Shape Analysis with Structural Invariant Checkers

Bor-Yuh Evan Chang
Xavier Rival
George Necula

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-80
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-80.html

June 4, 2007

Acknowledgement

This research was supported in part by the National Science Foundation under grants CCR-0326577, CCF-0524784, and CNS-0509544; and an NSF Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Shape Analysis with Structural Invariant Checkers*

Bor-Yuh Evan Chang[1], Xavier Rival[1,2], and George C. Necula[1]

[1] University of California, Berkeley, California, USA
[2] École Normale Supérieure, Paris, France
{bec,rival,necula}@cs.berkeley.edu

Abstract. Developer-supplied data structure specifications are important to shape analyses, as they tell the analysis what information should be tracked in order to obtain the desired shape invariants. We observe that data structure checking code (e.g., used in testing or dynamic analysis) provides shape information that can also be used in static analysis. In this paper, we propose a lightweight, automatic shape analysis based on these developer-supplied structural invariant checkers. In particular, we set up a parametric abstract domain, which is instantiated with such checker specifications to summarize memory regions using both notions of complete and partial checker evaluations. The analysis then automatically derives a strategy for canonicalizing or weakening shape invariants.

1 Introduction

Pointer manipulation is fundamental in almost all software developed in imperative programming languages today. For this reason, verifying properties of interest to the developer or checking the pre-conditions for certain complex program transformations (e.g., refactorings) often requires detailed aliasing and structural information. Shape analyses are unique in that they can provide this detailed must-alias and shape information that is useful for many higher-level analyses (e.g., typestate or resource usage analyses, race detection for concurrent programs). Unfortunately, because of precision requirements, shape analyses have been generally prohibitively expensive to use in practice.

The design of our shape analysis is guided by the desire to keep the abstraction close to informal developer reasoning and to maintain a reasonable level of interaction with the user in order to avoid excessive case analysis. In this paper, we propose a shape analysis guided by the developer through programmer-supplied data structure invariants. The novel aspect of our proposal is that these specifications are given as checking code, that is, code that could be used to verify instances dynamically. In this paper, we make the following contributions:

* This research was supported in part by the National Science Foundation under grants CCR-0326577, CCF-0524784, and CNS-0509544; and an NSF Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

- We observe that invariant checking code can help guide a shape analysis and provides a familiar mechanism for the developer to supply information to the analysis tool. Intuitively, checkers can be viewed as programmer-supplied summaries of heap regions bundled with a usage pattern for such regions.
- We develop a shape analysis based on programmer-supplied invariant checkers (utilizing the framework of separation logic [Rey02]).
- We introduce a notion of partial checker runs (using $-*$) as part of the abstraction in order to generalize programmer-supplied summaries when the data structure invariant holds only partially (Sect. 3).
- We notice that the iteration history of the analysis can be used to guide the weakening of shape invariants, which perhaps could apply to other shape analyses. We develop an automatic widening strategy for our abstraction based on this observation (Sect. 4.2).

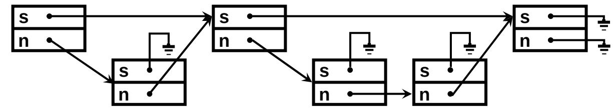
In this paper, we consider structural invariants, that is, invariants concerning the pointer structure (e.g., acyclic list, cyclic list, tree) but not data properties (e.g., orderedness). In the next section, we motivate the design of our shape analysis and highlight the challenges through an example.

2 Overview

In Fig. 1, we present an example analysis that checks a skip list [Pug90] rebalancing operation to verify that it preserves the skip list structure. At the top, we show the structure of a two-level skip list. In such a skip list, each node is either level 1 or level 0. All nodes are linked together with the next field (`n`), while the level 1 nodes are additionally linked with the skip field (`s`). A level 0 node has its `s` field set to `null`. In the middle left, we give the C type declaration of a `SkipNode` and in the middle right, we give a checking routine `skip1` that when viewed as C code (assumed type safe) either diverges if there is a cycle in the reachable nodes, returns false, or returns true when the nodes reachable from the argument `l` are arranged in a skip list structure. The `skip0` function is a helper function for checking a segment of level 0 nodes. Intuitively, `skip1` and `skip0` simply give the inductive structure of skip lists.

In the bottom section of Fig. 1, we present an analysis of the rebalancing routine (`rebalance`). The **assert** at the top ensures that `skip1(l)` holds (i.e., `l` is a skip list), and the **assert** at the bottom checks that `l` is again a skip list on return. We have made explicit these pre- and post-conditions here, but we can imagine a system that connects the checker to the type and verifies that the structure invariants are preserved at function or module boundaries. In the figure, we show the abstract memory state of the analysis at a number of program points using a graphical notation, which for now, we can consider as informal sketches a developer might draw to check the code by hand. For the program points inside the loop there are two memory states shown: one for the first iteration (left) and one for the fixed point (right).

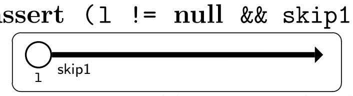
A programmer-defined checker can be used in static analysis by viewing the memory addresses it would dereference during a successful execution as describ-



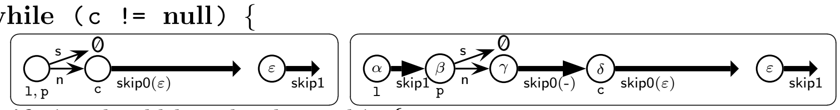

```
typedef
  struct SkipNode {
    int d;
    struct SkipNode* s;
    struct SkipNode* n;
  }
  SkipNode;
```

```
bool skip1(SkipNode* l) {
  if (l == null) return true;
  else return skip1(l->s) &&
              skip0(l->n, l->s);
}
bool skip0(SkipNode* l, SkipNode* e) {
  if (l == e) return true;
  else return l != null && l->s == null &&
              skip0(l->n, e);
}
```

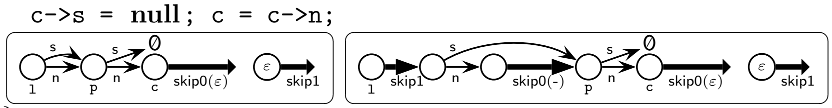
```
void rebalance(SkipNode* l) {
  SkipNode *p, *c;
  assert (l != null && skip1(l));
1
```



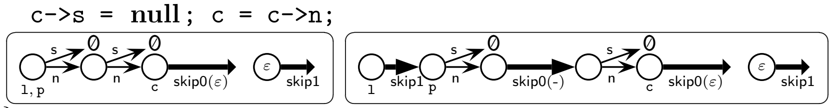

```
   p = l;       // previous level 1 node
2  c = l->n;    // cursor
3  l->s = null;
4  while (c != null) {
5
```




```
     if (c should be a level 1 node) {
6      p->s = c;    // set the skip pointer of the previous level 1 node
7      p = p->s;
8      c->s = null; c = c->n;
9
```




```
     }
     else {
10     c->s = null; c = c->n;
11
```




```
     }
   }
12 assert (l != null && skip1(l));
}
```

First Iteration — At Fixed Point

Fig. 1. Analysis of a skip list rebalancing

ing a class of memory regions arranged according to particular constraints. We build an abstraction around this summarization mechanism. To name heap objects, the analysis introduces *symbolic values* (i.e., fresh existential variables). To distinguish them from program variables, we use lowercase Greek letters ($\alpha, \beta, \gamma, \delta, \varepsilon, \pi, \rho, \ldots$). A graph node denotes a value (e.g., a memory address) and, when necessary, is labeled by a symbolic value; the `0` nodes represent `null`. We write a program variable (e.g., `l`) below a node to indicate that the value of that variable is that node. Each edge corresponds to a memory region. A thin edge denotes a *points-to* relationship, that is, a memory cell whose address is the source node and whose value is the destination node (e.g., on line 5 in the left graph, the edge labeled by `n` says that `l->n` points to `c`). A thick edge summarizes a memory region, i.e., some number of points-to edges. Thick edges, or *checker edges*, are labeled by a checker instantiation that describes the structure of the summarized region. There are two kinds of checker edges: complete checker edges, which have only a source node, and partial checker edges, which have both a source and a target node. Complete checker edges indicate a memory region that satisfies a particular checker (e.g., on line 1, the complete checker edge labeled `skip1` says there is a memory region from `l` that satisfies checker `skip1`). Partial checker edges are generalization that we introduce in our abstraction to describe memory states at intermediate program points, which we discuss further in Sect. 3. An important point is that two distinct edges in the graph denote disjoint memory regions.

To reflect memory updates in the graph, we simply modify the appropriate points-to edges (performing strong updates). For example, consider the transition from program point 5 to point 9 and the updates on lines 6 and 7. For the updates on line 8, observe that we do not have nodes for `c->s` or `c->n` in the graph at program point 5. However, we have that from `c`, an instance of `skip0` holds, which can be *unfolded* to materialize points-to edges for `c->s` and `c->n` (that is, conceptually unfolding one step of its computation). The update can then be reflected after unfolding.

As exemplified here, we want the work performed by our shape analysis to be close to the informal, on-paper verification that might be done by the developer. The abstractions used to summarize memory regions is developer-guided through the checker specifications. While it may be reasonable to build in generic summarization strategies for common structures, like lists and trees (cf., [DOY06,MNCL06]), it seems unlikely such strategies will suffice for other structures, like the skip lists in this example. Traversal code for checking seems like a useful and intuitive specification mechanism, as such code could be used in testing or dynamic analysis (cf., [SRW02]).

From this example, we make some observations that guide the design of our analysis and highlight the challenges. First, in our diagrams, we have implicitly assumed a disjointness property between the regions described by edges (to perform strong updates on points-to edges). This assumption is made explicit by utilizing separation logic to formalize these diagrams (see Sect. 3). This choice also imposes restrictions on the checkers. That is, all conjunctions are separating conjunctions; in terms of dynamic checking, a compilation of `skip1` must check that each address is dereferenced at most once during the traversal. Second, as with many data structure operations, the rebalance routine requires a traversal using a *cursor* (e.g., `c`). To check properties of such operations, we are often required to track information in detail locally around the cursor, but we may be able to summarize the rest rather coarsely. This summarization cannot be only for the suffix (yet to be visited by the cursor) but must also be for the prefix (already visited by the cursor) (see Sect. 3). Third, similar to other shape analyses, a central challenge is to *fold* the graphs sufficiently in order to find a fixed point (and to be efficient) while retaining enough precision. With arbitrary data structure specifications, it becomes particularly difficult. The key observation we make is that previous iterates are generally more abstract and can be used to guide the folding process (see Sect. 4.2).

3 Memory Abstraction

We describe our analysis within the framework of abstract interpretation [CC77]. Our analysis state is composed of an abstract memory state (in the form of a shape graph) and a pure state to track disequalities (the non-points-to constraints). We describe the memory state in a manner based largely on separation logic, so we use a notation that is borrowed from there.

memories	$M ::= \beta@f \mapsto r \mid M_1 * M_2 \mid \mathsf{emp} \mid \alpha.c(\beta) \mid \alpha_1.c(\beta) \ast\!\!- \alpha_2.c(\beta)$
r-values	$r \;\; ::= \alpha \mid \mathsf{null} \mid \cdots$
symbolic values	$\alpha, \beta, \gamma, \delta, \varepsilon, \pi, \rho, \ldots$
field names	f
checker names	c

A memory state M includes the points-to relation ($\beta@f \mapsto r$), the separating conjunction ($M_1 * M_2$), and the empty memory state (emp) from separation logic, which together can describe a set of possible memories that have a finite number of points-to relationships. The separating conjunction $M_1 * M_2$ describes a memory that can be divided into two disjoint regions (i.e., with disjoint domains) described by M_1 and M_2. A field offset expression $\beta@f$ corresponds to the base address β plus the offset of field f (i.e., `&(b.f)` in C). For simplicity, we assume that all pointers occur as fields in a `struct`. R-values r are symbolic expressions representing the contents of memory cells (whose precise form is unimportant but does include null). Memory regions are summarized with applications of user-supplied checkers. We write $\alpha.c(\beta)$ to mean checker c applied to α and β holds (i.e., c succeeds when applied to α and β). For example, $\alpha.\mathsf{skip1}()$ says that the `skip1` checker is successful when applied to α. We use this object-oriented style notation to distinguish the main traversal argument α from any additional parameters β. These additional parameters may be used to specify additional constraints (as in the `skip0` checker in Fig. 1), but we do not traverse from them. We also introduce a notion of a *partial checker*

$\alpha_1 @ f \mapsto \alpha_2$

$\alpha @ f \mapsto \mathsf{null}$

$\alpha.c(\beta)$

$\alpha_1.c(\beta) \ast\!\!- \alpha_2.c(\beta)$

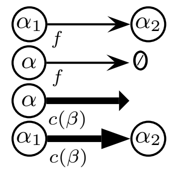


Fig. 2. Correspondence between formulas and edges

run $\alpha_1.c(\beta) \ast\!\!- \alpha_2.c(\beta)$ that describes a memory region summarized by a segment from α_1 to α_2, which will be described further in the subsections below. Visually, we regard a memory state as a directed graph. The edges correspond to formulas as shown in Fig. 2.[3] Each edge in a graph is considered separately conjoined (i.e., each edge corresponds to a disjoint region of memory).

Inductive Structure Checkers. The abstract domain provides generic support for inductive structures through *user-specified checkers*. Observe that a dynamic run of a checker, such as `skip1` (in Fig. 1), visits a region of memory starting from some root pointer, and furthermore, a successful, terminating run of a checker indicates how the user intends to access that region of memory. In the context of our analysis, a checker gives a corresponding inductively-defined predicate in separation logic and a successful, terminating run of the checker bears witness to a derivation of that predicate.

The definition of a checker c, with formals π and ρ, consists of a finite disjunction of rules. A rule is the conjunction of a separating conjunction of a series of points-to relations and checker applications M and a pure, first-order predicate P, written $\langle M \,;\, P \rangle$.

$$\text{checker definitions} \qquad \pi.c(\rho) \;:=\; \langle M_1 \,;\, P_1 \rangle \vee \cdots \vee \langle M_n \,;\, P_n \rangle$$

Free variables in the rules are considered as existential variables bound at the definition. Because we view checkers as executable code, the kinds of inductive predicates are restricted. More precisely, we have the following restrictions on the M_i's: (1) they do not contain partial checker applications (i.e., $\ast\!\!-$) and (2) the points-to edges correspond to finite access paths from π. In other words, each M_i can only correspond to a memory region reachable from π. A checker cannot, for example, posit the existence of some pointer that points to π.

Each rule specifies one way to prove that a structure satisfies the checker definition, by checking that the corresponding first-order predicate holds and that the store can be separated into a series of stores, which respectively allow proving each of the separating conjuncts. Base cases are rules with no checker applications.

[3] For presentation, we show the most common kinds of edges. In the implementation, we support field offsets in most places to handle, for example, pointer to fields.

Example 1 (A binary tree checker). A binary tree with fields lt and rt can be described by a checker with two rules:

$$\pi.\mathsf{tree}() := \langle \mathsf{emp}\,;\, \pi = \mathsf{null}\rangle \vee \langle (\pi@\mathsf{lt} \mapsto \gamma) * (\pi@\mathsf{rt} \mapsto \delta) * \gamma.\mathsf{tree}() * \delta.\mathsf{tree}()\,;\, \pi \neq \mathsf{null}\rangle$$

Example 2 (A skip list checker). The "C-like" checkers for the two-level skip list in Fig. 1 would be translated to the following:

$$\begin{aligned}
\pi.\mathsf{skip1}() \;&:= \langle \mathsf{emp}\,;\, \pi = \mathsf{null}\rangle \\
&\quad \vee \langle (\pi@\mathsf{s} \mapsto \gamma) * (\pi@\mathsf{n} \mapsto \delta) * \gamma.\mathsf{skip1}() * \delta.\mathsf{skip0}(\gamma)\,;\, \pi \neq \mathsf{null}\rangle \\
\pi.\mathsf{skip0}(\rho) &:= \langle \mathsf{emp}\,;\, \pi = \rho\rangle \\
&\quad \vee \langle (\pi@\mathsf{s} \mapsto \mathsf{null}) * (\pi@\mathsf{n} \mapsto \gamma) * \gamma.\mathsf{skip0}(\rho)\,;\, \pi \neq \rho \wedge \pi \neq \mathsf{null}\rangle
\end{aligned}$$

Segments and Partial Checker Runs. In the above, we have built some intuition on how user-specified checkers can be utilized to give precise summaries of memory regions. Unfortunately, the inductive predicates obtained from typical checkers, such as `tree` or `skip1`, are usually not general enough to capture the invariants of interest at all program points. To see this, consider the invariant at fixed point on line 5 (i.e., the loop invariant) in the skip list example (Fig. 1). Here, we must track some information in detail around a cursor (e.g., `p` and `c`), while we need to summarize *both* the already explored prefix before the cursor and the yet to be explored suffix after the cursor. Such a situation is typical when analyzing a traversal algorithm. The suffix can be summarized by a checker application $\delta.\mathsf{skip0}(\varepsilon)$ (i.e., the $\mathsf{skip0}$ edge from `c`), but unfortunately, the prefix segment (i.e., the region between `l` and `p`) cannot.

Rather than require more general checker specifications sufficient to capture these intermediate invariants, we introduce a generic mechanism for summarizing prefix segments. We make the observation that they are captured by *partial checker runs*. In terms of inductively-defined predicates, we want to consider partial derivations, that is, derivations with a hole in a subtree. This concept is internalized in the logic with the separating implication. For example, the segment from `l` to `p` on line 5 corresponds to the partial checker application $\alpha.\mathsf{skip1}() \mathrel{*\!-} \beta.\mathsf{skip1}()$. Informally, a memory region satisfies $\alpha.\mathsf{skip1}() \mathrel{*\!-} \beta.\mathsf{skip1}()$ if and only if for any disjoint region that satisfies $\beta.\mathsf{skip1}()$ (i.e., is a skip list from β), then conjoining that region satisfies $\alpha.\mathsf{skip1}()$ (i.e., makes a complete skip list from α). This statement entails that β is reachable from α. Our notation for separating implication is reversed compared to the traditional notation $-\!*$ to mirror more closely the graphical diagrams. Our use of separating implication is restricted to the form where the premise and conclusion are checker applications that differ only in the unfolding argument because these are the only partial checker edges our analysis generates.

Semantics of Shape Graphs. For completeness in presentation, we give the semantics of abstract memory states with checkers (i.e., graphs) in terms of sets of concrete stores, which follows mostly from separation logic. In Sect. 4, we describe the shape analysis algorithm that utilizes this memory abstraction.

We write $u, v \in \mathbf{Val}$ for concrete values and make no distinction between addresses and values, and we write $v @ f$ to mean the address $v + \text{offset}(f)$ (i.e., the base address v plus the field offset f). A concrete *store* $\sigma : \mathbf{Val} \rightharpoonup \mathbf{Val}$ maps addresses to values. We write $\sigma_1 * \cdots * \sigma_n$ for the store with disjoint sub-stores $\sigma_1, \ldots, \sigma_n$ (i.e., they have disjoint domains). For the empty store, we write $[\,]$, and for the store with one cell with address v and containing value u, we write $[v \mapsto u]$. A *valuation* ν is a substitution with concrete values for symbolic values (written $[\vec{v}/\vec{\alpha}]$). Finally, we write νM for applying the valuation ν to M.

We say a concrete store σ satisfies an abstract memory M if there exists a valuation ν such that $\sigma \models \nu M$ where the relation $\models$ is defined as the least relation satisfying the following rules:

$[\,] \models \mathsf{emp}$	(always)
$[v@f \mapsto u] \models [v@f \mapsto u]$	(always)
$\sigma_1 * \sigma_2 \models M_1 * M_2$	if $\sigma_1 \models M_1$ and $\sigma_2 \models M_2$

$\sigma \models v.c()$
    if there exists a rule $\langle M \,;\, P \rangle$ in the definition of $\pi.c()$ and there exist values $\vec{u}$ such that σ satisfies the pure formula $[v/\pi][\vec{u}/\vec{\alpha}]P$ and $\sigma \models [v/\pi][\vec{u}/\vec{\alpha}]M$ where $\vec{\alpha}$ are the free variables of the rule.

$\sigma \models v.c() \mathrel{*\!\!-} v'.c()$
    if for all σ' (disjoint from σ), if $\sigma' \models v'.c()$, then $\sigma * \sigma' \models v.c()$.

For presentation, we write the semantics with checkers with no additional parameters. They can be extended to checkers with parameters without any difficulty.

4 Analysis Algorithm

In this section, we describe our shape analysis algorithm. Like many other shape analyses, we have a notion of *materialization*, which reifies memory regions in order to track updates, as well as *blurring* or *weakening*, which (re-)summarizes certain memory regions in order to obtain a terminating analysis. For us, we materialize by *unfolding* checker edges (Sect. 4.1) and weaken by *folding* memory regions back into checker edges (Sect. 4.2). Like others, we materialize as needed to reflect updates and dereferences, but instead of weakening eagerly, we delay weakening in order to use history information to guide the process.

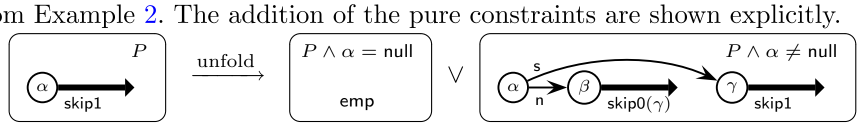
Our shape analysis is a standard forward analysis that computes an abstract state at each program point. In addition to the memory state (as described in Sect. 3), the analysis also keeps track of a number of pure constraints P (pointer equalities and disequalities). Furthermore, we maintain some disjunction, so our analysis state has essentially the following form: $\langle M_1 \,;\, P_1 \rangle \vee \langle M_2 \,;\, P_2 \rangle \vee \cdots \vee \langle M_n \,;\, P_n \rangle$ (for unfoldings and acyclic paths where needed). Additionally, we keep the values of the program variables (i.e., the stack frame) in an abstract environment E that maps program variables to symbolic values that denote their contents.[4]

4.1 Abstract Transition and Checker Unfolding

Because each edge in the graph denotes a separate memory region, the atomic operations (i.e., mutation, allocation, and deallocation) are straightforward and only affect graphs locally. As alluded to in Sect. 2, mutation reduces to the flipping of an edge when each memory cell accessed in the statement exists in the graph as a points-to edge. This strong update is sound because of separation (that is, because each edge is a disjoint region).

When there is no points-to edge corresponding to a dereferenced location because it is summarized as part of a checker edge, we first materialize points-to edges by *unfolding* the checker definition (i.e., conceptually unfolding one-step of the checker run). We unfold only as needed to expose the points-to edge that corresponds to the dereferenced location. Unfolding generates one graph per checker rule, obtained by replacing the checker edge with the points-to edges and the recursive checker applications specified by the rule; the pure constraints in the rule are also added to pure state. In case we derive a contradiction (in the pure constraints), then those unfolded elements are dropped. Though, unfolding may generate a *disjunction* of several graphs. A fundamental property of unfolding is that the join of the concretizations of the resulting graphs is equal to the concretization of the initial graph.

Example 3 (Unfolding a skip list). We exhibit an unfolding of the skip1 checker from Example 2. The addition of the pure constraints are shown explicitly.



4.2 History-Guided Folding

We need a strategy to identify sub-graphs that should be *folded* into complete or partial checker edges. What kinds of sub-graphs can be summarized without losing too much precision is highly dependent on the structures in question and the code being analyzed. To see this, consider the fixed-point graph at program point 5 in this skip list example (Fig. 1). One could imagine folding the points-to edges corresponding to `p->n` and `p->s` into one summary region from `p` to `c` (i.e., eliminating the node labeled γ), but it is necessary to retain the information that `p` and `c` are "separated" by at least *one* `n` field. Keeping node γ expresses this fact. Rather than using a *canonicalization* operation that looks only at one graph to identify the sub-graphs that should be summarized, our weakening strategy is based on the observation that previous iterates at loop join points can be utilized

[4] In implementation, we instead include the stack frame in M to enable handling address of local variable expressions (as in C) in a smooth manner.

to guide the folding process. In this subsection, we define the *approximation test* and *widening* operations (standard operations in abstract interpretation-based static analysis) over graphs as a simultaneous traversal over the input graphs.

Approximation Test. The approximation test on memory states $M_1 \sqsubseteq M_2$ takes two graphs as input and tries to establish that the concretization of M_1 is contained in the concretization of M_2 (i.e., $M_1 \Rightarrow M_2$). Static analyses rely on the approximation test in order to ensure the termination of fixed point computation. We also utilize it to collapse extraneous disjuncts in the analysis state and most importantly, as a sub-routine in the widening operation.

Roughly speaking, our approximation test checks that graph M_1 is equivalent to graph M_2 up to unfolding of M_2. That is, the basic idea is to determine whether $M_1 \sqsubseteq M_2$ by reducing to stronger statements either by matching edges on both sides or by unfolding M_2. To check this relation, we need a correspondence between nodes of M_1 and nodes of M_2. This correspondence is given by a mapping Φ from nodes of M_2 to those of M_1. The condition that Φ is such a function ensures any aliasing expressed in M_2 is also reflected in M_1. If at any point, this condition on Φ is violated, then the test fails.

Initialization. The mapping Φ plays an essential role in the algorithm itself since it gives the points from where we should compare the graphs. It is initialized using the environment and then extended as the input graphs are traversed. The natural starting points are the nodes that correspond to the program variables (i.e., the initial mapping $\Phi_0 = \{E_2(x) \to E_1(x) \mid x \in \mathbf{Var}\}$).

Traversal. After initialization, we decide the approximation relation by traversing the input graphs and attempting to match all edges. To check region disjointness (i.e., linearity), when edges are matched, they are "consumed". If the algorithm gets stuck where not all edges are "consumed", then the test fails. To describe this traversal, we define the judgment $M_1 \sqsubseteq M_2[\Phi]$ that says, "M_1 is approximated by M_2 under Φ."

In the following, we describe the rules that define $M_1 \sqsubseteq M_2[\Phi]$ by following the example derivation shown in Fig. 3 (from goal to axiom). A complete listing of the rules is given in Appendix A. In Fig. 3, the top line shows the initial goal with a particular initialization for Φ. Each subsequent line shows a step in the derivation (i.e., a rewriting step) that is obtained by applying the rule named on the right. The highlighting of nodes and edges indicates where the rewriting applies. We are able to prove that the left graph is approximated by the right graph because we reach $\mathsf{emp} \sqsubseteq \mathsf{emp}[\Phi]$.

First, consider the application of the $\mathsf{pointsto}$ rule (line 3 to 4). When both M_1 and M_2 have the same kind of edge from matched nodes, the approximation relation obviously holds for those edges, so those edges can be consumed. Any target nodes are then added to the mapping Φ so that the traversal can continue from those nodes. In this case, the s and n points-to edges match from the pair $\alpha \to \delta$. With this matching, the mappings $\beta \to \varepsilon, \gamma \to \varepsilon$ are added. We highlight

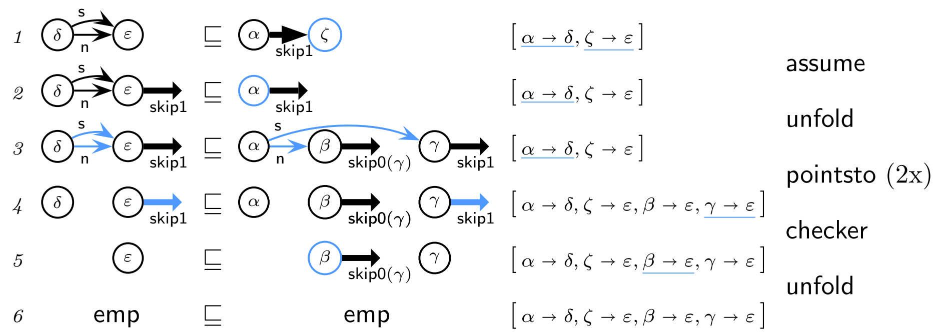


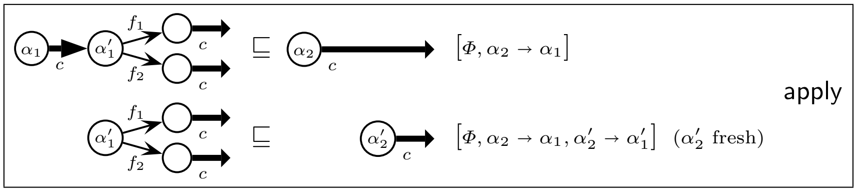
Fig. 3. Testing approximation by reducing to stronger statements

in Φ with underlines the mappings that must match for each rule to apply. The checker rule is the analogous matching rule for complete checker edges. We apply this edge matching only to points-to edges and complete checker edges. Partial checker edges are treated separately as described below.

Partial checker edges are handled by taking the separating implication interpretation, which becomes critical here. We use the assume rule (as in the first step in Fig. 3) to reduce the handling of partial checker edges in M_2 to the handling of complete checker edges (i.e., a "$-\!\!*$ right" in sequent calculus or "$-\!\!*$ introduction" in natural deduction). It extends the partial checker edge in M_2 to a complete checker edge by adding the corresponding completion to M_1. A key aspect of our algorithm is that this rule only applies when we have matched both the source and target nodes of the partial checker edge, that is, we have delineated in M_1 the region that corresponds to the partial checker edge in M_2.

Now, consider the first application of unfold in Fig. 3 (line 2 to 3) where we have a complete checker edge from α on the right, but we do not have an edge from δ on the left that can be immediately matched with it. In this case, we unfold the complete checker edge. In general, the unfolding results in a disjunction of graphs (one for each rule, Sect. 4.1), so the overall approximation check succeeds if the approximation check succeeds for any *one* of the unfolded graphs. Note that on an unfolding, we must also remember the pure constraint P from the rule, which must be conjoined to the pure state on the right when we check the approximation relation on the pure constraints. In the second application of unfold in Fig. 3 (line 5 to 6), the unfolding of $\beta.\mathsf{skip0}(\gamma)$ is to emp because we have that $\beta = \gamma$. This equality arises because they are both unified with ε (specifically, the pointsto steps added $\beta \to \varepsilon$ and $\gamma \to \varepsilon$ to Φ).

Finally, we also have a rule for partial checkers in M_1 (i.e., a corresponding "left" or "elimination" rule). Since it is not used in the above example, we present it below schematically:



The rule is presented in the same way as in the example (i.e., with the goal on top). Conceptually, this rule can be viewed as a kind of unfolding rule where the complete checker edge in M_2 is unfolded the necessary number of steps to match the the partial checker edge in M_1.

Informally, the soundness of the approximation test can be argued from separation logic principles and from the fact that unfoldings have equivalent concretizations. The approximation test is, however, incomplete (i.e., it may fail to establish that an approximation relation between two graphs when their concretizations are ordered by subset containment). Rather these rules have been primarily designed to be effective in the way the approximation test is used by the widening operation as described in the next subsection where we need to determine if M_1 is an unfolded version of M_2.

Widening. In this subsection, we present an upper bound operation $M_1 \triangledown M_2$ that we use as our widening operator at loop join points. The case of disjunctions of graphs will be addressed below. At a high-level, the upper bound operation works in a similar manner as compared to the approximation test. We maintain a node pairing Ψ that relates the nodes of M_1 and M_2. Because we are computing an upper bound here, the pairing Ψ need not have the same restriction as in the approximation test; it may be any relation on nodes in M_1 and M_2. From this pairing, we simultaneously traverse the input graphs M_1 and M_2 consuming edges. However, for the upper bound operation, we also construct the upper bound as we consume edges from the input graphs. Intuitively, the basic edge matching rules will lay down the basic structure of the upper bound and guide us to the regions of memory that need to be folded.

Initialization. The initialization of Ψ is the analogous to the approximation test initialization: we pair the nodes that correspond to the values of each variable from the environments (i.e., the initial pairing $\Psi_0 = \{\langle E_1(x), E_2(x)\rangle \mid x \in \mathbf{Var}\}$).

Traversal. To describe the upper bound computation, we define a set of rewriting rules of the form $\Psi \mathbin{⨾} (M_1 \triangledown M_2) \circledast M \twoheadrightarrow \Psi' \mathbin{⨾} (M_1' \triangledown M_2') \circledast M'$. Initially, M is emp, and then we try to rewrite until M_1' and M_2' are emp in which case M' is the upper bound. A node in M corresponds to a pair (from M_1 and M_2). Conceptually, we build M with nodes labeled with such pairs and then relabel each distinct pair with a distinct symbolic value at the end.

Figure 4 shows an example sequence of rewritings to compute an upper bound. A complete listing of the rewrite rules is given in Appendix B. We elide the pairing Ψ, as it can be read off from the nodes in the upper bound graph M

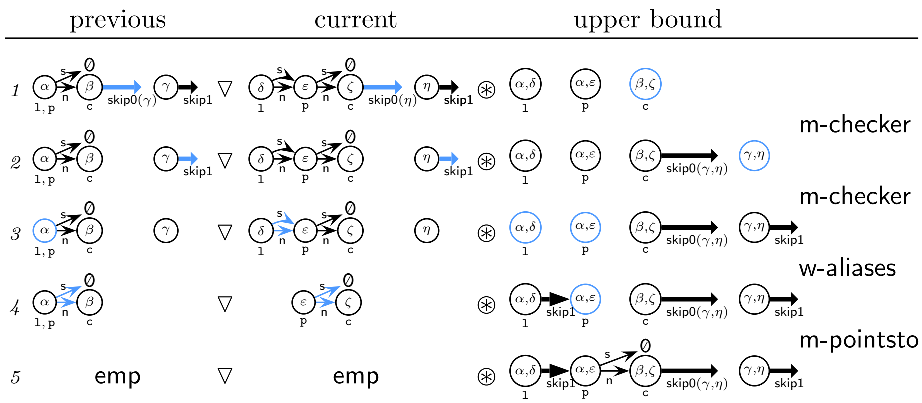
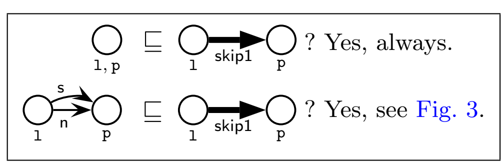


Fig. 4. An example of computing an upper bound. The inputs are the graphs on the first iteration at program points 5 and 9 in the skip list example (Fig. 1). The fixed-point graph at 5 is obtained by computing the upper bound of this result and the upper bound of the first-iteration graphs at 5 and 11

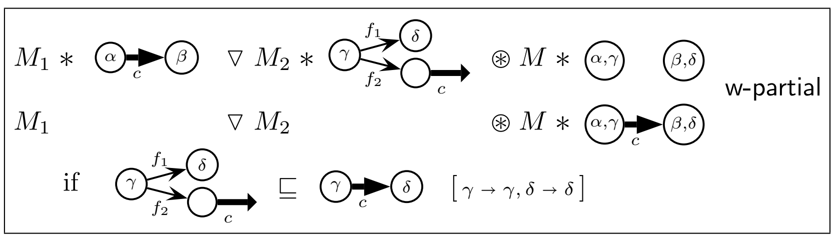
(the rightmost graph). The highlighting of nodes in the upper bound graph indicate the node pairings that are required to apply the rule, and the highlighting of edges in the input graphs show which edges are consumed in the rewriting step. Roughly speaking, the upper bound operation has two kinds of rules: matching rules for when we have the same kind of edge on both sides (like in the approximation test) and weakening rules where we have identified a memory region to fold. We use the prefix m- for the matching rules and w- for the weakening rules.

Line 1 shows the state after initialization: we have nodes in upper bound graph for the program variables. The first two steps (applying rule m-checker) match complete checker edges (first from $\langle\beta,\zeta\rangle$ and then from $\langle\gamma,\eta\rangle$). Note that the second application is enabled by the first where we add the pair $\langle\gamma,\eta\rangle$. Extra parameters are essentially implicit target nodes.



The core of the upper bound operation are three weakening rules where we fold memory regions. The next rule application w-aliases is such a weakening step (line 3 to 4). In this case, a node on one side is paired with two nodes on the other ($\langle\alpha,\delta\rangle$ and $\langle\alpha,\varepsilon\rangle$). This situation arises where on one side, we have must-alias information, while the other side does not (`l` and `p` are aliased on the left but not on the right). In this case, we want to weaken both sides to a partial checker edge. To see that this is indeed an upper bound for these regions, consider the diagram in the inset. As shown on the first line, aliases can always be weakened to a partial checker edge (intuitively, from a zero-step segment to a zero-or-more step segment). On the second line, we need to check that a skip1 checker edge is indeed weaker than the region between δ and ε. This check is done using the approximation test described in the previous subsection. The check we need to perform here is the example shown in Fig. 3. Observe that we utilize the edge matching rules that populates Ψ to delineate the region to be folded (e.g., the region between δ and ε in the right graph). For the w-aliases rule, we do not specify here how the checker c is determined, but in practice, we can limit the checkers that need to be tried by, for example, tracking the type of the node (or looking at the fields used in outgoing points-to edges).

There are two other weakening rules w-partial and w-checker that are not used in the above example. Rule w-partial applies when we identify that an (unfolded) memory region on one side corresponds to a partial checker edge on the other. In this case, we weaken to the partial checker edge if we can show the partial checker edge is weaker than the memory region. Rule w-partial is shown below schematically:



Observe that we find out that the region in the right graph must be folded because the corresponding region in the left graph is folded (and also indicates which checker to use). Rule w-checker is the analogous rule for a complete checker edge.

In Fig. 4, the last step is simply matching points-to edges. When we reach emp for M_1 and M_2, then M is the upper bound. In general, if, in the end, there are regions we cannot match or weaken in the input graphs, we can obtain an upper bound by weakening those regions to $\top$ in the resulting graph (i.e., a summary region that cannot be unfolded). This results in an enormous loss in precision that we would like avoid but can be done if necessary.

Soundness. The basic idea is that we compute an upper bound by rewriting based on the following derived rule of inference in separation logic:

$$\frac{M_1' \Rightarrow M \quad M_2' \Rightarrow M}{(M_1 * M_1') \vee (M_2 * M_2') \Rightarrow (M_1 \vee M_2) * M}$$

For each memory region in the input graphs, either they have the same structure in the input graphs and we preserve that structure or we weaken to a checker edge only when we can decide the weakening with $\sqsubseteq$. That is, during the traversal, we simply alternate between weakening memory regions in each input graph to make them match and applying the distributivity of separating conjunction over disjunction to factor out matching regions.

Termination. We shall use this upper bound operation as our widening operator, so we check that it has the stabilizing property (i.e., successive iterates eventually stabilize) to ensure termination of the analysis. Consider an infinite ascending chain

$$M_0 \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq \cdots$$

and the corresponding widening chain

$$M_0 \sqsubseteq (M_0 \triangledown M_1) \sqsubseteq ((M_0 \triangledown M_1) \triangledown M_2) \sqsubseteq \cdots$$

(i.e., the sequence of iterates). The widening chain stabilizes because the successive iterates are bounded by the size of M_0. Over the sequence of iterates, the only rule that may produce additional edges not present in M_0 is $\mathsf{w\text{-}aliases}$, but its applicability is limited by the number of nodes. Then, nodes are created in the result only in two cases: the target node when matching points-to edges ($\mathsf{m\text{-}pointsto}$) and any additional parameter nodes when matching complete checker edges ($\mathsf{m\text{-}checker}$). Points-to and complete checker edges are only created in the resulting graph because of matching, so the number of nodes is limited by the points-to and complete checker edges in M_0.

Strategy for applying rules. Unlike the approximation test, the upper bound rules as described above have a fair amount of non-determinism, and unfortunately, applying the rules in different orders may yield different results in terms of precision. To avoid an exponential explosion in computational complexity, we fix a particular strategy in which to apply the rules, which has been determined, in part, experimentally. We note, however, that neither soundness nor termination are affected by the strategy that we choose. Intuitively, we obtain a good result when we are able to consume all the edges in the input graphs by applying the upper bound rules. A potential bad interaction between the rules is if we prematurely match (and consume) points-to edges that rather should be weakened together with other edges. For example, in Fig. 4 before $\mathsf{w\text{-}aliases}$, if instead we match the points-to edges $\alpha @ \mathsf{n} \mapsto \beta$ on the left and $\delta @ \mathsf{n} \mapsto \varepsilon$ on the right (i.e., apply $\mathsf{m\text{-}pointsto}$) creating the pair $\langle \beta, \varepsilon \rangle$, then we will not be able to consume all edges. Our strategy is to first exhaustively match complete checker edges ($\mathsf{m\text{-}checker}$), as it does not prohibit any other rules and corresponds to identifying the "yet to be explored tail of the structure". Then, since the weakening rules ($\mathsf{w\text{-}aliases}$ and $\mathsf{w\text{-}partial}$) only apply once we have identified corresponding regions (and that can only be consumed by performing this weakening), we apply these rules exhaustively when applicable. To identify such regions, we then apply $\mathsf{m\text{-}pointsto}$ but incrementally (i.e., we match a points-to edge and restart). Finally, when nothing else applies, we try weakenings to complete checker edges ($\mathsf{w\text{-}checker}$).

Disjunctions of graphs. In general, we consider widening disjunctions of graphs. The widening operator for *disjunctions* is based on the operator for graphs and attempts to find pairs that can be widened precisely in the sense that no region need be weakened to $\top$ (i.e., because an input region could not be matched). In addition to this selective widening process, the widening may leave additional disjuncts, up to some fixed limit (perhaps based on trace partitioning [MR05]).

More precisely, let us consider two disjunctions of graphs

$$M_1 \vee \ldots \vee M_n \quad \text{and} \quad M'_1 \vee \ldots \vee M'_{n'}$$

(where we omit the pure formulas for the sake of clarity). Then, the widening on the two disjunctive states relies on the following algorithm:

- for each disjunct M'_j, if there exists an element M_i such that the rewriting rules for the graph widening algorithm for $M_i \triangledown M'_j$ does not get stuck, then add it to the result; if there exists no such element M_i, then add M'_j to the result;
- for each disjunct M_i such that no $M_i \triangledown M'_j$ has been added to the result, then add M_i to the result, unless this would cause the generation of more disjuncts than a fixed constant; in this case, an M'_j should be widened against M_i (with unmatched regions weakened to $\top$ if necessary).

The termination follows from the termination property of the widening operator for pairs of graphs and from the bound on the number of disjuncts.

4.3 Extensions and Limitations

The kinds of structures that can be described with our checkers are essentially trees with regular sharing patterns, which include skip lists, circular lists, doubly-linked lists, and trees with parent pointers. Intuitively, these are structures where one can write a recursive traversal that dereferences each field once (plus pointer equality and disequality constraints). However, the effectiveness of our shape analysis is not the same for all code using these structures. First, we materialize only when needed by unfolding inductive definitions, which means that code that traverse structures in a different direction than the checker are more difficult to analyze. This issue may be addressed by considering additional materialization strategies. Second, in our presentation, we consider partial checker edges with one hole (i.e., a separating implication with one premise). This formulation handles code that use cursors along a path through the structure but not code that uses multiple cursors along different branches of a structure.

5 Experimental Evaluation

We evaluate our shape analysis using a prototype implementation for analyzing C code. Our analysis is written in OCaml and uses the CIL infrastructure [NMRW02]. We have applied our analysis to a number of small data structure manipulation benchmarks and a larger Linux device driver benchmark (`scull`). In the table, we show the size in pre-processed lines of code, the analysis times on a 2.16GHz Intel Core Duo with 2GB RAM, the maximum number of graphs (i.e., number of disjuncts) at any program point, and the maximum

Table 1. Analysis statistics

Benchmark	Code Size (loc)	Analysis Time (sec)	Max. Graphs at Any Point (num)	Max. Iterations at Any Point (num)
list reverse	19	0.007	1	3
list remove element	27	0.016	4	6
list insertion sort	56	0.021	4	7
binary search tree find	23	0.010	2	4
skip list `rebalance`	33	0.087	6	7
`scull` driver	894	9.710	4	16

number iterations at any program point. In each case, we verified that the data structure manipulations preserved the structural invariants given by the checkers. Because we only fold into checkers based only on history information, we typically cannot generate the appropriate checker edge when a structure is being constructed. This issue could be resolved by using constructor functions with appropriate post-conditions or perhaps a one graph operation that can identify potential foldings. For these experiments, we use a few annotations that add a checker edge that say, for example, treat this `null` as the empty list (1 each in list insertion sort and skip list `rebalance`).

The `scull` driver is from the Linux 2.4 kernel and was used by McPeak and Necula [MN05]. The main data structure used by the driver is an array of doubly-linked lists. Because we also do not yet have support for arrays, we rewrote the array operations as linked-list operations (and ignored other `char` arrays). We analyzed each function individually by providing appropriate pre-conditions and inlining all calls, as our implementation does not yet support proper interprocedural analysis. One function (`cleanup_module`) was not completely analyzed because of an incomplete handling of the array issues; it is not included in the line count. We also had 6 annotations for adding checker edges in this example. In all the test cases (including the driver example), the number of graphs we need to maintain at any program point (i.e., the number of disjuncts) seems to stay reasonably low.

6 Related Work

Shape analysis. Shape analysis has long been an active area of research with numerous algorithms proposed and systems developed. Our analysis is closest to some more recent work on separation logic-based shape analyses by Distefano *et al.* [DOY06] and Magill *et al.* [MNCL06]. Their shape analyses infer invariants for programs that manipulate linked-lists. They summarize linked-list regions using a notion of list segments (ls), which is an inductively-defined predicate, that gets unfolded and folded during the course of their analyses. Also like their analyses, we utilize separation explicitly in our memory abstraction, which allows the update operation to affect the memory state in a local manner. The primary difference is that the list segment abstraction is built into their analyses, while our analysis is parameterized by inductive checker definitions. To ensure termination of the analysis, they use a canonicalization operation on list segments (an operation from a memory state to a memory state), while we use a history-guided approach to identify where to fold (an operation from two memory states to one). Note that these approaches are not incompatible with each other, and they have different trade-offs. The additional history information allowed us to develop a generic weakening strategy, but because we are history-dependent, we cannot weaken whenever (e.g., we cannot weaken aggressively after each update). It might be possible to derive automatically canonicalization rules in certain situations based on an analysis of checker definitions. If combined with history-guided weakening, canonicalization would not need to ensure finiteness and could be less aggressive in its folding. Recently, Berdine *et al.* [BCC$^+$07] have developed a shape analysis over generalized doubly-linked lists. They use a higher-order list segment predicate that is parameterized by the shape of the "node", which essentially adds a level of polymorphism to express, for example, a linked list of cyclic doubly-linked lists. We can instead describe custom structures monomorphically with the appropriate checkers, but an extension for polymorphism could be very useful.

Lee *et al.* [LYY05] propose a shape analysis where memory regions are summarized using grammar-based descriptions that correspond to inductively-defined predicates in separation logic (like our checkers). A nice aspect of their analysis is that these descriptions are derived from the construction of the data structure (for a certain class of tree-like structures). For weakening, they use a canonicalization operation to fold memory regions into grammar-based descriptions (non-terminals), but to ensure termination of the analysis, they must fix in advance a bound on the number of nodes that can be in a canonicalized graph.

TVLA [SRW02] is a very powerful and generic system based on three-valued logic and is probably the most widely applied tool for verifying deep properties of complex heap manipulations (e.g., [LRS06,LARSW00]). The framework is parametric in that users can provide specifications (instrumentation predicates) that affect the kinds of structures tracked by the tool. Our analysis is instead parameterized by inductive checker definitions, but since we focus on structural properties, we do not handle any data invariants. Much recent work has been targeted at improving the scalability of TVLA. Yahav and Ramalingam [YR04] partition the memory state into regions that are either tracked more precisely or less precisely depending on their relevance to the property in question. Manevich *et al.* [MSRF04] describe a strategy to merge memory states whose canonicalizations are "similar" (i.e., have isomorphic sets of individuals). Our folding strategy can be seen as being particularly effective when the memory states are "similar"; like them, we would like to use disjunction when the strategy is ineffective. Arnold [Arn06] identifies an instance where a more aggressive summarization loses little precision (by allowing summary nodes to represent zero-or-more concrete nodes instead of one-or-more). Our abstraction is related in that our checker edges denote zero-or-more steps.

Hackett and Rugina [HR05] present a novel shape analysis that first partitions the heap using region inference and then tracks updates on representative heap cells independently. While their abstraction cannot track certain global properties like the aforementioned shape analyses, they make this trade-off to obtain a very scalable shape analysis that can handle singly-linked lists. Recently, Cherem and Rugina [CR07] have extended this analysis to handle doubly-linked lists by including the tracking of neighbor cells.

McPeak and Necula [MN05] identify a class of axioms that can describe many common data structure invariants and give a complete decision procedure for this class. Their technique is based on verification-condition generation and thus requires loop invariant annotations. PALE [MS01] is a similar system also based on verification-condition generation but instead uses monadic second-order logic. Weis *et al.* [WKL+06] have extended PALE with non-deterministic field constraints (and some loop invariant inference), which enables some reasoning of skip list structures.

Inductive checkers. It comes at no surprise that inductive data structures are naturally described using inductive definitions (in some form). For inductive data structures in imperative languages, separation logic enables the specification of such structures in a particularly concise manner because disjointness is built into the logic [Rey02]. By restricting our attention to definitions that can be viewed as code, we impose strictures that are useful for shape analysis. All shape analyses based on separation logic (e.g., [DOY06,MNCL06]) use inductively-defined predicates for abstraction that fall into this class. Perry *et al.* [PJW06] have also observed inductive definitions in a substructural logic could be an effective specification mechanism. They describe shape invariants for dynamic analysis with linear logic (in the form of logic programs).

7 Conclusion

We have described a lightweight shape analysis based on user-supplied structural invariant checkers. These checkers, in essence, provide the analysis with user-specified memory abstractions. Because checkers are only unfolded when the regions they summarize are manipulated, these specifications allow the user to focus the efforts of the analysis by enabling it to expose disjunctive memory states only when needed. The key mechanisms we utilize to develop such a shape analysis is a generalization of checker-based summaries with partial checker runs and a folding strategy based on guidance from previous iterates. In this paper, we have focused on using structural checkers to analyze algorithms that traverse the structures unidirectionally. We believe such ideas could be applicable more broadly (both in terms of utilizable checkers and algorithms analyzed).

Acknowledgments. We would like to thank Hongseok Yang, Bill McCloskey, Gilad Arnold, Matt Harren, and the anonymous referees for providing helpful comments on drafts of this paper.

References

[Arn06] Gilad Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *Static Analysis Symposium (SAS)*, pages 204–220, 2006.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Conference on Computer-Aided Verification (CAV)*, 2007.

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

[CR07] Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302, 2006.

[HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Symposium on Principles of Programming Languages (POPL)*, pages 310–323, 2005.

[LARSW00] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 26–38, 2000.

[LRS06] Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *Static Analysis Symposium (SAS)*, pages 261–279, 2006.

[LYY05] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symposium on Programming (ESOP)*, pages 124–140, 2005.

[MN05] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Conference on Computer-Aided Verification (CAV)*, pages 476–490, 2005.

[MNCL06] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2006.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming (ESOP)*, pages 5–20, 2005.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 221–231, 2001.

[MSRF04] Roman Manevich, Shmuel Sagiv, Ganesan Ramalingam, and John Field. Partially disjunctive heap abstraction. In *Static Analysis Symposium (SAS)*, pages 265–279, 2004.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[PJW06] Frances Perry, Limin Jia, and David Walker. Expressing heap-shape contracts in linear logic. In *Conference on Generative Programming and Component Engineering (GPCE)*, pages 101–110, 2006.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[WKL+06] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin C. Rinard. Field constraint analysis. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 157–173, 2006.

[YR04] Eran Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 25–34, 2004.

A Approximation Test

In this section, we give the rules that define the approximation test as described in Sect. 4.2. Viewed from goal to premise, each rule either matches and consumes edges (pointsto and checker) or simplifies edges in order for matching to apply (assume, apply, and unfold) until all edges have been consumed (emp). Here, to make explicit the updating of the global mapping Φ, we extend the approximation test judgment slightly as follows: $M_1 \sqsubseteq M_2[\Phi_I][\Phi_O]$ where Φ_I is the input mapping (as in Sect. 4.2) and Φ_O is the output mapping (i.e., the resulting mapping after matching nodes).

$$\boxed{M_1 \sqsubseteq M_2[\Phi_I][\Phi_O]}$$

Edge Matching.

$$\frac{M_1 \sqsubseteq M_2[\Phi, r_2 \to r_1][\Phi'] \quad \alpha_2 \to \alpha_1 \in \Phi}{M_1 * (\alpha_1 @ f \mapsto r_1) \sqsubseteq M_2 * (\alpha_2 @ f \mapsto r_2)[\Phi][\Phi']}\ \textsf{pointsto}$$

$$\frac{M_1 \sqsubseteq M_2[\Phi, \beta_2 \to \beta_1][\Phi'] \quad \alpha_2 \to \alpha_1 \in \Phi}{M_1 * \alpha_1.c(\beta_1) \sqsubseteq M_2 * \alpha_2.c(\beta_2)[\Phi][\Phi']}\ \textsf{checker}$$

Partial Checkers.

$$\frac{\begin{array}{ll} M_1' * \alpha_1'.c(\beta_1) \sqsubseteq \alpha_2.c(\beta_2)[\Phi, \beta_2 \to \beta_1][\Phi'] & \alpha_2 \to \alpha_1, \alpha_2' \to \alpha_1' \in \Phi \\ M_1 \sqsubseteq M_2[\Phi'][\Phi''] & (\beta_1 \text{ fresh}) \end{array}}{M_1 * M_1'(\alpha_1 \rightsquigarrow \alpha_1') \sqsubseteq M_2 * \alpha_2.c(\beta_2) \mathrel{*\!\!-} \alpha_2'.c(\beta_2)[\Phi][\Phi'']}\ \textsf{assume}$$

$$\frac{\begin{array}{ll} M_1' \sqsubseteq \alpha_2'.c(\beta_2)[\Phi, \beta_2 \to \beta_1, \alpha_2' \to \alpha_1'][\Phi'] & \alpha_2 \to \alpha_1 \in \Phi \\ M_1 \sqsubseteq M_2[\Phi'][\Phi''] & (\alpha_2' \text{ fresh}) \end{array}}{M_1 * M_1'(\alpha_1' \rightsquigarrow) * \alpha_1.c(\beta_1) \mathrel{*\!\!-} \alpha_1'.c(\beta_1) \sqsubseteq M_2 * \alpha_2.c(\beta_2)[\Phi][\Phi'']}\ \textsf{apply}$$

Unfolding.

$$\frac{\begin{array}{l} M_1 \sqsubseteq M_2 * [\alpha_2, \beta_2/\pi, \rho][\vec{\delta}/\operatorname{fv}(M_i)]M_i[\Phi][\Phi'] \quad \alpha_2 \to \alpha_1 \in \Phi \quad (\vec{\delta} \text{ fresh}) \\ (\pi.c(\rho) := \cdots \vee \langle M_i \,;\, P_i \rangle \vee \cdots) \end{array}}{M_1 \sqsubseteq M_2 * \alpha_2.c(\beta_2)[\Phi][\Phi']}\ \textsf{unfold}$$

Finish.

$$\frac{}{\textsf{emp} \sqsubseteq \textsf{emp}[\Phi][\Phi]}\ \textsf{emp}$$

We write $M * M'(\alpha \rightsquigarrow \alpha')$ for splitting a graph into two sub-graphs: M', which is the slice from α to α' (all nodes and edges reachable from α but not from α'), and M, which is the remainder. Similarly, $M * M'(\alpha \rightsquigarrow)$ indicates M' is the slice from α (all nodes and edges reachable from α).

The unfold rule enables matching when M_1 is an unfolded instance of M_2. Algorithmically, we want to apply checker to match checker edges before considering unfolding. In the unfold rule, we write $\mathrm{fv}(M_i)$ for the free variables of M_i. For the overall approximation test on analysis states, we need to remember the P_i from unfolding to check the approximation relation on the pure constraints, which is left implicit here.

B Widening

In this section, we give the rewriting rules for the upper bound operation as described in Sect. 4.2.

$$\boxed{\Psi \,⨾\, (M_1 \triangledown M_2) \circledast M \;\rightharpoonup\; \Psi' \,⨾\, (M_1' \triangledown M_2') \circledast M'}$$

Edge Matching.

$$\frac{\langle\alpha_1, \alpha_2\rangle \in \Psi}{\begin{array}{l}\Psi \,⨾\, (M_1 * \alpha_1@f \mapsto r_1 \triangledown M_2 * \alpha_2@f \mapsto r_2) \circledast M \\ \quad \rightharpoonup \Psi, \langle r_1, r_2\rangle \,⨾\, (M_1 \triangledown M_2) \circledast M * \langle\alpha_1, \alpha_2\rangle@f \mapsto \langle r_1, r_2\rangle\end{array}} \;\textsf{m-pointsto}$$

$$\frac{\langle\alpha_1, \alpha_2\rangle \in \Psi}{\begin{array}{l}\Psi \,⨾\, (M_1 * \alpha_1.c(\beta_1) \triangledown M_2 * \alpha_2.c(\beta_2)) \circledast M \\ \quad \rightharpoonup \Psi, \langle\beta_1, \beta_2\rangle \,⨾\, (M_1 \triangledown M_2) \circledast M * \langle\alpha_1, \alpha_2\rangle.c(\langle\beta_1, \beta_2\rangle)\end{array}} \;\textsf{m-checker}$$

Folding.

$$\frac{\langle\alpha_1, \alpha_2\rangle \in \Psi \quad M_2' \sqsubseteq \alpha_2.c(\beta_2)[\alpha_2 \to \alpha_2] \qquad (\beta_2 \text{ fresh})}{\begin{array}{l}\Psi \,⨾\, (M_1 * \alpha_1.c(\beta_1) \triangledown M_2 * M_2'(\alpha_2 \rightsquigarrow)) \circledast M \\ \quad \rightharpoonup \Psi \,⨾\, (M_1 \triangledown M_2) \circledast M * \langle\alpha_1, \alpha_2\rangle.c(\langle\beta_1, \beta_2\rangle)\end{array}} \;\textsf{w-checker}$$

$$\frac{\begin{array}{l}\langle\alpha_1, \alpha_2\rangle, \langle\alpha_1', \alpha_2'\rangle \in \Psi \\ M_2' \sqsubseteq \alpha_2.c(\beta_2) \ast\!\!- \alpha_2'.c(\beta_2)[\alpha_2 \to \alpha_2, \alpha_2' \to \alpha_2'] \qquad (\beta_2 \text{ fresh})\end{array}}{\begin{array}{l}\Psi \,⨾\, (M_1 * \alpha_1.c(\beta_1) \ast\!\!- \alpha_1'.c(\beta_1) \triangledown M_2 * M_2'(\alpha_2 \rightsquigarrow \alpha_2')) \circledast M \\ \quad \rightharpoonup \Psi \,⨾\, (M_1 \triangledown M_2) \circledast M * \langle\alpha_1, \alpha_2\rangle.c(\langle\beta_1, \beta_2\rangle) \ast\!\!- \langle\alpha_1', \alpha_2'\rangle.c(\langle\beta_1, \beta_2\rangle)\end{array}} \;\textsf{w-partial}$$

$$\frac{\begin{array}{l}\langle\alpha_1, \alpha_2\rangle, \langle\alpha_1, \alpha_2'\rangle \in \Psi \\ M_2' \sqsubseteq \alpha_2.c(\beta_2) \ast\!\!- \alpha_2'.c(\beta_2)[\alpha_2 \to \alpha_2, \alpha_2' \to \alpha_2'] \qquad (\beta_1, \beta_2 \text{ fresh})\end{array}}{\begin{array}{l}\Psi \,⨾\, (M_1 \triangledown M_2 * M_2'(\alpha_2 \rightsquigarrow \alpha_2')) \circledast M \\ \quad \rightharpoonup \Psi \,⨾\, (M_1 \triangledown M_2) \circledast M * \langle\alpha_1, \alpha_2\rangle.c(\langle\beta_1, \beta_2\rangle) \ast\!\!- \langle\alpha_1, \alpha_2'\rangle.c'(\langle\beta_1, \beta_2\rangle)\end{array}} \;\textsf{w-aliases}$$