrow_\beta compat\_out^\sharp$

Proof. Let $SG = \langle E_v, E_s \rangle$ be a DSG in $\mathcal{DSG}$; let β denote $\beta[E_v]$; and let iis denote $iis[E_s]$.

Proof of (1). Suppose that l_1, l_2, and l_3 are shape-nodes in $shape\_nodes(SG)$ such that $compat\_in([y, l_1], \langle l_1, sel, l_2 \rangle, \langle l_3, sel', l_2 \rangle)$ holds. By definition, we have $[y, l_1] \in E_v$, $\langle l_1, sel, l_2 \rangle \in E_s$, $\langle l_3, sel', l_2 \rangle \in E_s$, and $l_3 \neq l_2$. Using Definitions 5.2.1

and 5.2.2 and Lemma 5.3.1, we have

$$\begin{array}{rcll}
true & \Rightarrow & compatible(l_1, l_2, l_3) & \text{Figure 12} \\
 & \Rightarrow & compatible^\sharp(\beta(l_1), \beta(l_2), \beta(l_3)) & \text{Lemma 5.3.1 (1)} \\
[y, l_1] \in E_v & \Rightarrow & [y, \beta(l_1)] \in \beta(E_v) & \text{Definition 5.2.1} \\
\langle l_1, sel, l_2 \rangle \in E_s & \Rightarrow & \langle \beta(l_1), sel, \beta(l_2) \rangle \in \beta(E_s) & \text{Definition 5.2.1} \\
\langle l_3, sel', l_2 \rangle \in E_s & \Rightarrow & \langle \beta(l_3), sel', \beta(l_2) \rangle \in \beta(E_s) & \text{Definition 5.2.1} \\
l_3 \neq l_2 & \Rightarrow & \beta(l_2) \neq^\sharp \beta(l_3) & \text{Lemma 5.3.1 (3)} \\
\begin{array}{r} \langle l_1, sel, l_2 \rangle \in E_s, \\ \langle l_3, sel', l_2 \rangle \in E_s \end{array} & \Rightarrow & \begin{array}{l} \langle l_1, sel, l_2 \rangle \in E_s, \\ \langle l_3, sel', l_2 \rangle \in E_s, \\ (\neg iis(l_2) \vee iis(l_2)) \end{array} & \begin{array}{l} true \Rightarrow \\ (\neg iis(l_2) \vee iis(l_2)) \end{array} \\
 & \Rightarrow & (l_1 = l_3 \wedge sel = sel') \vee iis(l_2) & \begin{array}{l} \langle l_1, sel, l_2 \rangle \in E_s, \\ \langle l_3, sel', l_2 \rangle \in E_s, \\ \text{and Definition 5.2.2} \end{array} \\
 & \Rightarrow & (\beta(l_1) =^\sharp \beta(l_3) \wedge sel = sel') \vee iis(l_2) & \text{Lemma 5.3.1 (2)} \\
 & \Rightarrow & (\beta(l_1) =^\sharp \beta(l_3) \wedge sel = sel') \vee is^\sharp(\beta(l_2)) & \text{Definition 5.2.1}
\end{array}$$

Therefore, $compat\_in^\sharp([y, \beta(l_1)], \langle \beta(l_1), sel, \beta(l_2) \rangle, \langle \beta(l_3), sel', \beta(l_2) \rangle)$ holds.

Proof of (2). Suppose that l_1 and l_2 are shape-nodes in $shape\_nodes(SG)$ such that
$compat\_self([y, l_1], \langle l_1, sel, l_2 \rangle, \langle l_2, sel', l_2 \rangle)$ holds, i.e., $[y, l_1] \in E_v$, $\langle l_1, sel, l_2 \rangle \in E_s$, and $\langle l_2, sel', l_2 \rangle \in E_s$. Using Definitions 5.2.1 and 5.2.2 and Lemma 5.3.1, we have

$$\begin{array}{rcll}
true & \Rightarrow & compatible(l_1, l_2) & \text{Figure 12} \\
 & \Rightarrow & compatible^\sharp(\beta(l_1), \beta(l_2)) & \text{Lemma 5.3.1 (1)} \\
[y, l_1] \in E_v & \Rightarrow & [y, \beta(l_1)] \in \beta(E_v) & \text{Definition 5.2.1} \\
\langle l_1, sel, l_2 \rangle \in E_s & \Rightarrow & \langle \beta(l_1), sel, \beta(l_2) \rangle \in \beta(E_s) & \text{Definition 5.2.1} \\
\langle l_2, sel', l_2 \rangle \in E_s & \Rightarrow & \langle \beta(l_2), sel', \beta(l_2) \rangle \in \beta(E_s) & \text{Definition 5.2.1} \\
\begin{array}{r} \langle l_1, sel, l_2 \rangle \in E_s, \\ \langle l_2, sel', l_2 \rangle \in E_s \end{array} & \Rightarrow & \begin{array}{l} \langle l_1, sel, l_2 \rangle \in E_s, \\ \langle l_2, sel', l_2 \rangle \in E_s, \\ (\neg iis(l_2) \vee iis(l_2)) \end{array} & \begin{array}{l} true \Rightarrow \\ (\neg iis(l_2) \vee iis(l_2)) \end{array} \\
 & \Rightarrow & (l_1 = l_2 \wedge sel = sel') \vee iis(l_2) & \begin{array}{l} \langle l_1, sel, l_2 \rangle \in E_s, \\ \langle l_2, sel', l_2 \rangle \in E_s, \\ \text{and Definition 5.2.2} \end{array} \\
 & \Rightarrow & (\beta(l_1) =^\sharp \beta(l_2) \wedge sel = sel') \vee iis(l_2) & \text{Lemma 5.3.1 (2)} \\
 & \Rightarrow & (\beta(l_1) =^\sharp \beta(l_2) \wedge sel = sel') \vee is^\sharp(\beta(l_2)) & \text{Definition 5.2.1}
\end{array}$$

Therefore, $compat\_self^\sharp([y, \beta(l_1)], \langle \beta(l_1), sel, \beta(l_2) \rangle, \langle \beta(l_2), sel', \beta(l_2) \rangle)$ holds.

Proof of (3). Suppose that l_1, l_2, and l_3 are shape-nodes in $shape\_nodes(SG)$ such that
$compat\_out([y, l_1], \langle l_1, sel, l_2 \rangle, \langle l_2, sel', l_3 \rangle)$ holds, i.e., $[y, l_1] \in E_v$, $\langle l_1, sel, l_2 \rangle \in E_s$, $\langle l_2, sel', l_3 \rangle \in E_s$, and $l_3 \neq l_2$. Using Definitions 5.2.1 and 5.2.2 and Lemma 5.3.1, we have

$$\begin{array}{rcll}
true & \Rightarrow & compatible(l_1, l_2, l_3) & \text{Figure 12} \\
 & \Rightarrow & compatible^\sharp(\beta(l_1), \beta(l_2), \beta(l_3)) & \text{Lemma 5.3.1 (1)} \\
[y, l_1] \in E_v & \Rightarrow & [y, \beta(l_1)] \in E_v^\sharp & \text{Definition 5.2.1} \\
\langle l_1, sel, l_2 \rangle \in E_s & \Rightarrow & \langle \beta(l_1), sel, \beta(l_2) \rangle \in E_s^\sharp & \text{Definition 5.2.1} \\
\langle l_2, sel', l_3 \rangle \in E_s & \Rightarrow & \langle \beta(l_2), sel, \beta(l_3) \rangle \in E_s^\sharp & \text{Definition 5.2.1} \\
l_3 \neq l_2 & \Rightarrow & \beta(l_3) \neq^\sharp \beta(l_2) & \text{Lemma 5.3.1 (3)}
\end{array}$$

The restrictions on the number of outgoing selector-edges that a field of a DSG shape-node can have, given in Definition 3.2.2, can be expressed in alternative form as follows:

OBSERVATION A.2. *Suppose* $\langle c_1, sel_1, c_3\rangle$ *and* $\langle c_2, sel_2, c_4\rangle$ *are selector-edges in* E_s *such that* $\langle c_1, sel_1, c_3\rangle \neq \langle c_2, sel_2, c_4\rangle$. *Then* $c_1 \neq c_2 \vee sel_1 \neq sel_2$.

In our case, we have

$$\begin{array}{lll} \langle l_1, sel, l_2\rangle \in E_s, & & \\ \langle l_2, sel', l_3\rangle \in E_s, & \Rightarrow \langle l_1, sel, l_2\rangle \neq \langle l_2, sel', l_3\rangle & \\ l_3 \neq l_2 & & \\ & \Rightarrow l_1 \neq l_2 \vee sel \neq sel' & \text{Observation A.2} \\ & \Rightarrow \beta(l_1) \neq^\sharp \beta(l_2) \vee sel \neq sel' & \text{Lemma 5.3.1 (3)} \end{array}$$

Combined with the properties shown above, this shows that

$$compat\_out^\sharp([y, \beta(l_1)], \langle\beta(l_1), sel, \beta(l_2)\rangle, \langle\beta(l_2), sel', \beta(l_3)\rangle)$$

holds.

B. LOCAL SAFETY OF THE ABSTRACT SEMANTICS

In this section, we show that the abstract semantics of static shape-graphs is safe with respect to the concrete semantics.

THEOREM 5.3.6(LOCAL SAFETY THEOREM). *For all assignment statements st, and for every* $SG \in \mathcal{DSG}$, $\beta([\![st]\!](SG)) \sqsubseteq [\![st]\!]^\sharp(\beta(SG))$.

PROOF. Let $SG = \langle E_v, E_s\rangle$, and define $\langle E'_v, E'_s\rangle$, $\langle\langle E^\sharp_v, E^\sharp_s\rangle, is^\sharp\rangle$, and $\langle\langle E^{\sharp\prime}_v, E^{\sharp\prime}_s\rangle, is^{\sharp\prime}\rangle$ as follows (in accordance with the diagram shown in Figure 18):

$$\langle E'_v, E'_s\rangle \stackrel{\text{def}}{=} [\![st]\!](\langle E_v, E_s\rangle) \tag{3}$$
$$\langle\langle E^\sharp_v, E^\sharp_s\rangle, is^\sharp\rangle \stackrel{\text{def}}{=} \beta[E_v](\langle E_v, E_s\rangle) \tag{4}$$
$$\langle\langle E^{\sharp\prime}_v, E^{\sharp\prime}_s\rangle, is^{\sharp\prime}\rangle \stackrel{\text{def}}{=} [\![st]\!]^\sharp(\langle\langle E^\sharp_v, E^\sharp_s\rangle, is^\sharp\rangle) \tag{5}$$

Throughout the rest of the proof, $\langle E_v, E_s\rangle$ and $\langle E'_v, E'_s\rangle$ are understood. We will use π as a shorthand for $\pi[E_v]$ and π' as a shorthand for $\pi[E'_v]$. Similarly, β is a shorthand for $\beta[E_v]$, and β' is a shorthand for $\beta[E'_v]$.

We need to show that $\beta'(\langle E'_v, E'_s\rangle) \sqsubseteq \langle\langle E^{\sharp\prime}_v, E^{\sharp\prime}_s\rangle, is^{\sharp\prime}\rangle$. This amounts to showing the following:

$$[z, l] \in E'_v \Rightarrow \beta'([z, l]) \in E^{\sharp\prime}_v \tag{6}$$
$$\langle l, sel', l'\rangle \in E'_s \Rightarrow \beta'(\langle l, sel', l'\rangle) \in E^{\sharp\prime}_s \tag{7}$$
$$iis[E'_s](l) \Rightarrow is^{\sharp\prime}(\beta'(l)) \tag{8}$$

In some cases it is more convenient to use the set form of (7), i.e.,

$$\beta'(E'_s) \subseteq E^{\sharp\prime}_s \tag{9}$$

The cases of $st \equiv x := \mathbf{nil}$, $st \equiv x := \mathbf{new}$, and $st \equiv x := y$ are left to the reader. The cases of $st \equiv x := y.sel$, $st \equiv x.sel := \mathbf{nil}$, and $st \equiv x.sel := y$ are shown in Lemmas B.1, B.4, and B.6, respectively. In all cases, the direction of the argument of the proof follows the arrow shown in Figure 18.

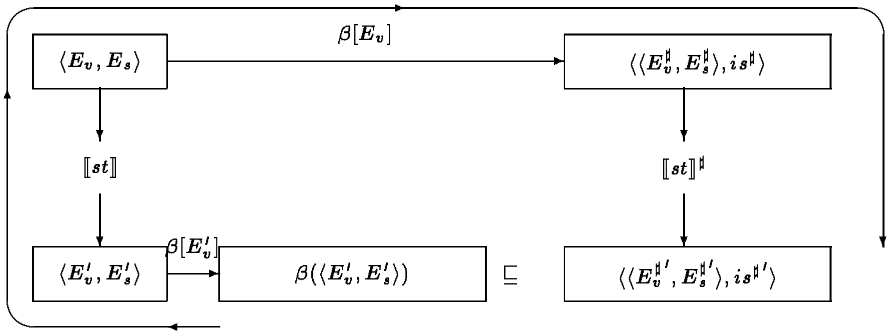


Fig. 18. The direction of the proof of the safety relationship between the DSG meaning function and the SSG meaning function.

LEMMA B.1. $\beta([\![x := y.sel]\!](SG)) \sqsubseteq [\![x := y.sel]\!]^\sharp(\beta(SG))$.

PROOF. Let us define $X_{y.sel}$ to be the cons-cell pointed to by $y.sel$ in $\langle E_v, E_s\rangle$, i.e.,

$$X_{y.sel} \stackrel{\text{def}}{=} \{l \mid [y, l_y] \in E_v, \langle l_y, sel, l\rangle \in E_s\} \tag{10}$$

Because $\langle E_v, E_s\rangle$ is a DSG, $X_{y.sel}$ is either the empty set or a singleton set. We observe that the operational semantics of $x := y.sel$, defined in Figure 6, guarantees that the following observations hold:

OBSERVATION B.2. *For every* $l \in shape\_nodes(\langle E_v, E_s\rangle) - X_{y.sel}$,

$$\begin{aligned} [z, l] \in E_v' &\iff [z, l] \in E_v && (11)\\ \beta'(l) &= \beta(l). && (12)\end{aligned}$$

OBSERVATION B.3. *For* $l \in X_{y.sel}$, $\pi'(l) = \pi(l) \cup \{x\}$.

We now show that (6), (7), and (8) hold for $st \equiv x := y.sel$.

Part I. In this part we show that (6) holds. Let $[z, l] \in E_v'$. There are two cases to consider:

Case 1: $l \notin X_{y.sel}$.

$$\begin{aligned} [z, l] \in E_v' &\Rightarrow [z, l] \in E_v && \text{Observation B.2, (11)}\\ &\Rightarrow \beta([z, l]) \in E_v^\sharp && \text{Definition 5.2.1}\\ &\Rightarrow [z, \beta(l)] \in E_v^\sharp && \text{Definition 5.2.1}\\ &\Rightarrow [z, \beta(l)] \in E_v^{\sharp\prime} && \text{Figure 9}\\ &\Rightarrow [z, \beta'(l)] \in E_v^{\sharp\prime} && \text{Observation B.2, (12)}\\ &\Rightarrow \beta'([z, l]) \in E_v^{\sharp\prime} && \text{Definition 5.2.1}\end{aligned}$$

Case 2: $l \in X_{y.sel}$. By (10), there exist $[y, l_y] \in E_v$ and $\langle l_y, sel, l\rangle \in E_s$. Because $[y, l_y] \in E_v$, we have

$$\begin{aligned} [y, l_y] \in E_v &\Rightarrow \beta([y, l_y]) \in E_v^\sharp && \text{Definition 5.2.1}\\ &\Rightarrow [y, \beta(l_y)] \in E_v^\sharp && \text{Definition 5.2.1}\end{aligned}$$

Because $\langle l_y, sel, l\rangle \in E_s$, we have

$$\begin{aligned} \langle l_y, sel, l\rangle \in E_s &\Rightarrow \beta(\langle l_y, sel, l\rangle) \in E_s^\sharp && \text{Definition 5.2.1}\\ &\Rightarrow \langle \beta(l_y), sel, \beta(l)\rangle \in E_s^\sharp && \text{Definition 5.2.1}\\ &\Rightarrow \langle \beta(l_y), sel, n_{\pi(l)}\rangle \in E_s^\sharp && \text{Definition 5.2.1}\end{aligned}$$

There are two subcases to consider:

Case 2.1: $[z,l] \in E_v$.

$$\begin{array}{lll}
[z,l] \in E_v & \Rightarrow \beta([z,l]) \in E_v^\sharp & \text{Definition 5.2.1} \\
 & \Rightarrow [z,\beta(l)] \in E_v^\sharp & \text{Definition 5.2.1} \\
 & \Rightarrow [z,n_{\pi(l)}] \in E_v^\sharp & \text{Definition 5.2.1} \\
 & \Rightarrow [z,n_{\pi(l)\cup\{x\}}] \in {E_v^\sharp}' & [y,\beta(l_y)] \in E_v^\sharp \wedge \langle \beta(l_y), sel, n_{\pi(l)} \rangle \in E_s^\sharp \wedge \text{Figure 9} \\
 & \Rightarrow [z,n_{\pi'(l)}] \in {E_v^\sharp}' & \text{Observation B.3} \\
 & \Rightarrow [z,\beta'(l)] \in {E_v^\sharp}' & \text{Definition 5.2.1} \\
 & \Rightarrow \beta'([z,l]) \in {E_v^\sharp}' & \text{Definition 5.2.1}
\end{array}$$

Case 2.2: $[z,l] \in E'_v - E_v$.

$$\begin{array}{lll}
[z,l] \in E'_v - E_v & \Rightarrow z = x & \text{Figure 6} \\
 & \Rightarrow [x,n_{\pi(l)\cup\{x\}}] \in {E_v^\sharp}' & \begin{array}{l} \quad [y,\beta(l_y)] \in E_v^\sharp \\ \wedge\ \langle \beta(l_y), sel, n_{\pi(l)} \rangle \in E_s^\sharp \\ \wedge\ \text{Figure 9} \end{array} \\
 & \Rightarrow [x,n_{\pi'(l)}] \in {E_v^\sharp}' & \text{Observation B.3} \\
 & \Rightarrow [x,\beta'(l)] \in {E_v^\sharp}' & \text{Definition 5.2.1} \\
 & \Rightarrow \beta'([x,l]) \in {E_v^\sharp}' & \text{Definition 5.2.1}
\end{array}$$

Part II. In this part we show that (7) holds. Let $\langle l, sel', l' \rangle \in E'_s$. We have

$$\begin{array}{lll}
\langle l, sel', l' \rangle \in E'_s & \Rightarrow \langle l, sel', l' \rangle \in E_s & \text{Figure 6} \\
 & \Rightarrow \langle \beta(l), sel', \beta(l') \rangle \in E_s^\sharp & \text{Definition 5.2.1}
\end{array}$$

Therefore, there are four cases to consider:

Case 1: $l, l' \notin X_{y.sel}$. First, observe that because $\langle E_v, E_s \rangle$ is deterministic, it cannot be that $[y,l] \in E_v$ and $sel' = sel$. Therefore, we have

$$\begin{array}{lll}
\begin{array}{l} \langle \beta(l), sel', \beta(l') \rangle \in E_s^\sharp \wedge \\ \neg([y,l] \in E_v \wedge sel' = sel) \end{array} & \Rightarrow \begin{array}{l} \langle n_{\pi(l)}, sel', \beta(l') \rangle \in E_s^\sharp \wedge \\ \neg(y \in \pi(l) \wedge sel' = sel) \end{array} & \text{Definition 5.2.1} \\
 & \Rightarrow \langle n_{\pi(l)}, sel', \beta(l') \rangle \in {E_s^\sharp}' & \text{Figure 9} \\
 & \Rightarrow \langle \beta(l), sel', \beta(l') \rangle \in {E_s^\sharp}' & \text{Definition 5.2.1} \\
 & \Rightarrow \langle \beta'(l), sel', \beta'(l') \rangle \in {E_s^\sharp}' & \text{Observation B.2, (12)} \\
 & \Rightarrow \beta'(\langle l, sel', l' \rangle) \in {E_s^\sharp}' & \text{Definition 5.2.1}
\end{array}$$

Case 2: $l \notin X_{y.sel}$, $l' \in X_{y.sel}$. Let l_y be the unique cons-cell such that $l_y \in E_v(y)$, which must exist because $l' \in X_{y.sel}$.

$$
\begin{array}{lll}
\begin{array}{l}[y, l_y] \in E_v \\ \langle l_y, sel, l' \rangle \in E_s \\ \langle l, sel', l' \rangle \in E_s \\ l' \neq l\end{array} & \Rightarrow\ compat\_in \begin{pmatrix}[y, l_y], \\ \langle l_y, sel, l' \rangle, \\ \langle l, sel', l' \rangle\end{pmatrix} & \text{Figure 12} \\
& \Rightarrow\ compat\_in^\natural \begin{pmatrix}\beta([y, l_y]), \\ \beta(\langle l_y, sel, l' \rangle), \\ \beta(\langle l, sel', l' \rangle)\end{pmatrix} & \text{Lemma A.1 (1)} \\
& \Rightarrow\ compat\_in^\natural \begin{pmatrix}[y, \beta(l_y)], \\ \langle \beta(l_y), sel, \beta(l') \rangle, \\ \langle \beta(l), sel', \beta(l') \rangle\end{pmatrix} & \text{Definition 5.2.1} \\
& \Rightarrow\ compat\_in^\natural \begin{pmatrix}[y, \beta(l_y)], \\ \langle \beta(l_y), sel, n_{\pi(l')} \rangle, \\ \langle \beta(l), sel', n_{\pi(l')} \rangle\end{pmatrix} & \text{Definition 5.2.1} \\
& \Rightarrow\ \langle \beta(l), sel', n_{\pi(l') \cup \{x\}} \rangle \in E_s^{\natural\prime} & \text{Figure 9} \\
& \Rightarrow\ \langle \beta(l), sel', n_{\pi'(l')} \rangle \in E_s^{\natural\prime} & \text{Observation B.3} \\
& \Rightarrow\ \langle \beta(l), sel', \beta'(l') \rangle \in E_s^{\natural\prime} & \text{Definition 5.2.1} \\
& \Rightarrow\ \langle \beta'(l), sel', \beta'(l') \rangle \in E_s^{\natural\prime} & \text{Observation B.2 (12)} \\
& \Rightarrow\ \beta'(\langle l, sel', l' \rangle) \in E_s^{\natural\prime} & \text{Definition 5.2.1}
\end{array}
$$

Case 3: $l \in X_{y.sel}$, $l' \notin X_{y.sel}$. Let l_y be the unique cons-cell such that $l_y \in E_v(y)$, which must exist because $l \in X_{y.sel}$.

$$
\begin{array}{lll}
\begin{array}{l}[y, l_y] \in E_v \\ \langle l_y, sel, l \rangle \in E_s \\ \langle l, sel', l' \rangle \in E_s \\ l' \neq l\end{array} & \Rightarrow\ compat\_out \begin{pmatrix}[y, l_y], \\ \langle l_y, sel, l \rangle, \\ \langle l, sel', l' \rangle\end{pmatrix} & \text{Figure 12} \\
& \Rightarrow\ compat\_out^\natural \begin{pmatrix}\beta([y, l_y]), \\ \beta(\langle l_y, sel, l \rangle), \\ \beta(\langle l, sel', l' \rangle)\end{pmatrix} & \text{Lemma A.1 (3)} \\
& \Rightarrow\ compat\_out^\natural \begin{pmatrix}[y, \beta(l_y)], \\ \langle \beta(l_y), sel, \beta(l) \rangle, \\ \langle \beta(l), sel', \beta(l') \rangle\end{pmatrix} & \text{Definition 5.2.1} \\
& \Rightarrow\ compat\_out^\natural \begin{pmatrix}[y, \beta(l_y)], \\ \langle \beta(l_y), sel, n_{\pi(l)} \rangle, \\ \langle n_{\pi(l)}, sel', \beta(l') \rangle\end{pmatrix} & \text{Definition 5.2.1} \\
& \Rightarrow\ \langle n_{\pi(l) \cup \{x\}}, sel', \beta(l') \rangle \in E_s^{\natural\prime} & \text{Figure 9} \\
& \Rightarrow\ \langle n_{\pi'(l)}, sel', \beta(l') \rangle \in E_s^{\natural\prime} & \text{Observation B.3} \\
& \Rightarrow\ \langle \beta'(l), sel', \beta(l') \rangle \in E_s^{\natural\prime} & \text{Definition 5.2.1} \\
& \Rightarrow\ \langle \beta'(l), sel', \beta'(l') \rangle \in E_s^{\natural\prime} & \text{Observation B.2 (12)} \\
& \Rightarrow\ \beta'(\langle l, sel', l' \rangle \in E_s^{\natural\prime} & \text{Definition 5.2.1}
\end{array}
$$

Case 4: $l, l' \in X_{y.sel}$. Because $\langle E_v, E_s\rangle$ is deterministic $l = l'$. Let l_y be the unique cons-cell such that $l_y \in E_v(y)$, which must exist because $l \in X_{y.sel}$.

$$
\begin{array}{lll}
\begin{array}{l}[y, l_y] \in E_v \\ \langle l_y, sel, l\rangle \in E_s \\ \langle l, sel', l\rangle \in E_s\end{array} & \Rightarrow \mathit{compat\_self}\begin{pmatrix}[y, l_y], \\ \langle l_y, sel, l\rangle, \\ \langle l, sel', l\rangle\end{pmatrix} & \text{Figure 12} \\
& \Rightarrow \mathit{compat\_self}^\sharp\begin{pmatrix}\beta([y, l_y]), \\ \beta(\langle l_y, sel, l\rangle), \\ \beta(\langle l, sel', l\rangle)\end{pmatrix} & \text{Lemma A.1 (2)} \\
& \Rightarrow \mathit{compat\_self}^\sharp\begin{pmatrix}[y, \beta(l_y)], \\ \langle \beta(l_y), sel, \beta(l)\rangle, \\ \langle \beta(l), sel', \beta(l)\rangle\end{pmatrix} & \text{Definition 5.2.1} \\
& \Rightarrow \mathit{compat\_self}^\sharp\begin{pmatrix}[y, \beta(l_y)], \\ \langle \beta(l_y), sel, n_{\pi(l)}\rangle, \\ \langle n_{\pi(l)}, sel', n_{\pi(l)}\rangle\end{pmatrix} & \text{Definition 5.2.1} \\
& \Rightarrow \langle n_{\pi(l)\cup\{x\}}, sel', n_{\pi(l)\cup\{x\}}\rangle \in E_s^{\sharp\prime} & \text{Figure 9} \\
& \Rightarrow \langle n_{\pi'(l)}, sel', n_{\pi'(l)}\rangle \in E_s^{\sharp\prime} & \text{Observation B.3} \\
& \Rightarrow \langle \beta'(l), sel', \beta'(l)\rangle \in E_s^{\sharp\prime} & \text{Definition 5.2.1} \\
& \Rightarrow \beta'(\langle l, sel', l\rangle) \in E_s^{\sharp\prime} & \text{Definition 5.2.1}
\end{array}
$$

Part III. In this part we show that (8) holds. Note that $E_s = E_s'$ and $\mathit{shape\_nodes}(\langle E_v', E_s'\rangle) = \mathit{shape\_nodes}(\langle E_v, E_s\rangle)$. Let $l \in \mathit{shape\_nodes}(\langle E_v, E_s\rangle)$, and suppose that $\mathit{iis}[E_s](l)$ holds. By Definition 5.2.1, $\mathit{is}^\sharp(\beta(l))$ holds. There are two cases to consider:

Case 1: $l \notin X_{y.sel}$.

$$
\begin{array}{lll}
\mathit{is}^\sharp(\beta(l)) & \Rightarrow \mathit{is}^{\sharp\prime}(\beta(l)) & \text{Figure 9} \\
& \Rightarrow \mathit{is}^{\sharp\prime}(\beta'(l)) & \text{Observation B.2 (12)}
\end{array}
$$

Case 2: $l \in X_{y.sel}$. By (10), there exist $[y, l_y] \in E_v$ and $\langle l_y, sel, l\rangle \in E_s$. Because $[y, l_y] \in E_v$, we have

$$
\begin{array}{lll}
[y, l_y] \in E_v & \Rightarrow \beta([y, l_y]) \in E_v^\sharp & \text{Definition 5.2.1} \\
& \Rightarrow [y, \beta(l_y)] \in E_v^\sharp & \text{Definition 5.2.1}
\end{array}
$$

Because $\langle l_y, sel, l\rangle \in E_s$, we have

$$
\begin{array}{lll}
\langle l_y, sel, l\rangle \in E_s & \Rightarrow \beta(\langle l_y, sel, l\rangle) \in E_s^\sharp & \text{Definition 5.2.1} \\
& \Rightarrow \langle \beta(l_y), sel, \beta(l)\rangle \in E_s^\sharp & \text{Definition 5.2.1} \\
& \Rightarrow \langle \beta(l_y), sel, n_{\pi(l)}\rangle \in E_s^\sharp & \text{Definition 5.2.1}
\end{array}
$$

Finally, because $\mathit{is}^\sharp(\beta(l))$ holds, we have

$$
\begin{array}{lll}
\mathit{is}^\sharp(\beta(l)) & \Rightarrow \mathit{is}^\sharp(n_{\pi(l)}) & \text{Definition 5.2.1} \\
& \Rightarrow \mathit{is}^{\sharp\prime}(n_{\pi(l)\cup\{x\}}) & [y, \beta(l_y)] \in E_v^\sharp \wedge \langle \beta(l_y), sel, n_{\pi(l)}\rangle \in E_s^\sharp \wedge \text{Figure 9} \\
& \Rightarrow \mathit{is}^{\sharp\prime}(n_{\pi'(l)}) & \text{Observation B.3} \\
& \Rightarrow \mathit{is}^\sharp(\beta'(l)) & \text{Definition 5.2.1.}\square
\end{array}
$$

LEMMA B.4. $\beta([\![x.sel := \mathbf{nil}]\!](SG)) \sqsubseteq [\![x.sel := \mathbf{nil}]\!]^\sharp(\beta(SG))$.

PROOF. Because the set of variable-edges is unchanged in the transformers for statements of the form $x.sel := \mathbf{nil}$ in both the operational and the abstract semantics, (6) trivially holds. Also, the following observation holds:

Observation B.5. *$E_v' = E_v$ and therefore $\beta' = \beta$.*

We now show that (7) and (8) hold for $st \equiv x.sel :=$ **nil**.

Part I. In this part we show that (7) holds.

$$\begin{array}{rcll}
\langle l, sel', l'\rangle \in E_s' & \Rightarrow & \langle l, sel', l'\rangle \in E_s \wedge \\ & & \neg([x,l] \in E_v \wedge sel' = sel) & \text{Figure 6} \\
& \Rightarrow & \beta(\langle l, sel', l'\rangle) \in E_s^\sharp \wedge \\ & & \neg([x,l] \in E_v \wedge sel' = sel) & \text{Definition 5.2.1} \\
& \Rightarrow & \langle \beta(l), sel', \beta(l')\rangle \in E_s^\sharp \wedge \\ & & \neg([x,l] \in E_v \wedge sel' = sel) & \text{Definition 5.2.1} \\
& \Rightarrow & \langle n_{\pi(l)}, sel', \beta(l')\rangle \in E_s^\sharp \wedge \\ & & \neg([x,l] \in E_v \wedge sel' = sel) & \text{Definition 5.2.1} \\
& \Rightarrow & \langle n_{\pi(l)}, sel', \beta(l')\rangle \in E_s^\sharp \wedge \\ & & \neg(x \in \pi(l) \wedge sel' = sel) & \text{Definition 5.2.1} \\
& \Rightarrow & \langle n_{\pi(l)}, sel', \beta(l')\rangle \in E_s^{\sharp'} & \text{Figure 8} \\
& \Rightarrow & \langle \beta(l), sel', \beta(l')\rangle \in E_s^{\sharp'} & \text{Definition 5.2.1} \\
& \Rightarrow & \langle \beta'(l), sel', \beta'(l')\rangle \in E_s^{\sharp'} & \text{Observation B.5} \\
& \Rightarrow & \beta'(\langle l, sel', l'\rangle \in E_s^{\sharp'} & \text{Definition 5.2.1}
\end{array}$$

Part II. In this part we show that (8) holds. Suppose that $iis[E_s'](l)$ holds. By Figure 8, to prove that $is^{\sharp'}(\beta'(l))$ holds, it is sufficient to show that (a) $is^\sharp(\beta(l))$ holds and (b) $iis^\sharp[E_s^{\sharp'}](\beta'(l))$ holds.

Proof of (a)

$$\begin{array}{rcll}
iis[E_s'](l) & \Rightarrow & iis[E_s](l) & \text{Figure 6}, E_s' \subseteq E_s \\
& \Rightarrow & is^\sharp(\beta(l)) & \text{Definition 5.2.1} \\
& \Rightarrow & is^\sharp(\beta'(l)) & \text{Observation B.5}
\end{array}$$

Proof of (b)

$$\begin{array}{rcll}
iis[E_s'](l) & \Rightarrow & iis^\sharp[\beta'(E_s')](\beta'(l)) & \text{Lemma 5.3.1 (4)} \\
& \Rightarrow & iis^\sharp[E_s^{\sharp'}](\beta'(l)) & \text{Part I and the fact that } iis^\sharp[E_s^\sharp] \text{ is monotonic in } E_s^\sharp. \square
\end{array}$$

Lemma B.6. $\beta([\![x.sel := y]\!](SG)) \sqsubseteq [\![x.sel := y]\!]^\sharp(\beta(SG))$.

Proof. Because the set of variable-edges is unchanged in the transformers for statements of the form $x.sel := y$ in both the operational and the abstract semantics, (6) trivially holds. Also, the following observation holds:

Observation B.7. *$E_v' = E_v$ and therefore $\beta' = \beta$.*

We now show that (7) and (8) hold.

Part I. In this part, we show that (7) holds. Let $\langle l, sel', l'\rangle$ be a selector edge in E_s'. By Figure 6, there are two cases to consider:

Case 1: $\langle l, sel', l'\rangle \in E_s$

$$\begin{array}{rcll}
\langle l, sel', l'\rangle \in E_s & \Rightarrow & \beta(\langle l, sel', l'\rangle) \in E_s^\sharp & \text{Definition 5.2.1} \\
& \Rightarrow & \beta'(\langle l, sel', l'\rangle) \in E_s^{\sharp'} & \text{Observation B.7}
\end{array}$$

Case 2: $[x,l],[y,l'] \in E_v$

$$\begin{aligned}
[x,l],[y,l'] \in E_v &\Rightarrow \beta([x,l]),\beta([y,l']) \in E_v^\sharp && \text{Definition 5.2.1}\\
&\Rightarrow [x,\beta(l)],[y,\beta(l')] \in E_v^\sharp && \text{Definition 5.2.1}\\
&\Rightarrow \langle\beta(l), sel, \beta(l')\rangle \in E_v^{\sharp\,\prime} && \text{Figure 8}\\
&\Rightarrow \langle\beta'(l), sel, \beta'(l')\rangle \in E_v^{\sharp\,\prime} && \text{Observation B.7}\\
&\Rightarrow \beta'(\langle l, sel, l'\rangle) \in E_v^{\sharp\,\prime} && \text{Definition 5.2.1}
\end{aligned}$$

Part II. In this part we show that (8) holds. Let $l \in shape\_nodes(\langle E'_v, E'_s\rangle)$ such that $iis[E'_s](l)$. There are two cases to consider.

Case 1: $iis[E_s](l)$.

$$\begin{aligned}
iis[E_s](l) &\Rightarrow is^\sharp(\beta(l)) && \text{Definition 5.2.1}\\
&\Rightarrow is^{\sharp\,\prime}(\beta(l)) && \text{Figure 8}\\
&\Rightarrow is^{\sharp\,\prime}(\beta'(l)) && \text{Observation B.7}
\end{aligned}$$

Case 2: $\neg iis[E_s](l)$. In this case, by Figure 6, $[y,l] \in E_v$ and therefore, by Definition 5.2.1, $[y,\beta(l)] \in E_v^\sharp$. Also,

$$\begin{aligned}
iis[E'_s](l) &\Rightarrow iis^\sharp[\beta'(E'_s)](\beta'(l)) && \text{Lemma 5.3.1 (4)}\\
&\Rightarrow iis^\sharp[E_s^{\sharp\,\prime}](\beta'(l)) && \text{Part I and the fact that } iis^\sharp[E_s^\sharp] \text{ is monotonic is } E_s^\sharp\\
&\Rightarrow is^{\sharp\,\prime}(\beta'(l)) && \text{Figure 8.}\square
\end{aligned}$$