Relational Inductive Shape Analysis

Bor-Yuh Evan Chang
University of California, Berkeley
bec@cs.berkeley.edu

Xavier Rival
INRIA * and University of California, Berkeley
rival@di.ens.fr

Abstract

Shape analyses are concerned with precise abstractions of the heap to capture detailed structural properties. To do so, they need to build and decompose *summaries* of *disjoint* memory regions. Unfortunately, many data structure invariants require *relations* be tracked across disjoint regions, such as intricate numerical data invariants or structural invariants concerning back and cross pointers. In this paper, we identify issues inherent to analyzing relational structures and design an abstract domain that is parameterized both by an abstract domain for pure data properties and by user-supplied specifications of the data structure invariants to check. Particularly, it supports hybrid invariants about shape *and* data and features a generic mechanism for materializing summaries at the beginning, middle, or end of inductive structures. Around this domain, we build a shape analysis whose interesting components include a pre-analysis on the user-supplied specifications that guides the abstract interpretation and a widening operator over the combined shape and data domain. We then demonstrate our techniques on the proof of preservation of the red-black tree invariants during insertion.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords shape analysis, inductive definitions, heap analysis, separation logic, symbolic abstract domain, materialization

1. Introduction

Shape analyses define precise heap abstractions to provide the detailed aliasing and structural information often necessary for verification or program transformation tasks when typically no other static program analysis can. Most shape analyses are extremely effective when the analysis can be done non-relationally, that is, the property of interest can be decomposed so that the checking of one part is (mostly) independent of the checking of others. A significant challenge for almost all shape analyses is to step beyond non-relational abstractions, which become clearly necessary when combining structural shape analysis with numerical analyses.

To obtain the necessary precision, shape analyses rely on specialized descriptions for abstracting memory. In prior work (Chang et al. 2007), we proposed a shape analysis parameterized by inductively-defined predicates provided by the user. The novel aspect of our proposal is that these inductive definitions can come from checking code, that is, code that could be used to verify data structure instances dynamically. A nice property of using *invariant checkers* is that they are not only a familiar way for the developer to describe the data structure invariants but also express developer intent on how the data structure should be used.

In some respects, inductively-defined predicates are a natural fit for shape analysis (seemingly evidenced by the many shape analyses being built around them (Lee et al. 2005; Distefano et al. 2006; Berdine et al. 2007; Magill et al. 2007; Guo et al. 2007)). A key component of shape analysis is a *materialization* (i.e., a partial concretization) operation that then enables strong updates, which are critical for precision. With an inductively-defined predicate, a natural materialization operation is to unfold its definition. For example, consider the following definition for an acyclic doubly-linked list:

$$\texttt{l.dll}(\texttt{l}_{\text{prev}}) \;:=\; \textbf{if } (\texttt{l} = \textbf{null}) \textbf{ then } [] \;\; \textbf{else } \texttt{l@next -> n * l@prev -> l}_{\text{prev}} \texttt{ * n.dll(l)}$$

Here, we write inductive checkers in a pseudo-code notation that defines a class of memory regions by a traversal from a distinguished root pointer (the *traversal parameter*). The `[]` indicates an end to the traversal (i.e., an empty region), while `->` indicates the address and value of a field (i.e., a dereference of a field). The `*` indicates the components corresponding to disjoint memory regions (i.e., the traversal is allowed to dereference each object field once). New variables (e.g., `n`) are considered as local variables bound to the value of the specified field. In the above, `dll` says a doubly-linked list is either empty or has `next` and `prev` fields where `prev` must contain $\texttt{l}_{\text{prev}}$ and `next` must be a doubly-linked list whose `prev` is the current root pointer `l`. A singly-linked list checker is similar and can be obtained by simply dropping the constraint on `prev` and the $\texttt{l}_{\text{prev}}$ parameter. Circular lists can be described by adding a parameter for a distinguished "head node" and stopping the traversal when the head is reached (instead of **null**).

The kinds of invariant checkers considered in our earlier paper were non-relational. At a high-level, a non-relational checker describes each segment of the data structure independently. This includes checkers for singly-linked lists, singly-linked circular lists, trees, and skip lists, but *not* the `dll` checker. Syntactically, in a non-relational checker, the additional parameters of the checker (i.e., the state of the checker) are constant across all recursive calls. This condition clearly holds for a stateless checker like the one for a singly-linked list. In contrast, the `dll` checker uses an additional parameter $\texttt{l}_{\text{prev}}$ to specify that the `prev` field of the next element points to the current element.

The main difficulty with relational checkers is that simply unfolding instances into their definitions does not reveal these relations. Thus, code that utilizes such relations are not analyzable

* Abstraction Project-team, shared with CNRS and École Normale Supérieure, Paris

POPL'08, January 7–12, 2008, San Francisco, California, USA.

without other techniques. With the dll checker, an unfolding reveals that next points to another dll, but not that prev must also point to a segment of dll, nor that next and prev are inverses (i.e., following next and then prev gets back to the same node). As such, analyzing code that traverses a doubly-linked list using the next field with the dll checker is easier than analyzing code that traverses using the prev field. Part of the problem is that there are a number of ways to traverse a doubly-linked list (i.e., a number of inductive schemes). For example, an alternative checker could start at the tail of the list following prev fields, but then the above difficulty is simply reversed for the fields.

The relational issue becomes even more salient when we consider inductive invariants with more involved data constraints than pointer equality. For example, consider the following checkers:

```
t.bst(t_lo, t_up) :=
  if (t = null) then [] && t_lo < t_up
  else t@l -> l * t@d -> d * t@r -> r
       * l.bst(t_lo, d) * r.bst(d, t_up) && t_lo < d < t_up
l.listn(l_len) := if (l = null) then [] && l_len = 0
                  else l@next -> n * n.listn(l_len − 1)
```

The bst checker describes a binary search tree, while listn specifies a list of a particular length. Each of these relational checkers uses parameters to capture very different kinds of relations and thus pose different challenges. The bst checker enforces a global ordering property on the data fields (d) with the range narrowing in recursive calls (*the shape constrains the data*), while listn uses l_{len} to specify the recursion depth (*the data constrains the shape*). Finally, the dll checker describes a kind of *invertible structural invariant* with a data structure that points to previous roots.

In this paper, we identify a dividing line in the complexity between non-relational and relational inductive shape analysis. We extend our earlier work to the relational case and make the following contributions:

- We propose a parametric abstract domain for relational inductive shape analysis. Our domain is not only parameterized by programmer-supplied invariant checkers, but also by a numerical domain for data constraints (Section 3). In order to support relational shape analysis, we strengthen prior notions of partial checker runs, which are used to abstract memory regions where user-supplied invariants hold only partially.
- We observe that shape analysis with invertible checkers, such as dll, can be strengthened with a notion of *backward unfolding*. The novel aspect of our proposal is that the backward unfolding operation can be derived automatically from the standard forward unfolding (Section 5.1). Another interesting aspect is that we use a pre-analysis on checkers, in particular, a type inference, to guide the abstract interpretation (Section 4).
- We describe how our shape domain interacts with example base data domains in the abstract interpretation phase to compute precise fixed-point invariants. In particular, we show that the widening requires careful coordination between shape and data (Section 5.2).

In the next section, we highlight the challenges that we aim to address in this paper with an example analysis of red-black tree insertion. After describing our analysis, we return to this example in Section 7 to indicate how our algorithm can be used to verify the correctness of operations on such complex data structures.

2. Background and Overview

To set the stage for this paper, we first present the basic ideas of inductive checker-based shape analysis by tracing through the example in Figure 1. Our shape analysis is a standard forward abstract interpretation (Cousot and Cousot 1977) that computes an

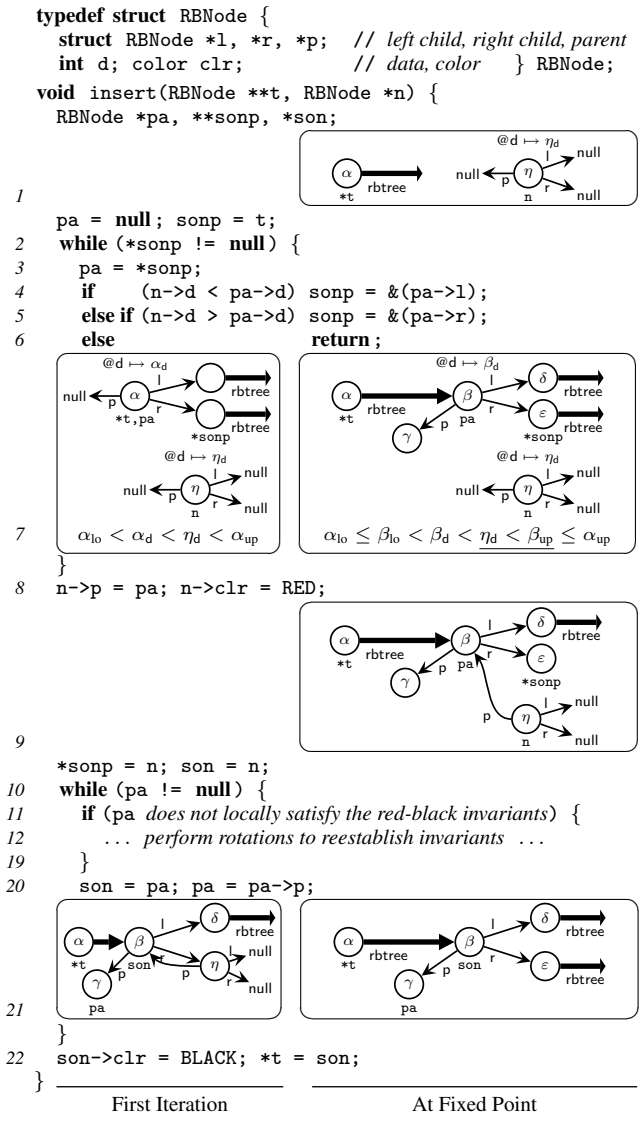


Figure 1. Red-black tree insertion in C.

abstract memory state at each program point. In the figure, we show the abstract memory state of the analysis at a number of program points using a graphical notation. For the program points inside loops, we give two memory states: one for the first iteration (left) and one for the fixed point (right). Our memory abstraction is built around using user-supplied inductive checkers to summarize memory regions. Intuitively, a programmer-defined checker describes the class of memory regions arranged according to particular constraints on which an execution of the checker would succeed. In a graph, each node denotes a value (e.g., a memory address) and, when necessary, is labeled by a *symbolic value*. Symbolic values are existentially quantified variables used to name heap objects. To distinguish them from program variables, we use lowercase Greek letters ($\alpha, \beta, \gamma, \ldots$). A program variable (e.g., pa) below a node indicates that the value of that variable is that node. Edges describe disjoint memory regions. A thin edge gives a *points-to* relationship, that is, a memory cell whose address is the source node and whose value is the destination node. To keep the diagram compact, we draw points-to edges only for the pointer fields, and, when necessary, we notate the values of data fields above the node (e.g., at pro-

gram point 7). A thick edge (a *checker* edge) summarizes a memory region, that is, some number of points-to edges with certain properties. 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 memory regions that satisfy particular checkers (e.g., on line 1, the complete checker edge labeled `rbtree` says the memory region from α satisfies the red-black tree checker). To describe memory states at intermediate program points, partial checker edges capture the notion of a checker that holds on just a segment of a data structure. For example, at program point 7 in the fixed-point graph, the partial checker edge from α to β summarizes a memory region that is a `rbtree` along that segment. For data constraints, our memory abstraction is parameterized by a *base abstract domain* whose coordinates (i.e., variables) are the symbolic values. In the examples, we note the data constraints that are necessary to get the desired results and assume the base domain can capture them.

A memory update is captured in the graph by modifying the appropriate points-to edges (performing strong updates). For example, consider the transition from program point 7 to 9 where we see the updating of `n`'s parent and color fields (line 8). In the figure, we show only the disjunct where `sonp` went right (`r`) on the final iteration; the case for the `l` field is similar. Sometimes, checker edges are *unfolded* to materialize the points-to edges for an update. For example, the `rbtree` checker edge (as on line 1) is unfolded to materialize the `l` and `r` fields on lines 4 and 5. To reach a fixed point, we *fold* subgraphs into checker edges. We determine where to fold in the graph (as to retain enough precision) by consulting the iteration history (as described in Chang et al. (2007)).

The analysis of the red-black tree insertion routine poses a number of challenges that we aim to address in this paper. These challenges appear in both the unfolding to materialize needed points-to edges and the folding to retain sufficient precision in the data constraints. In particular, while the non-relational shape analysis described in our prior work could obtain the structural invariants (i.e., graphs) at program points 1, 7, and 9, it would fail to infer the key data constraints and even the fixed-point graph at point 21.

To better understand these challenges, we first describe informally the invariants of the red-black tree we consider here. The *inverse invariant* (cf., doubly-linked lists) specifies that the `p` field is the inverse of the `l` and `r` fields (i.e., for each node `n`, if `n->l` $\neq$ **null**, then `n->l->p` $=$ `n` and analogously for the `r` field). Next, the *order invariant* (cf., binary search trees) says that for each node `n`, the values in the left subtree (i.e., the value of the `d` fields in the nodes reachable from `n->l`) are less than `n->d`, and the values of the right subtree are greater than `n->d`. Finally, the *balance invariant* (cf., lists of a given length) is as follows: (a) a node is either red or black (given by the `clr` field); **null** is considered black; (b) the root is black; (c) both children of a red node are black; (d) every simple path (i.e., following `l` and `r` fields) from a node to a leaf contains the same number of black nodes. Observe that each of these invariants describes a relation between nodes, and in the case of the order and balance invariants, they describe a global relation on all the nodes of the tree. It is clear that to write code that checks this invariant, a checker needs to carry some state (in order to turn the global relations specified above into local checks). In terms of checkers, we define the following red-black checker for a node `t`:

```
t.rbtree(t_p, t_lo, t_up, t_redok, t_bh) :=
  if (t = null) then [] && t_lo < t_up && t_bh = 0
  else t@l -> l * t@r -> r * t@p -> t_p * t@d -> d * t@clr -> c
       * l.rbtree(t, t_lo, d, c ≠ red, ite(c = red, t_bh, t_bh − 1))
       * r.rbtree(t, d, t_up, c ≠ red, ite(c = red, t_bh, t_bh − 1))
       && t_lo < d < t_up && (c ≠ red || t_redok)
```

where `ite` is an if-then-else expression. The key point is that the additional parameters are used to impose the following constraints: $\texttt{t}_\text{p}$ is where the `p` field should point; $\texttt{t}_\text{lo}$ is a lower bound on the `d` field; $\texttt{t}_\text{up}$ is an upper bound on the `d` field; $\texttt{t}_\text{redok}$ gives whether the `clr` is allowed to be red; $\texttt{t}_\text{blkht}$ is the number of black nodes on all paths to a leaf (i.e., the black height). We can see that the kinds of relations used by the `rbtree` checker have parallels to each of the relational checkers presented in Section 1. As such, the red-black tree insertion routine is a fairly representative example of the kinds of challenges in relational shape analysis. In the example figure, we have elided the additional parameters on the instances of `rbtree`. Instead, we adopt the convention of referring to the additional parameters by subscripting the node name on which the checker applies. For example, the checker edge on line 1 conveys the following: $\alpha.\texttt{rbtree}(\alpha_\text{p}, \alpha_\text{lo}, \alpha_\text{up}, \alpha_\text{redok}, \alpha_\text{bh})$ where any constraints on the additional parameters are given in the data domain.

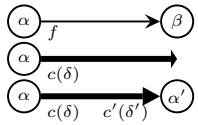
Returning to the example analysis in Figure 1, we comment on the points where the relational aspect of the `rbtree` checker poses obstacles to overcome in the analysis. First, consider the memory state at program point 21 in the first iteration. Note that the value of `pa` is γ, which has no outgoing edges from it (neither points-to nor checker). However, on the next iteration, we will analyze statements that access the fields of `pa` and thus need to materialize them. From our intuitive understanding of the `rbtree` checker, we know that γ lies on the segment between α and β (when it is non-empty), so if we were to unfold the segment *backward* from β, we could materialize the fields of γ. To justify the unfolding of partial checker edges, we must strengthen the logical representation of such segments (Section 3.1). The novel aspect of our proposal is that we determine when to apply this backward unfolding automatically using a pre-analysis on checkers (Section 4). We also reduce the soundness justification of backward unfolding to that of forward unfolding (Section 5.1.1). In other words, while the user supplies the forward unfolding axiom (as an inductive checker definition), the backward unfolding is derived automatically and not axiomatized (cf., Berdine et al. (2007)).

Second, consider the memory states at program point 7, which is a location where the combination of shape and relational data constraints poses challenges. We get the data constraints in the first iteration simply by unfolding `rbtree` and from the guard on the conditional. The challenge is on widening to generate the fixed-point invariant. In this example, we need the underlined constraint to know that the insertion preserves the ordering invariant, but it is not necessarily easy to obtain. At a high-level, the core of the difficulty is that while the top of the tree can be fairly easily summarized into the segment from α to β, the data invariant we desire requires synthesizing relations between the prefix segment (α to β) and the suffix (from β). To address this difficulty, we make some observations that allow us to apply of some standard analysis techniques in this context. One, we observe that because the coordinates of the data domain are given by heap nodes, the data domain is rather sensitive to the large changes that result from widening in the shape domain. Thus, we delay widening elements of the data domain until the shape portion has converged (keeping symbolic joins instead). Two, we notice that we can separate this synthesis task into finding appropriate arguments for the checker parameters and inferring the appropriate relation on those arguments. Specifically, we can make the finding of checker arguments shape-guided by using additional data fields that correspond to the checker parameters.

3. Memory Abstraction with Inductive Segments

The core component of our analysis state is an abstract memory state M (as shown in Figure 2). The memory state is based largely on separation logic (Reynolds 2002), so we use notation that is borrowed from there. The first three items are as in separation logic: `emp` is the empty memory, $\alpha@f \mapsto \beta$ is a points-to relation describing a single memory cell, and $M_0 * M_1$ is the separating

memories	$M ::= \mathsf{emp}$		empty
	$\mid \alpha@f \mapsto \beta$		memory cell
	$\mid M_0 * M_1$		disjoint regions
	$\mid \alpha.c(\delta)$		checker region
	$\mid \alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$		checker segment
data constraints	$P \in \mathbb{P}^\sharp$		
environment	$E ::= \cdot \mid E, x \mapsto \alpha$		
analysis state	$A ::= \bot \mid \langle E, M, P\rangle \mid A_1 \vee A_2$		
symbolic values	$\alpha \in \mathbf{Val}^\sharp$	field names	f
program variables	$x \in \mathbf{Var}$	checker names	c



$\alpha@f \mapsto \beta$

$\alpha.c(\delta)$

$\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$

Figure 2. Analysis state.

conjunction specifying a memory that can be divided into two disjoint regions (i.e., with disjoint domains), which together can symbolically describe a finite memory. A field offset expression $\alpha@f$ corresponds to the base address α plus the offset of field f (i.e., `&(a.f)` in C). For simplicity, we assume that all pointers occur as fields in a `struct`. Also for simplicity in presentation, we consider only symbolic values on the right side of points-to (β) (i.e., for the contents of memory cells) and assume any additional constraints on those values are captured elsewhere (e.g., is `null`). The next two items are used to summarize memory regions. An application of a user-supplied checker $\alpha.c(\delta)$ describes a memory region where c succeeds when applied to α and δ. For presentation, we write checkers with one traversal parameter and one additional parameter, though everything applies to checkers with zero-or-more additional parameters. The last item provides a generic mechanism for specifying segments of user-supplied checkers, which we will describe further below. In Figure 2, we show the correspondence between formulas and graphs. The thick edges (for checker regions and checker segments) can be intuitively thought of as representing possible subgraphs of thin points-to edges arranged in particular shapes and with certain constraints.

Inductive Checker Definitions. Formally, the definition of a checker c, with traversal parameter π and a sequence of zero-or-more additional parameters $\overrightarrow{\rho}$, is a finite disjunction of rules. A rule R consists of a conjunction of a memory portion M and a pure portion F (given as a first-order formula). Further, M is separated into two parts: an unfolded region M^{u} given by a series of points-to edges and a folded region M^{f} given by a series of checker applications. Schematically, we write checker definitions as follows:

$$\pi.c(\overrightarrow{\rho}) \;:=\; \langle M_0^{\mathrm{u}} * M_0^{\mathrm{f}}, F_0\rangle \vee \cdots \vee \langle M_n^{\mathrm{u}} * M_n^{\mathrm{f}}, F_n\rangle$$

Free variables in the rules are considered as existential variables bound at the definition. Note that because we view checkers as code, the kinds of inductive predicates are further restricted. In particular, M^{u} correspond to finite access paths from π and thus existentials are only for the values of fields along those access paths from π. Each checker call in M^{f} is applied to arguments among the parameters (the traversal or the additional ones) or the existentials introduced in M^{u}. These kinds of inductive definitions are apt for analysis, as we know that unfolding such a definition corresponds to materializing points-to edges from π.

Inductive Segments. Inductive predicates from checkers, such as `dll` and `rbtree`, give us rather precise summaries of memory regions, but unfortunately, they are typically not general enough to capture the invariants of interest at all program points. For example, in Figure 1 at program points 7 and 9, we need to summarize the region between α and β (i.e., between the root pointer $*\mathtt{t}$ and the cursor pointer `pa`). As observed in our prior work (Chang et al. 2007), such segment regions are captured by a *partial checker run*. In terms of inductively-defined predicates, we want to describe a segment as a partial derivation (i.e., a derivation with a hole in a subtree). In the past work, we have used the standard separating implication $*\!\!-$ from separation logic for this purpose. While sufficient and useful for summarizing segments, this definition does not allow for unfolding. As noted in Section 2, unfolding of segments, particularly backward unfolding, is often necessary with relational checkers (e.g., the rebalancing of the red-black tree in Figure 1).

In this paper, we strengthen this connector by imposing an inductive structure. This additional structure then enables the definition of forward and backward unfolding schemes on segments. Informally, $\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$ is a memory region where it satisfies $\alpha.c(\delta)$ up to some number of unfoldings and conjoining any disjoint memory that satisfies $\alpha'.c'(\delta')$ makes the combined region satisfy $\alpha.c(\delta)$. In Section 3.1, we discuss this definition in further detail, and in Section 5.1.1, we describe and justify the unfolding operations on segments.

Analysis State. To track data constraints (i.e., non-points-to constraints), we maintain a pure state P, which we assume is an element of an abstract domain $\mathbb{P}^\sharp$. Note that the base data domain $\mathbb{P}^\sharp$ is a parameter of the analysis. To connect the abstract memory with the program, we also keep an environment E that maps program variables to symbolic values that denote their addresses. Finally, the overall analysis state A is a finite disjunction of $\langle E, M, P\rangle$ tuples.

3.1 Semantics of the Memory Abstraction

In this subsection, we give the semantics of our memory abstraction. We focus mostly on segments, as other aspects of the graph follow mostly from separation logic. We write $u \in \mathbf{Val}$ for concrete values and make no distinction between addresses and values. Further, we write $u + f$ for the base address u plus the offset of field f. A *concrete store* $\sigma\colon \mathbf{Val} \rightharpoonup_{\text{fin}} \mathbf{Val}$ maps addresses into values. We write $[\cdot]$ for the empty concrete store and $[u_0 \mapsto u_1]$ for the store of one cell mapping u_0 to u_1. A compound store $\sigma_0 * \sigma_1$ is a store with disjoint substores σ_0 and σ_1 where σ_0 and σ_1 must have disjoint domains (i.e., overloading the operators from the abstract memory). To capture relations among memory cells, we consider a *valuation* that essentially assigns an interpretation to symbolic values. More precisely, a valuation $\nu\colon \mathbf{Val}^\sharp \to \mathbf{Val}$ is a mapping from symbolic values into concrete values.

Checker Regions. We write $\langle \sigma, \nu\rangle \models M$ to mean a concrete store σ and a valuation ν satisfy an abstract memory M. The semantics of a checker application is defined by induction over the "height of the underlying calling tree". We write $\alpha.c^i(\delta)$ for a checker application of height at most i. We also write $[\alpha/\pi]M$ for substituting α for π in M. Now, we define $\models$ as follows:

$\langle [\cdot], \nu\rangle \models \mathsf{emp}$ (always, for all ν)

$\langle [\nu(\alpha) + f \mapsto \nu(\beta)], \nu\rangle \models \alpha@f \mapsto \beta$ (always, for all ν)

$\langle \sigma_0 * \sigma_1, \nu\rangle \models M_0 * M_1$ iff $\langle \sigma_0, \nu\rangle \models M_0$ and $\langle \sigma_1, \nu\rangle \models M_1$

$\langle \sigma, \nu\rangle \models \alpha.c(\delta)$ iff there exists an i such that $\langle \sigma, \nu\rangle \models \alpha.c^i(\delta)$

$\langle \sigma, \nu\rangle \models \alpha.c^{i+1}(\delta)$

iff there exists a rule $\langle M^{\mathrm{u}} * M^{\mathrm{f}}, F\rangle$ in the definition of $\pi.c(\rho)$ where $M^{\mathrm{f}} = \beta_0.c_0(\gamma_0) * \cdots * \beta_m.c_m(\gamma_m)$ and such that

ν satisfies $[\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F$ and

$\langle \sigma, \nu\rangle \models [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]M^{\mathrm{u}} * \beta_0.c_0^i(\gamma_0) * \cdots * \beta_m.c_m^i(\gamma_m)$

where $\vec{\kappa}$ are the free variables of the rule and $\vec{\varepsilon}$ are fresh.

Intuitively, a checker application of height 1 should make no recursive calls (i.e., it should correspond to a case where M^{f} is emp). Observe that the valuation ν is what connects regions and thus allows checkers to be relational.

Segment Regions. In a similar way, the semantics of a segment $\alpha.c(\delta) \ *\!\!= \ \alpha'.c'(\delta')$ is defined by induction over the number of checker rule applications needed to build a derivation of $\alpha.c(\delta)$ from a derivation of $\alpha'.c'(\delta')$. For this purpose, we add an index i on segments to indicate its length (when it is known). Thus, the standard segment is one of zero-or-more steps:

$\langle \sigma, \nu \rangle \models \alpha.c(\delta) \ *\!\!= \ \alpha'.c'(\delta')$
iff there exists an i such that $\langle \sigma, \nu \rangle \models \alpha.c(\delta) \ *\!\!=^{i} \ \alpha'.c'(\delta')$

The definition of $*\!\!=^{i}$ then proceeds by induction on i. Intuitively, we want a 0-step segment to be an "empty partial derivation" of $\alpha.c(\delta)$, which means it should be case that the checkers and arguments match and correspond to an empty store. Formally,

$\langle [\cdot], \nu \rangle \models \alpha.c(\delta) \ *\!\!=^{0} \ \alpha'.c(\delta')$ iff $\nu(\alpha) = \nu(\alpha')$ and $\nu(\delta) = \nu(\delta')$

The main purpose of these restrictions is to guarantee that $*\!\!=$ has many of the same properties as the separating implication $*\!\!-$. Moreover, in Section 5.1.1, we will see that these restrictions are critical for the backward unfolding (as used in the red-black tree example of Figure 1). The definition of the inductive case is very similar to the corresponding case for checker applications. For an $(i+1)$-step segment, we unfold the head checker c once materializing points-to edges and a series of recursive checker calls where one of these checker calls should be replaced with a segment of rank i. More precisely,

$\langle \sigma, \nu \rangle \models \alpha.c(\delta) \ *\!\!=^{i+1} \ \alpha'.c'(\delta')$
iff there exists a rule $\langle M^{\mathrm{u}} * (M^{\mathrm{f}} * \beta.c''(\gamma)), F \rangle$ in the definition of $\pi.c(\rho)$ such that

$$\nu \text{ satisfies } [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F \quad \text{and}$$
$$\langle \sigma, \nu \rangle \models [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]M^{\mathrm{u}} * M^{\mathrm{f}} * (\beta.c''(\gamma) \ *\!\!=^{i} \ \alpha'.c'(\delta'))$$

where $\vec{\kappa}$ are the free variables of the rule and $\vec{\varepsilon}$ are fresh.

Concretization. Finally, we tie these definitions together by giving the concretization of our abstract domain, which is a reduced product of the shape domain (of graphs) and the base data domain. As the concrete counterpart of the environment E, we write $\theta \colon \mathbf{Var} \to \mathbf{Val}$ for a *concrete environment* that maps variables to concrete values. Here, we overload the concretization operator $\boldsymbol{\gamma}$ to apply to each of the component parts.

Definition 1 (Concretization)**.**

$$\begin{aligned} \boldsymbol{\gamma}(\, M \,) &= \{\, (\sigma, \nu) \mid \langle \sigma, \nu \rangle \models M \,\} \\ \boldsymbol{\gamma}(\, \langle M, P \rangle \,) &= \{\, (\sigma, \nu) \mid (\sigma, \nu) \in \boldsymbol{\gamma}(M) \wedge \nu \in \boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P) \,\} \\ \boldsymbol{\gamma}(\, \langle E, M, P \rangle \,) &= \{\, (\theta, \sigma) \mid \exists \nu,\ \theta = \nu \circ E \wedge (\sigma, \nu) \in \boldsymbol{\gamma}(\langle M, P \rangle) \,\} \end{aligned}$$

We write $\boldsymbol{\gamma}_{\mathbb{P}^\sharp}$ for the concretization function in the base domain, which yields a set of satisfying valuations (independent of a store).

3.2 Properties of Inductive Segments

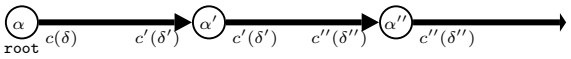
While inductive segments $*\!\!=$ are specialized to inductive checkers, it still has many of the properties of the separating implication $*\!\!-$ (when applied to checkers). We want to be able to unfold segments, but we also need to maintain the properties necessary for analysis. In particular, we can prove that $*\!\!=$ is a restriction of $*\!\!-$ (by induction over the length of the segment). The semantics of $*\!\!-$ is the standard one in separation logic.

Theorem 1 (Stronger than Separating Implication)**.**
If $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \ *\!\!= \ \alpha'.c'(\delta') \,)$,
then $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \ *\!\!- \ \alpha'.c'(\delta') \,)$.

As a consequence, we get the elimination rule.
If $(\sigma, \nu) \in \boldsymbol{\gamma}(\, (\alpha.c(\delta) \ *\!\!= \ \alpha'.c'(\delta')) \ * \ \alpha'.c'(\delta') \,)$,
then $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \,)$.

We also prove the following basic but important properties.
If $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \ *\!\!= \ \alpha'.c'(\delta') \ * \ \alpha'.c'(\delta') \ *\!\!= \ \alpha''.c''(\delta'') \,)$,
then $(\sigma, \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \ *\!\!= \ \alpha''.c''(\delta'') \,)$.
For all ν, $([\cdot], \nu) \in \boldsymbol{\gamma}(\, \alpha.c(\delta) \ *\!\!= \ \alpha.c(\delta) \,)$.

In terms of the shape graph, these facts allow the analysis to discard intermediate nodes when they are no longer needed, such as α' and α'' in the following graph:



Furthermore, it allows us to discover when a region again satisfies a complete run of a checker (i.e., the entire structure again adheres to the data structure invariant). In the above, dropping the intermediate nodes allows us to derive that the region from α (pointed to by `root`) satisfies the invariants of checker c. The last fact allows us to introduce segments anywhere when needed.

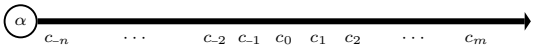
We remark that our notion of segments is designed to capture precisely a partial run (i.e., a partial derivation) and not only structure segments where an empty structural segment need not necessarily have equal additional parameters (i.e., δ's). This distinction shows up in the above properties, which are crucial to our analysis.

4. Typing Checker Parameters

As alluded to in Section 2, we perform a pre-analysis on checker definitions to gather information that we then use to guide the shape analysis. In particular, we saw that at program point 21 in Figure 1, in order to materialize the fields of γ (i.e., the node pointed to by `pa`), we need to unfold the segment between α and β "backward" from β. However, note that this reasoning was based only on our intuitive understanding of the `rbtree` checker. Nevertheless, we notice that with some additional type information on the checker parameters, the analysis can then make this unfolding decision automatically.

In this section, we prescribe a type system to checker parameters that classifies them into various kinds of pointers and non-pointers. This type information will then instruct the shape analysis where to perform unfolding (see Section 5.1). Intuitively, we consider a checker parameter to have pointer type if it appears on the left side of points-to in some disjunct of a checker unfolding. That is, in terms of checker runs, a parameter has pointer type if one of its fields is ever dereferenced along some path in a checker run (on a satisfying store). Thus far in the paper, we have in essence implicitly assigned a pointer type to the traversal parameter (i.e., π in a definition $\pi.c(\rho)$).

To direct unfolding decisions, we introduce further refinements of the pointer type to indicate, for example, which fields f are dereferenced. We thus write a pointer type as a record of fields that are dereferenced $\{f_0\langle \ell_0 \rangle, \ldots, f_n\langle \ell_n \rangle\}$ (in some disjunct). We consider the non-pointer type to be simply the empty record of fields $\{\}$. The level ℓ provides a relative measure of where in the sequence of checker calls a field is dereferenced (i.e., where the points-to edge is materialized). To describe this notion, consider a derivation of a checker application $\alpha.c_{-n}()$, shown below diagrammatically:



where the c_i's are the sequence of checker calls (of various checkers) through a path in the derivation (i.e., a path in the computation tree of $\alpha.c_{-n}()$). For a pointer argument β of a call, such as at c_0, we want to track an approximation of where along the run are the fields of β dereferenced (i.e., materialized). A level ℓ is thus an integer indicating in how many checker calls is the field dereferenced or `unk` if unknown (e.g., at different levels depending on a conditional); negative integers indicate backward from the current call, while positive integers indicate forward.

Example 1 (Typing the Doubly-Linked List Checker)**.** The following assigns parameters types to the `dll` checker from Section 1.

types $\tau ::= \{f_0\langle \ell_0\rangle, \ldots, f_n\langle \ell_n\rangle\}$
levels $\ell ::= n \mid \mathsf{unk}$

$\boxed{\Gamma \vdash_\Sigma M \ \mathsf{ok}}$

$$\frac{}{\Gamma \vdash \mathsf{emp}\ \mathsf{ok}}\ \textsf{t-emp} \qquad \frac{\{f\langle 0\rangle\} <: \Gamma(\alpha) \quad \Gamma \vdash M\ \mathsf{ok}}{\Gamma \vdash M * \alpha@f \mapsto \beta\ \mathsf{ok}}\ \textsf{t-field}$$

$$\frac{\Gamma(\alpha_i) - 1 = \Sigma(c)(\pi_i) \quad \Gamma \vdash M\ \mathsf{ok} \quad (\text{for } \pi_0.c(\pi_1) := \ldots)}{\Gamma \vdash M * \alpha_0.c(\alpha_1)\ \mathsf{ok}}\ \textsf{t-checker}$$

$\boxed{\vdash_\Sigma \mathit{chkdef}\ \mathsf{ok}}$

$$\frac{\Sigma(c) \vdash M_i\ \mathsf{ok}}{\vdash \pi.c(\rho) := \langle M_0, F_0\rangle \vee \cdots \vee \langle M_n, F_n\rangle\ \mathsf{ok}}\ \textsf{t-chkdef}$$

$\boxed{\tau_0 <: \tau_1 \quad \ell_0 <:: \ell_1}$

$$\frac{\ell_i <:: \ell'_i \quad (0 \le i \le n \le m)}{\{f_0\langle \ell_0\rangle, \ldots, f_n\langle \ell_n\rangle\} <: \{f_0\langle \ell'_0\rangle, \ldots, f_m\langle \ell'_m\rangle\}} \qquad \frac{}{\ell <:: \mathsf{unk}} \qquad \frac{}{\ell <:: \ell}$$

Figure 3. Type-checking checker parameters.

$$(\pi : \{\mathsf{next}\langle 0\rangle, \mathsf{prev}\langle 0\rangle\}).\mathsf{dll}(\rho : \{\mathsf{next}\langle -1\rangle, \mathsf{prev}\langle -1\rangle\}) :=$$
$$\exists(\ \gamma : \{\mathsf{next}\langle 1\rangle, \mathsf{prev}\langle 1\rangle\}).\ \langle \mathsf{emp}, \pi = \mathsf{null}\rangle$$
$$\vee\ \langle \pi@\mathsf{next} \mapsto \gamma * \pi@\mathsf{prev} \mapsto \rho * \gamma.\mathsf{dll}(\pi), \pi \neq \mathsf{null}\rangle$$

Example 2 (An Alternative Doubly-Linked List Checker)**.** The following pair of checkers define a doubly-linked list where the unfolding materializes the next field of the current node and then the prev field of the next node (if it is non-null) instead of the next and prev fields of the current node as in dll.

$$(\pi : \{\mathsf{next}\langle 0\rangle, \mathsf{prev}\langle -1\rangle\}).\mathsf{npdll}() :=$$
$$\exists(\ \gamma : \{\mathsf{next}\langle 2\rangle, \mathsf{prev}\langle 1\rangle\}).\ \langle \pi@\mathsf{next} \mapsto \gamma * \gamma.\mathsf{npdll}'(\pi), \pi \neq \mathsf{null}\rangle$$
$$(\pi : \{\mathsf{next}\langle 1\rangle, \mathsf{prev}\langle 0\rangle\}).\mathsf{npdll}'(\rho : \{\mathsf{next}\langle -1\rangle, \mathsf{prev}\langle -2\rangle\}) :=$$
$$\langle \mathsf{emp}, \pi = \mathsf{null}\rangle\ \vee\ \langle \pi@\mathsf{prev} \mapsto \rho * \pi.\mathsf{npdll}(), \pi \neq \mathsf{null}\rangle$$

In Figure 3, we present a type-checking algorithm for the pointer types described above, which will then lead to an inference algorithm. We write Γ for a type environment mapping symbolic values α to types τ and Σ for a checker environment mapping checker names c to a type environment Γ for the types of all the parameters of c. Also, we assume that $\Sigma(c)$ includes types for all the free symbolic values in the definition of c (i.e., the existentially quantified variables). Both the types and levels form (semi-)lattices where the ordering is given by the $<:$ and $<::$ relations, respectively. For types, records are ordered by subset containment of fields modulo ordering in the levels, while the level lattice is the flat lattice on integers with unk as the top element. Intuitively, the absence of a field indicates it is not dereferenced anywhere. The core judgment is $\Gamma \vdash_\Sigma M$ ok, which checks that an abstract memory is well-formed with respect to the parameter types Γ and checker environment Σ, which are fixed throughout. For a points-to edge, we check that the field is in the set of dereferenced fields (rule t-field). For a checker application, we check that the set of dereferenced fields for the actuals and formals are the same, but we need to shift the frame of reference of the levels (rule t-checker). We write $\tau - 1$ for the function on types that decrements each of the field levels (and where unk maps to unk). Finally, at the top-level, we typecheck each checker separately and for each checker, we check the memory specification of each rule.

The type-checking algorithm is fairly straightforward, but it also yields a natural extension to inferring parameter types by a least fixed-point computation. The inference proceeds by initializing the global environment Σ to map all symbolic values to $\{\}$ (the bottom element). At each place where we check conformance in Figure 3, we instead compute the join in the type lattice and update the appropriate environments. Then, we iterate until we reach a fixed point. Since the type lattice has finite height (as the number of fields is fixed for any set of checker definitions), the process terminates.

This procedure provides fairly generic support for unfolding (forward and backward). The graph unfolding described in Section 5.1 combined with this pre-analysis allows support for backward pointers that go back a finite number of steps (e.g., a list with a pointer that goes back two nodes), but more importantly, it makes the unfolding process less sensitive to how the checkers are written. For example, the alternative doubly-linked list checker of Example 2 works equally well. The types are slightly different, which in turn guides the graph unfolding algorithm appropriately. Also, observe that in t-checker, we make no distinction between the traversal parameters and the additional parameters. Thus, with this type information, one could consider checkers with, for example, multiple forward parameters (at least in terms of unfolding).

5. Analysis Algorithm

Our analysis works as a typical abstract interpretation on programs. In particular, the analyzer computes an abstract state A for each control point. To do that, we need transfer functions for commands, such as assignment and condition testing. As alluded to in Section 2, the key domain operation is the unfolding of checker edges (forward and backward) in order to *materialize* points-to edges. A novel aspect of our proposal is that we reduce the problem of unfolding segments backward to unfolding them forward (Section 5.1). To infer loop invariants and obtain a terminating analysis, we define comparison and join operations on abstract states that summarize graphs by folding them into checker edges. The primary source of complexity in folding is the interaction between the shape graph and the base data domain. A nice property of our algorithm is that at a high-level, we separate the computation of the comparison and join operations into phases: first, a traversal over the shape graph gathering constraints, and then, a propagation of these constraints to the base data domain before applying the corresponding base domain operation (Section 5.2).

5.1 Abstract Transition with Segment Unfolding

Recall that the complete checker edges and partial checker edges summarize memory regions. In order to reflect memory updates in a precise manner, we often need to partially concretize these summaries, which we do by unfolding. To describe unfolding in detail, we consider a schematic example of doubly-linked list traversals that illustrate the various forms of unfolding (shown in Figure 4). Initially, we have that `l` points to the head of a doubly-linked list (satisfying the dll checker from Section 1). From program point 3 to 4, we perform a forward unfolding of a complete checker edge to materialize the edge corresponding to $\mathtt{c}_0\mathtt{->next}$ (the one from α in the first iteration and the one from ψ in the fixed-point iteration). Observe that the analysis determines that it should, for example, unfold forward at α in the first iteration because while there is no outgoing points-to edge for the next field (corresponding to $\mathtt{c}_0\mathtt{->next}$), node α does have an outgoing checker edge (i.e., α is the traversal argument to a checker). Because checkers are inductive definitions where points-to edges emanate from the traversal parameter, unfolding the inductive predicate at its traversal argument (e.g., α in $\alpha.\mathsf{dll}(\mathsf{null})$) is likely to materialize the desired points-to edge.

From program point 7 to 8, to materialize the edge $\mathtt{c}_1\mathtt{->next}$, we must unfold forward at the head of a dll segment (the one from α to ω in the first iteration and the one from ζ to ω in the fixed-point iteration). Like in the previous case, the analysis determines it should unfold forward because it arrives at an outgoing checker edge, but in this case, it is a partial checker edge and thus how

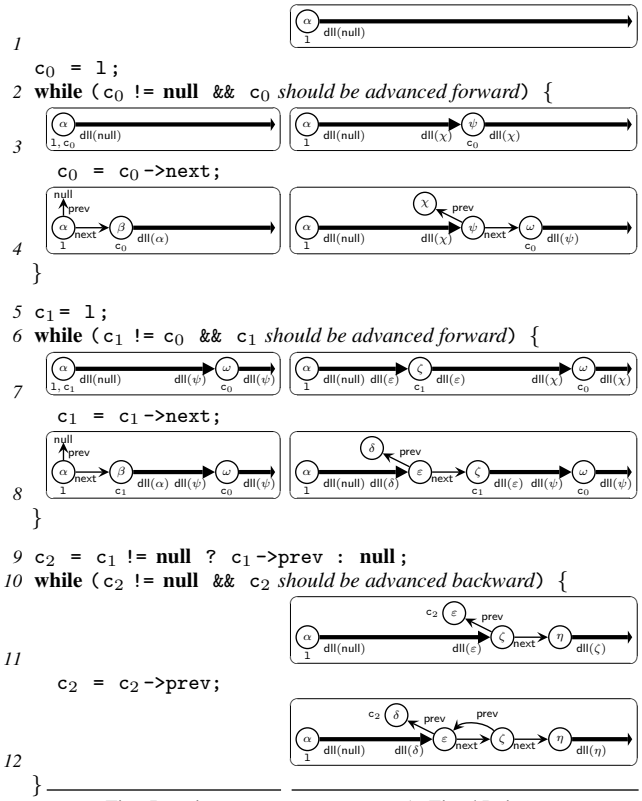


Figure 4. Unfolding of doubly-linked lists (checker `dll`).

to unfold it does not come directly from a user-supplied inductive definition.

Finally, from program point 11 to 12, we can only make progress by unfolding backward at the tail of a segment (the one from α to ζ) to materialize the edge for `c2->prev`. There are two distinct aspects to both forward and backward unfolding (though they are more evident in the backward case): (1) implementing the unfolding operations on the abstract domain, while justifying their soundness (described in Section 5.1.1) and (2) determining when to apply which unfolding operation (discussed in Section 5.1.2). Unlike forward unfolding, determining when and where to apply backward unfolding is not so obvious.

5.1.1 Unfolding Operations

We apply an unfolding in order to expose a heap cell (i.e., a points-to edge) before performing an operation on it. Relational checker definitions include not only shape information (i.e., how are the points-to edges arranged) but also data information in the form of pure constraints on the exposed heap cells. Thus, unfolding not only modifies the shape graph M but must also add additional constraints to the pure state P. In general, unfolding takes an element of the product domain $\langle M, P\rangle$ and yields a disjunction of abstractions $\langle M'_0, P'_0\rangle \vee \ldots \vee \langle M'_n, P'_n\rangle$. Though, it is often the case that all but one are ruled out by the data domain (i.e., we derive a contradiction in the pure constraints).

To mark the phases of unfolding, we distinguish an unfolding operation that works only on shape graphs from the overall unfolding operation. This distinction is also useful in describing other domain operations that rely on unfolding. We write $\mathbf{u}_\alpha$ for the forward unfolding at node α that takes a shape graph M and returns a set of graph and formula pairs where the first-order formulas give the pure constraints on the unfolded graphs. The overall unfolding operation $\mathbf{unfold}_\alpha$ then takes an element of the product domain $\langle M, P\rangle$ and returns a disjunction of such elements.

Unfolding Inductive Checkers. Let us consider the unfolding of a complete checker edge $\alpha.c(\delta)$, in the abstract element $\langle M * \alpha.c(\delta), P\rangle$. We describe unfolding of complete checker edges formally, as it serves as a basis for the unfolding of segments. The unfolding of a checker proceeds by unfolding each rule separately:

$$\mathbf{unfold}_\alpha(M * \alpha.c(\delta), P) \stackrel{\text{def}}{=} \bigvee_{R \in c} \mathbf{unfold}_\alpha(M * \alpha.R(\delta), P)$$

$$\mathbf{u}_\alpha(M * \alpha.c(\delta)) \stackrel{\text{def}}{=} \{\, \mathbf{u}_\alpha(M * \alpha.R(\delta)) \mid R \in c \,\}$$

where we write $R \in c$ for a rule R of checker definition c and overload checker application to also apply to rules. To unfold a rule, we first materialize the fields of the rule and then add the data constraint. We assume the base domain provides a $\mathbf{guard}_{\mathbb{P}^\sharp}(P, F)$ function that is a sound approximation of constraining P with F.

$$\mathbf{unfold}_\alpha(M * \alpha.R(\delta), P) \stackrel{\text{def}}{=} \langle M', \mathbf{guard}_{\mathbb{P}^\sharp}(P, F')\rangle \quad \text{where } (M', F') = \mathbf{u}_\alpha(M * \alpha.R(\delta))$$

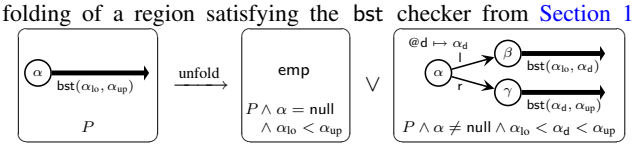
$$\mathbf{u}_\alpha(M * \alpha.R(\delta)) \stackrel{\text{def}}{=} (M * [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}](M^{\mathrm{u}} * M^{\mathrm{f}}), [\alpha, \delta, \vec{\varepsilon}/\pi, \rho, \vec{\kappa}]F)$$

In the shape graph, we simply unfold the rule and perform substitutions (where $\vec{\varepsilon}$ are fresh; and rule R has formals π and ρ, has free variables $\vec{\kappa}$, and has the form $\langle M^{\mathrm{u}} * M^{\mathrm{f}}, F\rangle$). This scheme can be performed in an automatic way and generates a finite number of disjuncts, which are well-formed elements of the domain. Furthermore, this algorithm is sound in that unfolding the checker edge results in a *weaker* disjunction.

Theorem 2 (Soundness of Unfolding)**.** *If unfolding transforms* $\langle M, P\rangle$ *into* $\vee_i \langle M'_i, P'_i\rangle$*, then* $\boldsymbol{\gamma}(\langle M, P\rangle) \subseteq \cup_i \boldsymbol{\gamma}(\langle M'_i, P'_i\rangle)$.

The proof is by an (easy) induction on the height of the checker "call tree".

Example 3 (Unfolding a Binary Search Tree)**.** We show the unfolding of a region satisfying the `bst` checker from Section 1.



Forward Unfolding of Inductive Segments. Because the semantics of a partial checker edge $\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')$ is defined by induction over the sequence of derivations (from α to α'), we can define an unfolding scheme analogous to the one for complete checker edges. We call this operation *forward unfolding* since it proceeds by unfolding checker definitions at the top of the derivation tree in the "standard" way (i.e., corresponding to the materialization of edges at the head α). This unfolding operation is exactly what is needed to materialize the edge for `c1->next` at program point 7 in Figure 4.

We extend the definition of forward unfolding ($\mathbf{unfold}_\alpha$) for segments. Like for complete checker edges, $\mathbf{unfold}_\alpha$ on partial checker edges generates a finite disjunction of $\langle M, P\rangle$ pairs. However, for partial checker edges, we must consider an additional case for the empty segment (i.e., the 0-step segment); only if the segment is non-empty (i.e., is of 1-or-more steps) do we get materializations corresponding to the rules of c.

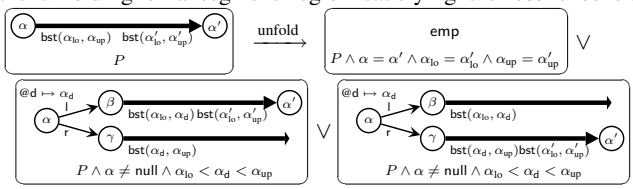
$$\mathbf{unfold}_\alpha(M * (\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')), P) \stackrel{\text{def}}{=} \mathbf{unfold}^0_\alpha(M * (\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')), P) \vee \left(\bigvee_{R \in c} \mathbf{unfold}_\alpha(M * (\alpha.R(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')), P)\right)$$

$$\mathbf{unfold}^0_\alpha(M * (\alpha.c(\delta) \mathrel{*\!\!=} \alpha'.c'(\delta')), P) \stackrel{\text{def}}{=} \begin{cases} (M, \mathbf{guard}_{\mathbb{P}^\sharp}(P, \alpha = \alpha' \wedge \delta = \delta')) & \text{if } c = c' \\ \bot & \text{if } c \neq c' \end{cases}$$

Observe that for the empty segment ($\mathbf{unfold}^0_\alpha$), we assert the additional equalities ($\alpha = \alpha'$ and $\delta = \delta'$) in the base domain. Only with these equalities can we determine in the analysis that the segment at program point 7 of the example is non-empty.

We omit the definition for $\mathbf{unfold}_\alpha$ on rules and the corresponding definitions of $\mathbf{u}_\alpha$ for the sake of brevity, as like for complete checker edges, they follow directly from the semantics given in Section 3.1. This extended unfolding function $\mathbf{unfold}_\alpha$ is sound in the same sense as the complete checker unfolding (i.e., it satisfies Theorem 2). The proof proceeds by induction over the length i of $*\!=^i$ derivations.

Example 4 (Unfolding a Binary Search Tree Segment)**.** We show the unfolding of a segment region satisfying the `bst` checker.



Note that using separating implication ($*\!-$) for partial checker edges would make them very difficult to unfold, as it would require involved restrictions to be made on checkers. Instead, our notion of segments as we define in this paper ($*\!=$) seems to be closer to our intuitive understanding of partial derivations of checkers and thus leads to a natural forward unfolding operation.

Backward Unfolding of Inductive Segments. The unfolding function defined above allows the analysis to materialize memory regions from the traversal argument of a checker edge. However, these unfolding operations do not apply to algorithms walking backward through invertible data structures, such as doubly-linked lists, as the sequence of edge dereferences does *not* follow the recursive checker calls that the forward unfolding would uncover. For example, this situation arises at program point 11 in Figure 4. From our intuitive understanding of `dll`, we know that if the segment between α and ζ is non-empty, then ε—the value of $\texttt{c}_2$—lies along that segment "just before ζ" (i.e., ε's `next` field points to ζ). Thus, if we are able to *unfold backward* along the segment from ζ, we could materialize the edge for $\texttt{c}_2\texttt{->prev}$.

The key observation we make to define the backward unfolding operation is that we can split segments into subsegments. For example, we can split a $*\!=^{i+1}$ segments into a pair of subsegments: $*\!=^i$ and $*\!=^1$. This segment splitting property is captured by the following lemma:

Lemma 1 (Splitting Inductive Segments)**.** *Let*

$$(\sigma, \nu) \in \boldsymbol{\gamma}(\alpha.c(\delta) \; *\!=^{i+1} \; \alpha'.c'(\delta'))$$

Then, there exists a checker c'' and nodes α'', δ'' such that

$$(\sigma, \nu) \in \boldsymbol{\gamma}(\alpha.c(\delta) \; *\!=^{i} \; \alpha''.c''(\delta'') \; * \; \alpha''.c''(\delta'') \; *\!=^{1} \; \alpha'.c'(\delta')).$$

The proof proceeds by induction on i. We remark that only checkers that may be called transitively from c need be considered for c'' and that the nodes α'', δ'' are fresh (modulo renaming).

Observe that Lemma 1 makes it possible to decompose a segment into a *finite* set of disjuncts with shorter segments. We can then define a backward unfolding operation by first splitting an $(i{+}1)$-step segment into an i-step segment and a 1-step segment and then apply forward unfolding to the 1-step segment (while separately considering the 0-step segment case).

More precisely, we define a *backward unfolding* function $\mathbf{unfold}^{\leftarrow 1}_{\alpha'}$, which should be applied at a node α' in an abstract state of the form $\langle M \; * \; (\alpha.c(\delta) \; *\!= \alpha'.c'(\delta')), P\rangle$ and conceptually unfolds the checker application just before α' in the sequence of calls from α to α'.

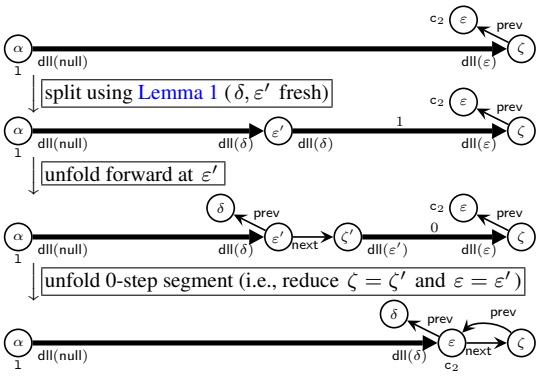


Figure 5. Backward unfolding of a doubly-linked list segment.

$\mathbf{unfold}^{\leftarrow 1}_{\alpha'}(M \; * \; (\alpha.c(\delta) \; *\!= \alpha'.c'(\delta')), P)$ returns a disjunction composed of the following:

1. the term $\mathbf{unfold}^0_\alpha(M \; * \; (\alpha.c(\delta) \; *\!= \alpha'.c'(\delta')), P)$, which corresponds to the empty segment;
2. the disjuncts that result from
$$\mathbf{unfold}_{\alpha''}(M * (\alpha.c(\delta) *\!= \alpha''.c''(\delta'')) * (\alpha''.c''(\delta'') *\!=^1 \alpha'.c'(\delta')), P)$$
for each possible c'' and with α'', δ'' fresh (as in Lemma 1), which corresponds to splitting the non-empty segment and applying the forward unfolding on the $*\!=^1$ edge.

This backward unfolding function is sound in the same sense as in Theorem 2. Note that Lemma 1 can be generalized to splitting segments of length $i + k$, which allows us to unfold backward k steps in one operation (for any constant k). We write $\mathbf{unfold}^{\leftarrow k}_{\alpha'}$ for the k-step backward unfolding function at α'.

In Figure 5, we show the individual steps in the backward unfolding of a doubly-linked list segment that is needed in the example shown in Figure 4. At the top, we show the subgraph of interest from program point 11. The empty segment case is ruled out because we have that $\varepsilon \neq \text{null}$ from the loop condition; since the parameter at α is `null`, an empty segment would imply that $\varepsilon = \text{null}$ (as ε is the parameter at ζ). Figure 5 shows the steps for the non-empty segment case. It is in the last step that we discover that $\zeta = \zeta'$ and $\varepsilon = \varepsilon'$, which allows us to find out that $\varepsilon\texttt{@next} \mapsto \zeta$ (i.e., is the points-to edge for $\texttt{c}_2\texttt{->next}$).

5.1.2 Expression Evaluation and Controlling Unfolding

The basic transfer functions for atomic operations (e.g., mutation, allocation, deallocation, and condition testing) are all fairly straightforward, as updates affect graphs locally. As described in Section 2, once points-to edges have been materialized, pointer updates amount to the swinging of an edge. Determining which edge to swing and to where is a simple walk of the graph from variables following the sequence of field dereferences of the command. This strong update is sound because each edge is a disjoint region of memory (i.e., the separation constraint).

For data operations (e.g., arithmetic expressions) on heap values, we symbolically evaluate the expressions by obtaining symbolic values for each memory access. We then create a new symbolic value to stand for this expression in the graph and assert this equality relation in the data domain $\mathbb{P}^\sharp$. For example, consider the following assignment statement showing an example transition:

$$\langle \texttt{x} \mapsto \alpha_{\texttt{x}}, \; \alpha_{\texttt{x}}\texttt{@data} \mapsto \beta, \; P\rangle$$

```
x->data = x->data + x->data;
```

$$\langle \texttt{x} \mapsto \alpha_{\texttt{x}}, \; \alpha_{\texttt{x}}\texttt{@data} \mapsto \gamma, \; \mathbf{guard}_{\mathbb{P}^\sharp}(P, \gamma = \beta + \beta)\rangle$$

where γ is a fresh symbolic value.

As described above, evaluating expressions requires following points-to edges in the shape graph. In case the relevant points-to edges are folded into complete or partial checker edges (i.e., summarized), we need to unfold the appropriate checker edge to materialize the desired points-to edge (using the operations defined in Section 5.1.1). To choose the appropriate edge and unfolding operation, we take advantage of the type inference on checker parameters defined in Section 4.

We are faced with deciding where to perform unfolding when the evaluation of an expression requires dereferencing a field f of a node α, but there is no such points-to edge from α. If there is a complete checker edge $\alpha.c(\delta)$ or a partial checker edge $\alpha.c(\delta) \ast\!\!= \alpha'.c'(\delta')$ starting from α (i.e., α is the traversal argument for some checker edge), then we may unfold this summary edge using the forward unfolding function $\mathbf{unfold}_\alpha$. If the points-to edge for $\alpha@f$ is materialized, then the evaluation of the expression can be resumed in the new unfolded graph(s). This materialization step is the basic one based on the knowledge that points-to edges emanate from the traversal parameter in inductive checker definitions and is what applies at program points 3 and 7 in Figure 4. Note that as an optimization, we need only consider outgoing checker edges where the type of the traversal parameter of the checker includes $f\langle\ell\rangle$ (for a level ℓ that is non-negative or unk).

Otherwise, if there is no outgoing checker edge from α, we look for a potential backward unfolding. We look elsewhere for a partial checker edge $\beta.c(\delta) \ast\!\!= \beta'.c'(\alpha)$ where α is a parameter at the tail. If additionally, the corresponding parameter of checker c' has a type that includes $f\langle n\rangle$ where $n < 0$, then we apply the backward unfold function at β' ($\mathbf{unfold}^{\leftarrow|n|}_{\beta'}$). The magnitude of the integer level tells us how many steps backward (i.e., how to split the segment from β to β'). In the doubly-linked list example at program point 11, we are trying to materialize $\varepsilon@\mathsf{prev}$ when ε has no outgoing edges. However, we have the edge $\alpha.\mathsf{dll}(\mathsf{null}) \ast\!\!= \zeta.\mathsf{dll}(\varepsilon)$ in the graph. Since the type of the additional parameter to dll contains $\mathsf{prev}\langle-1\rangle$ (see the type of ρ in Example 1), we know to unfold backward 1-step from ζ. Observe that the checker parameter typing does not affect soundness; it is utilized only as guide to decide where to unfold.

5.2 Folding with Relational Data Constraints

To obtain loop invariants in the shape domain, we need a way to identify subgraphs that should be *folded* into complete or partial checker edges. Because checker edges incorporate both shape and data properties, this summarization requires careful coordination between the shape domain and the data domain (in order to avoid losing precision unnecessarily). As observed in our prior work (Chang et al. 2007), the folding of the shape graph can be guided by consulting the iteration history through a widening operator defined on shape graphs. In this paper, we describe a widening algorithm that applies in the presence of relational checkers (i.e., with the introduction of inductive segments and rich data domains).

In this subsection, we define the *comparison* and *widening* operations, which both first perform a simultaneous traversal over the input shape graphs gathering constraints before then applying the corresponding operation in the base domain. We describe the comparison algorithm first, as it has similar but slightly simpler structure as compared to the widening and is also the key subroutine used by the widening.

5.2.1 Comparison of Abstract States

The comparison operator checks inclusion between two abstract elements in a conservative way. More precisely, it takes as input two abstract elements $A_\ell = \langle E_\ell, M_\ell, P_\ell\rangle$, $A_r = \langle E_r, M_r, P_r\rangle$ and returns **true** if it can establish that $\boldsymbol{\gamma}(A_\ell) \subseteq \boldsymbol{\gamma}(A_r)$ and **false** otherwise (which does not necessarily mean that the inclusion does not hold at the concrete level). Recall that the nodes in the shape graph correspond to existentially-quantified symbolic values, so at the basis of the comparison is a notion of node equivalence, which states that valuations should map nodes in A_ℓ and A_r to the *same* value for the inclusion to hold. For instance, if x is a variable, then the address of x should be the same on both sides, or the inclusion *cannot* hold. In fact, these equality relations constrain the valuations. Thus, when it succeeds, the comparison algorithm should return a *valuation transformer* Ψ that is a function mapping nodes of A_r into nodes of A_ℓ. The condition that Ψ is a function ensures that any aliasing expressed in A_r is also reflected in A_ℓ, so if at any point, this condition on Ψ is violated, then the comparison returns **false**.

At a high-level, the algorithm proceeds in three stages:

- First, the *initialization* of the algorithm creates an initial valuation transformer Ψ_{init} defined by the environments E_ℓ and E_r. Each variable should be mapped to the same address, so it is defined as follows: $\forall x \in \mathbf{Var}, \Psi_{\text{init}}(E_r(x)) = E_\ell(x)$.
- Second, a *comparison in the shape domain* is performed, which proceeds by checking inclusion locally. When new node relations are established as required for the inclusion to hold, the valuation transformer should be extended in order to include these constraints. Finally, it returns the following: (1) the final valuation transformer Ψ; (2) a first-order formula F, which collects pure constraints, which may arise during the computation that must ultimately be proven (i.e., are temporarily assumed).
- Last, a *comparison in the data domain* is performed that shows the inclusion of P_ℓ in P_r. We must also ask the data domain to prove and discharge the first-order side-conditions F computed in the previous step hold under the assumption of P_ℓ. All this is done modulo application of the valuation transformer Ψ.

Comparison in the Shape Domain. The basic idea of the graph comparison algorithm is to determine semantic inclusion by iteratively reducing to stronger statements until the inclusion is obvious. It does so using a set of rules that apply to the graph locally. While applying this set of rules, the algorithm carries along and enriches the pair (Ψ, F) introduced above.

The rules are presented in Figure 6. For conciseness, we omit the explicit bookkeeping of the node relations in the valuation transformer Ψ, that is, the rules assume the "final" Ψ is given and state the soundness of the whole computation. In practice, the state of Ψ also determines when a rule applies. We show this aspect indirectly by underlining the constraints on Ψ that are added once the rule applies, while the mappings that are not underlined must be in Ψ for the rule to apply. For instance, rule $\mathsf{c\text{-}pt}$ applies when α_ℓ and α_r match (i.e., when $\Psi(\alpha_r) = \alpha_\ell$) and when there is a field edge with label f from each node in both graphs. Then, the edges can be removed from both abstract elements (since $\alpha_\ell@f \mapsto \beta_\ell$ is obviously weaker than $\alpha_r@f \mapsto \beta_r$). A correspondence between β_ℓ and β_r can be added into Ψ, for these two nodes should correspond to the same value. When adding such a correspondence is not possible, because it would make Ψ not a function, the algorithm should return **false** (i.e., the inclusion cannot be established because this situation would mean that one value in M_r should be equal to two possibly distinct values in M_ℓ).

In the following, we briefly summarize the behavior of the other rules, whereas we show how the rules apply concretely in Example 5. Rule $\mathsf{c\text{-}emp}$ allows returning true when the proof is finished. Similar to $\mathsf{c\text{-}pt}$, rule $\mathsf{c\text{-}chk}$ matches two checker edges from related nodes. When there is a partial checker edge in M_ℓ and a checker edge in M_r, we split out the "prefix segment" in the right to match the left (rule $\mathsf{c\text{-}segchk}$ for a complete checker edge

$$\frac{}{\mathsf{emp} \sqsubseteq^{\mathsf{true}}_{\Psi} \mathsf{emp}}\ \textsf{c-emp} \qquad \frac{\Psi(\alpha_r) = \alpha_\ell \quad M_\ell \sqsubseteq^F_\Psi M_r \quad \underline{\Psi(\beta_r) = \beta_\ell}}{M_\ell * \alpha_\ell @ f \mapsto \beta_\ell \sqsubseteq^F_\Psi M_r * \alpha_r @ f \mapsto \beta_r}\ \textsf{c-pt} \qquad \frac{\Psi(\alpha_r) = \alpha_\ell \quad M_\ell \sqsubseteq^F_\Psi M_r \quad \underline{\Psi(\delta_r) = \delta_\ell}}{M_\ell * \alpha_\ell.c(\delta_\ell) \sqsubseteq^F_\Psi M_r * \alpha_r.c(\delta_r)}\ \textsf{c-chk}$$

$$\frac{\Psi(\alpha_r) = \alpha_\ell \quad M_\ell \sqsubseteq^F_\Psi M_r * \alpha'_r.c'(\delta'_r) \quad \underline{\Psi(\delta_r) = \delta_\ell} \quad \underline{\Psi(\alpha'_r) = \alpha'_\ell} \quad \underline{\Psi(\delta'_r) = \delta'_\ell} \quad (\alpha'_r, \delta'_r \text{ fresh})}{M_\ell * \alpha_\ell.c(\delta_\ell) \mathrel{-\!\!*} \alpha'_\ell.c'(\delta'_\ell) \sqsubseteq^F_\Psi M_r * \alpha_r.c(\delta_r)}\ \textsf{c-segchk}$$

$$\frac{\Psi(\alpha_r) = \alpha_\ell \quad M_\ell \sqsubseteq^F_\Psi M_r * \alpha'_r.c'(\delta'_r) \mathrel{-\!\!*} \alpha''_r.c''(\delta''_r) \quad \underline{\Psi(\delta_r) = \delta_\ell} \quad \underline{\Psi(\alpha'_r) = \alpha'_\ell} \quad \underline{\Psi(\delta'_r) = \delta'_\ell} \quad (\alpha'_r, \delta'_r \text{ fresh})}{M_\ell * \alpha_\ell.c(\delta_\ell) \mathrel{-\!\!*} \alpha'_\ell.c'(\delta'_\ell) \sqsubseteq^F_\Psi M_r * \alpha_r.c(\delta_r) \mathrel{-\!\!*} \alpha''_r.c''(\delta''_r)}\ \textsf{c-segseg}$$

$$\frac{M_\ell \sqsubseteq^F_\Psi M'_r \quad \text{where } (M'_r, F') \in \mathbf{u}_{\alpha_r}(M_r * \alpha_r.c(\delta_r))}{M_\ell \sqsubseteq^{F \wedge F'}_\Psi M_r * \alpha_r.c(\delta_r)}\ \textsf{c-uchk} \qquad \frac{M_\ell \sqsubseteq^F_\Psi M'_r \quad \text{where } (M'_r, F') \in \mathbf{u}_{\alpha_r}(M_r * \alpha_r.c(\delta_r) \mathrel{-\!\!*} \alpha'_r.c'(\delta'_r))}{M_\ell \sqsubseteq^{F \wedge F'}_\Psi M_r * \alpha_r.c(\delta_r) \mathrel{-\!\!*} \alpha'_r.c'(\delta'_r)}\ \textsf{c-useg}$$

Figure 6. The comparison operation in the shape domain.

in M_r and rule c-segseg for a partial checker edge). Rules c-uchk and c-useg unfold complete or partial checker edges in M_r when no other rule applies. Intuitively, when the comparison succeeds, Ψ gives us a relationship between the valuation of nodes on the left and on the right. In other words, a valuation ν_ℓ for A_ℓ can be composed with Ψ, to give a valuation for A_r. We write this composition as $\nu_\ell \circledcirc \Psi$, which we use to state soundness.

Theorem 3 (Soundness of Comparison in the Shape Domain)**.** *The above comparison function is sound: if* $M_\ell \sqsubseteq^F_\Psi M_r$, $(\sigma, \nu) \in \boldsymbol{\gamma}(M_\ell)$ *and* $\nu \circledcirc \Psi$ *satisfies* F*, then* $(\sigma, \nu \circledcirc \Psi) \in M_r$.

The proof proceeds by induction on the derivation of $\sqsubseteq^F_\Psi$. Soundness of rules c-pt and c-chk is straightforward. Proving rules c-segchk and c-segseg requires an induction over the length of the segments. Finally, the soundness of c-uchk and c-useg follows from the soundness of unfolding.

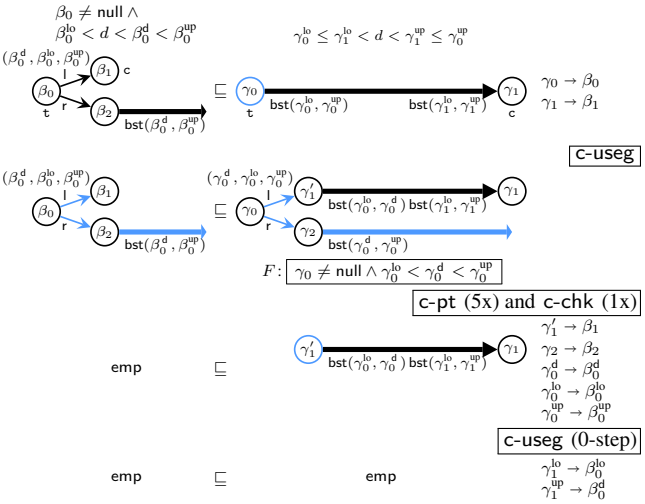
We remark that the rules differ significantly from the rules proposed in our prior work because of the introduction of inductive segments that we need for segment unfolding. In particular, the rules c-segseg (matching of partial checker edges) and c-useg (unfolding of partial checker edges) are new and replace the "assume" rule, which does not hold for inductive segments. Furthermore, the proofs for rules c-segchk and c-uchk are also quite different. The comparison algorithm in the shape domain is incomplete (i.e., the comparison may fail to prove the inclusion when it does hold at the concrete level). These rules have been primarily designed to be effective in the way the comparison is used in the join and widening algorithms where we need to see if M_ℓ is an unfolded version of M_r (see Section 5.2.2).

Comparison in the Combined Domain. If the comparison in the shape domain succeeds, then the comparison holds in the combined domain if we can discharge the side-conditions F and show the inclusion in the data domain. The key is that the comparison in the shape domain has computed Ψ, the correspondence between values in the left and values in the right, which captures the relationship between the shape and data domains. To define the overall comparison, we assume that the data domain has a function $\mathbf{prove}_{\mathbb{P}^\sharp}$ that takes as input an abstract element $P \in \mathbb{P}^\sharp$ and a first-order formula F and tries to prove that any valuation ν in $\boldsymbol{\gamma}_{\mathbb{P}^\sharp}(P)$ satisfies F, as well as a conservative comparison function $\sqsubseteq_{\mathbb{P}^\sharp}$. Furthermore, we assume the data domain can apply $\circledcirc$ at the abstract level, that is, $P \circledcirc \Psi$ applies the valuation transformer Ψ to rename symbolic values and capture any relations. Conceptually, this operation can be implemented by asserting equalities for each mapping in Ψ then projecting out the symbolic values in the range of Ψ. With that, the comparison function for the product domain is defined as follows:

$\langle M_\ell, P_\ell \rangle \sqsubseteq_\Psi \langle M_r, P_r \rangle$ iff there exists an F such that
$M_\ell \sqsubseteq^F_\Psi M_r$ and $\mathbf{prove}_{\mathbb{P}^\sharp}(P_\ell \circledcirc \Psi, F)$ and $P_\ell \circledcirc \Psi \sqsubseteq_{\mathbb{P}^\sharp} P_r$.

Moreover, $\langle E_\ell, M_\ell, P_\ell \rangle \sqsubseteq \langle E_r, M_r, P_r \rangle$ if and only if the above comparison evaluates successfully when started with $\Psi = \Psi_{\text{init}}$. The $\sqsubseteq_\Psi$ operator is sound, that is, if $\langle M_\ell, P_\ell \rangle \sqsubseteq_\Psi \langle M_r, P_r \rangle$, then $\boldsymbol{\gamma}(\langle M_\ell, P_\ell \rangle) \subseteq \boldsymbol{\gamma}(\langle M_r, P_r \rangle)$. Similarly, if $\langle E_\ell, M_\ell, P_\ell \rangle \sqsubseteq \langle E_r, M_r, P_r \rangle$, then $\boldsymbol{\gamma}(\langle E_\ell, M_\ell, P_\ell \rangle) \subseteq \boldsymbol{\gamma}(\langle E_r, M_r, P_r \rangle)$.

Example 5 (Verifying a Loop Invariant of Search Tree Traversal)**.** We highlight some aspects of the comparison algorithm by following an example derivation. This example checks, for a region, the inclusion of an iteration in a loop invariant for finding the value d in a binary search tree (essentially, the first stage prior to the insertion in the red-black tree example, as shown in Figure 1, lines 2–8).



The first line shows the initial goal: on the left-side of the comparison, we have the state where the cursor `c` has advanced to the left subtree of the root `t`. We want to show that this subgraph is contained in the segment from `t` to `c`. At the top, we show the pure constraints for each side: on the left, we have that $d < \beta_0^{\mathsf{d}}$, which is why `c` advanced to the left subtree. We want to show that d is in the range of the subtree from γ_1. In the right column, we show the valuation transformer Ψ as it is extended through the course of the computation. We show the rules used to transition between each step boxed and flush right. The highlighting of nodes and edges indicates where the rules apply. To keep the diagram compact, we write the values of the data fields as a tuple (e.g., $\beta_0^{\mathsf{d}}, \beta_0^{\mathsf{lo}}, \beta_0^{\mathsf{up}}$).

The first step applies c-useg that unfolds the segment on the right producing a proof obligation F (shown boxed). The next step matches points-to and complete checker edges, which extends Ψ. Finally, the last step unfolds the segment at γ'_1 as a 0-step segment, which produces key additional constraints on Ψ that come from the semantics of the 0-step segment.

To complete the proof, we need to discharge the above proof obligation and show inclusion at the data level. Applying the valuation transformer to the element of the data domain on the left side (i.e., $P_\ell \circledcirc \Psi$), we get the following:

$$\gamma_0 \neq \mathsf{null} \ \wedge \ \gamma_0^{\mathsf{lo}} = \gamma_1^{\mathsf{lo}} < d < \gamma_0^{\mathsf{d}} = \gamma_1^{\mathsf{up}} < \gamma_0^{\mathsf{up}}$$

which clearly implies the proof obligation and the inequality constraints on the right side (i.e., the loop invariant).

5.2.2 Join and Widening of Abstract States

The join and widening operators combine shape and data constraints to build an over-approximation of two abstract elements $A_\ell = \langle E_\ell, M_\ell, P_\ell \rangle$ and $A_r = \langle E_r, M_r, P_r \rangle$. Furthermore, the widening operation should ensure the termination of sequences of abstract iterates. In particular, termination should be achieved at both the shape and data levels. We first consider the join of abstract elements, that is, the computation of a sound approximation of both A_ℓ and A_r. Like for the comparison operator, we need to track the correspondence between symbolic values in the inputs and those in the output. Intuitively, a node α in the result should over-approximate the values corresponding to a pair of nodes (α_ℓ, α_r), where α_ℓ is in A_ℓ and α_r in A_r, so we maintain a pair of valuation transformers (Ψ_ℓ, Ψ_r) that describe these relations. For convenience, we also write $\Psi(\alpha)$ for $(\Psi_\ell(\alpha), \Psi_r(\alpha))$.

At a high-level, we can partition the join into stages in a similar manner as the comparison operation (by utilizing the valuation transformers Ψ_ℓ, Ψ_r):

- First, during *initialization*, for each variable x in the environment, a node α_x is created so as to represent the address of x. The valuation transformers Ψ_ℓ, Ψ_r are initialized so that $\forall x \in \mathbf{Var},\ \Psi(\alpha_x) = (E_\ell(x), E_r(x))$, and the resulting environment E is defined by $\forall x \in \mathbf{Var},\ E(x) = \alpha_x$.
- Second, a *join in the shape domain* builds a new shape abstraction M and returns it together with valuation transformers Ψ_ℓ, Ψ_r and residual first-order constraints F_ℓ, F_r that should be proven at the data level. Like in the comparison, it also enriches Ψ_ℓ and Ψ_r whenever a new node is created in order to preserve the consistency of the node pairing.
- Last, a *join in the data domain* is applied to $P_\ell \circledcirc \Psi_\ell$ and $P_r \circledcirc \Psi_r$. We must also ask the data domain to discharge the first-order constraints F_ℓ, F_r (so as to check that the abstractions performed in the shape join are valid with respect to data constraints).

Join in the Shape Domain. The join in the shape domain iteratively attempts to replace fragments in each of the input shape graphs (m_ℓ and m_r) with a new fragment (m) through a set of rewriting rules. A rule "consumes" fragments m_ℓ of M_ℓ and m_r of M_r and produces a fragment m for the result, which should be a sound approximation of m_ℓ and m_r modulo the application of Ψ_ℓ and Ψ_r, respectively.

In Figure 7, we present the fragment rewriting rules. We write $\langle m_\ell, m_r \rangle \rightsquigarrow^{F}_{\Psi} m$ for such a rewriting rule where Ψ is the valuation transformer and F is the residual first-order constraint from the rewriting. Like for the comparison, we do not explicitly show the extending of Ψ but rather assume the "final" Ψ is given. Also, the rules for the join are intended to be symmetric; for conciseness, we elide the left-sided version of the non-symmetric right-sided rules. Further discussion on how we decide in what order to apply the rules is found elsewhere (Chang et al. 2007). A key rule is j-waliases, which introduces a segment as a weakening for both sides. Note that the weakening on the left (from emp) is justified by one of basic properties of inductive segments (see Section 3.2).

Theorem 4 (Soundness of Join in the Shape Domain)**.** *If the join algorithm returns $M = M_\ell \sqcup M_r$ together with the valuation transformers Ψ_ℓ, Ψ_r and the first-order constraints F_ℓ, F_r, then for each side $i \in \{\ell, r\}$ and for all $(\sigma, \nu) \in \boldsymbol{\gamma}(M_i)$, if $\nu \circledcirc \Psi_i$ satisfies F_i, then $(\sigma, \nu \circledcirc \Psi_i) \in \boldsymbol{\gamma}(M)$.*

The proof proceeds by induction on the sequence of rewriting steps and case analysis on the rules used.

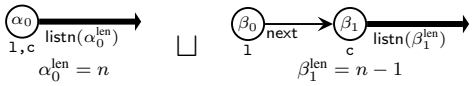
Join in the Combined Domain. The join operator for the combined domain $\langle E, M, P \rangle = \langle E_\ell, M_\ell, P_\ell \rangle \sqcup \langle E_r, M_r, P_r \rangle$ is defined as follows: (1) E is the environment computed from E_ℓ and E_r during initialization; (2) the shape join returns M together with Ψ_ℓ, Ψ_r, F_ℓ, and F_r when applied to M_ℓ and M_r; (3) the residual side-conditions are discharged, that is, $\mathbf{prove}_{\mathbb{P}^\sharp}(P_\ell \circledcirc \Psi_\ell, F_\ell)$ and $\mathbf{prove}_{\mathbb{P}^\sharp}(P_r \circledcirc \Psi_r, F_r)$ succeed; and (4) P is the join in the data domain (i.e., $P = (P_\ell \circledcirc \Psi_\ell) \sqcup_{\mathbb{P}^\sharp} (P_r \circledcirc \Psi_r)$). This join operator is sound, that is, $\boldsymbol{\gamma}(\langle E_\ell, M_\ell, P_\ell \rangle) \cup \boldsymbol{\gamma}(\langle E_r, M_r, P_r \rangle) \subseteq \boldsymbol{\gamma}(\langle E, M, P \rangle)$.

Widening. A widening operator $\triangledown$ is a join operator with a stabilizing property so as to ensure termination of the analysis (Cousot and Cousot 1977). This operator should ensure that both shapes and data invariants are stable after finitely many iterations. The shape join already has the stabilizing property, so a widening operation for the combined domain can be obtained by simply using the widening operator $\triangledown_{\mathbb{P}^\sharp}$ instead of the join $\sqcup_{\mathbb{P}^\sharp}$ in the data domain.

Theorem 5 (Widening Termination)**.** *Given any sequence $(A'_n)_{n\in\mathbb{N}}$, the sequence $(A_n)_{n\in\mathbb{N}}$ (where $A_n = (E_n, M_n, P_n)$) defined by $A_0 = A'_0$ and $A_{n+1} = A_n \triangledown A'_{n+1}$ is ultimately stationary.*

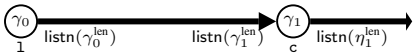
The proof is based arguments similar to those required to prove the termination of widening in cofibered domains (Venet 1996): the shape graphs stabilize first, and then the data abstract values eventually converge since a widening operator is used.

Example 6 (Traversal of a List of Given Length)**.** In this example, we consider the join of the first two iterates that arise during the traversal of a list of length n:



$\alpha_0^{\mathsf{len}} = n$ $\qquad$ $\beta_1^{\mathsf{len}} = n - 1$

The above join algorithm produces the following shape invariant M after applying rules j-chk and j-waliases:



Rule j-chk extends the valuation transformer so that $\Psi(\eta_1^{\mathsf{len}}) = (\alpha_0^{\mathsf{len}}, \beta_1^{\mathsf{len}})$. From rule j-waliases, we get on the left side that $\gamma_0^{\mathsf{len}} = \gamma_1^{\mathsf{len}}$ (i.e., $\Psi_\ell(\gamma_0^{\mathsf{len}}) = \Psi_\ell(\gamma_1^{\mathsf{len}})$). On the right side, one unfolding step is required in the comparison (to fold from β_0 to β_1 into a listn segment), so by the definition of the additional parameter of the recursive call in the definition of checker listn, we have the relation that $\gamma_0^{\mathsf{len}} = \gamma_1^{\mathsf{len}} + 1$. This relation cannot be tracked by Ψ_r as we have defined it in this paper, but we can consider an *extended valuation transformer* that not only maps nodes of the result into nodes of one of the inputs, but also allows expressing such relations among the nodes of the output graph. Such relations typically arise in the unfoldings performed during the comparisons required for applying rules j-waliases, j-wchk, and j-wseg. After the join of the shape abstractions, the above relations are propagated to P_ℓ and P_r (i.e., by applying $\circledcirc$). This results in the following for the join in the data domain $\mathbb{P}^\sharp$:

$$\left[\eta_1^{\mathsf{len}} = n \ \wedge \ \gamma_0^{\mathsf{len}} = \gamma_1^{\mathsf{len}}\right] \ \sqcup_{\mathbb{P}^\sharp} \ \left[\eta_1^{\mathsf{len}} = n - 1 \ \wedge \ \gamma_0^{\mathsf{len}} = \gamma_1^{\mathsf{len}} + 1\right]$$

If we let $\mathbb{P}^\sharp$ be the domain of linear equalities (Karr 1976), the result of the join is $[\gamma_0^{\mathsf{len}} - \gamma_1^{\mathsf{len}} = n - \eta_1^{\mathsf{len}}]$, which says that the sum of the lengths corresponding to the partial and the complete checker segments is n (i.e., the length of the list does not change

$$\frac{\Psi(\alpha)=(\alpha_\ell,\alpha_r)\quad \underline{\Psi(\beta)=(\beta_\ell,\beta_r)}}{\langle \alpha_\ell @ f\mapsto\beta_\ell, \alpha_r @ f\mapsto\beta_r\rangle \rightsquigarrow_\Psi \alpha @ f\mapsto\beta}\ \text{j-pt}$$

$$\frac{\Psi(\alpha)=(\alpha_\ell,\alpha_r)\quad \underline{\Psi(\delta)=(\delta_\ell,\delta_r)}}{\langle \alpha_\ell.c(\delta_\ell), \alpha_r.c(\delta_r)\rangle \rightsquigarrow_\Psi \alpha.c(\delta)}\ \text{j-chk}$$

$$\frac{\Psi(\alpha)=(\alpha_\ell,\alpha_r)\quad m_r \sqsubseteq^F_{\Psi_r} \alpha.c(\delta)\quad \underline{\Psi_\ell(\delta)=\delta_\ell}}{\langle \alpha_\ell.c(\delta_\ell), m_r\rangle \rightsquigarrow^F_\Psi \alpha.c(\delta)}\ \text{j-wchk}$$

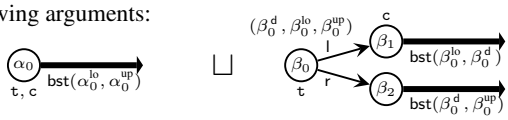
$$\frac{\Psi(\alpha)=(\alpha_\ell,\alpha_r)\quad \Psi(\alpha')=(\alpha'_\ell,\alpha'_r)\quad m_r \sqsubseteq^F_{\Psi_r} \alpha.c(\delta) \mathrel{-\!\!*} \alpha'.c'(\delta')\quad \underline{\Psi_\ell(\delta)=\delta_\ell}\quad \underline{\Psi_\ell(\delta')=\delta'_\ell}}{\langle \alpha_\ell.c(\delta_\ell) \mathrel{-\!\!*} \alpha'_\ell.c'(\delta'_\ell), m_r\rangle \rightsquigarrow \alpha.c(\delta) \mathrel{-\!\!*} \alpha'.c'(\delta')}\ \text{j-wseg}$$

$$\frac{\Psi(\alpha)=(\alpha_\ell,\alpha_r)\quad \Psi(\alpha')=(\alpha_\ell,\alpha'_r)\quad m_r \sqsubseteq^F_{\Psi_r} \alpha.c(\delta) \mathrel{-\!\!*} \alpha'.c(\delta')\quad \underline{\Psi_\ell(\delta)=\delta_\ell}\quad \underline{\Psi_\ell(\delta')=\delta_\ell}}{\langle \mathsf{emp}, m_r\rangle \rightsquigarrow^F_\Psi \alpha.c(\delta) \mathrel{-\!\!*} \alpha'.c(\delta')}\ \text{j-waliases}$$

Figure 7. Fragment rewriting rules for the join operation in the shape domain.

during the traversal). This invariant is the most precise one can hope for on this example.

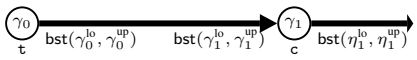
Example 7 (Inferring a Loop Invariant for Search Tree Traversal)**.** Consider again the code for finding a value d in a binary search tree (i.e., lines 2–8 in Figure 1). For this example, we assume $\mathbb{P}^\sharp$ is an abstract domain that supports inequalities among pairs of variables (e.g., octagons (Miné 2006)). Suppose in the first iteration, the cursor `c` is advanced to the left subtree (i.e., d is smaller than the data at the root), then the the first widening is applied to the following arguments:



$(*)$

$-\infty = \alpha_0^{\mathsf{lo}} < d < \alpha_0^{\mathsf{up}} = \infty \qquad -\infty = \beta_0^{\mathsf{lo}} < d < \beta_0^{\mathsf{d}} < \beta_0^{\mathsf{up}} = \infty$

The join in the shape domain yields the following shape graph:



$(\dagger)$

The valuation transformer Ψ is initialized to $\Psi(\gamma_0) = (\alpha_0, \beta_0)$, $\Psi(\gamma_1) = (\alpha_0, \beta_1)$ from the environment. Then, rule j-chk applies to add the complete checker edge from γ_1, which also extends Ψ so that $\Psi(\eta_1^{\mathsf{lo}}) = (\alpha_0^{\mathsf{lo}}, \beta_0^{\mathsf{lo}})$ and $\Psi(\eta_1^{\mathsf{up}}) = (\alpha_0^{\mathsf{up}}, \beta_0^{\mathsf{d}})$. Finally, rule j-waliases applies to create the segment, which enriches Ψ so that the following relations hold: $\Psi_\ell(\gamma_0^{\mathsf{lo}}) = \Psi_\ell(\gamma_1^{\mathsf{lo}})$, $\Psi_\ell(\gamma_0^{\mathsf{up}}) = \Psi_\ell(\gamma_1^{\mathsf{up}})$ for the initial state and $\Psi_r(\gamma_0^{\mathsf{lo}}) = \Psi_r(\gamma_1^{\mathsf{lo}}) = \beta_0^{\mathsf{lo}}$, $\Psi_r(\gamma_0^{\mathsf{up}}) = \beta_0^{\mathsf{up}}$, $\Psi_r(\gamma_1^{\mathsf{up}}) = \beta_0^{\mathsf{d}}$ for the first iterate. Note that the inclusion check that we need to weaken the subgraph from β_0 to β_1 to a partial checker edge is the comparison in the shape domain shown in Example 5. The extensions to Ψ_r can be read from there.

Then, applying the valuation transformers Ψ_ℓ, Ψ_r to the respective input elements, the data invariants to join are as follows:

$$\left[\gamma_0^{\mathsf{lo}} = \gamma_1^{\mathsf{lo}} \wedge \gamma_0^{\mathsf{up}} = \gamma_1^{\mathsf{up}} \wedge -\infty = \eta_1^{\mathsf{lo}} < d < \eta_1^{\mathsf{up}} = \infty\right]$$
$$\sqcup_{\mathbb{P}^\sharp} \left[-\infty = \gamma_0^{\mathsf{lo}} = \gamma_1^{\mathsf{lo}} = \eta_1^{\mathsf{lo}} < d < \gamma_1^{\mathsf{up}} = \eta_1^{\mathsf{up}} < \gamma_0^{\mathsf{up}} = \infty\right]$$

However, the join of these two data invariants is problematic because the first invariant is, in a sense, too general, for any $(\gamma_0^{\mathsf{lo}}, \gamma_0^{\mathsf{up}}) = (\gamma_1^{\mathsf{lo}}, \gamma_1^{\mathsf{up}})$ approximates a 0-step segment. Specifically, the equality constraint $(\gamma_0^{\mathsf{lo}}, \gamma_0^{\mathsf{up}}) = (\eta_0^{\mathsf{lo}}, \eta_0^{\mathsf{up}})$ is not required in the initial state (i.e., the left data element) but is required in the first iterate (i.e., the right data element). Moreover, the segment between γ_0 and γ_1 is only an approximation of the corresponding subgraph on the right when $(\gamma_0^{\mathsf{lo}}, \gamma_0^{\mathsf{up}}) = (-\infty, \infty)$.

This example illustrates one of the difficulties that we sketched in Section 2. As the symbolic values form the coordinates of the base data domain, it is quite sensitive to large changes in the shape graph. Here, we see that between the graph at the initial state and the join, α_0 has been "split" into γ_0 and γ_1 and $(\gamma_0^{\mathsf{lo}}, \gamma_0^{\mathsf{up}})$ has been "split" into $(\gamma_0^{\mathsf{lo}}, \gamma_0^{\mathsf{up}})$ and $(\gamma_1^{\mathsf{lo}}, \gamma_1^{\mathsf{up}})$. However, we observe that this becomes a non-issue once the graph stabilizes. Therefore, we propose to delay the join of the data invariants until the next iteration, which is a common static analysis technique.

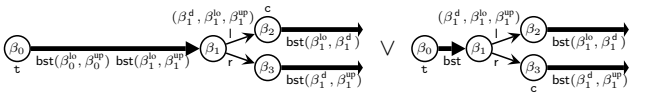
Now, for the case where the cursor is advanced to the right subtree in the first iteration, the result of the widening yields the same shape graph shown above and marked as $(\dagger)$. The data constraints are, however, as follows:

$$-\infty = \gamma_0^{\mathsf{lo}} < \gamma_1^{\mathsf{lo}} = \eta_1^{\mathsf{lo}} < d < \gamma_1^{\mathsf{up}} = \eta_1^{\mathsf{up}} = \gamma_0^{\mathsf{up}} = \infty$$

The join of the numerical invariants corresponding to the left and right branches after one iteration is as follows:

$$-\infty = \gamma_0^{\mathsf{lo}} \leq \gamma_1^{\mathsf{lo}} = \eta_1^{\mathsf{lo}} < d < \gamma_1^{\mathsf{up}} = \eta_1^{\mathsf{up}} \leq \gamma_0^{\mathsf{up}} = \infty \qquad (\ddagger)$$

After the next iteration, we get the following two shape graphs:



The computation of the join of each of these invariants with the result of the first widening output $(\dagger)$ reveals that the latter is stable at the shape level. Furthermore, from this point, the data invariant marked as $(\ddagger)$ above is also stable, so we have obtained a fixed point. The loop invariant says that at any step of the find, the cursor `c` points to a subtree of `t` where the range of the data values in the subtree contains d.

Example 7 also shows the other difficulty alluded to in Section 2, which was solved by a different technique. In the above, the range of the subtree is not only expressed in the checker parameters of a folded region but also as fields of unfolded nodes. Without these fields, the resulting situation of the first widening (shown in display $(*)$) is similar to what is described above without the delayed join of data invariants. Specifically, we would get a "too general" instantiation of the partial checker edge where the lower bound at the head (γ_0^{lo}) could be any value smaller than the key at the root. The valuation transformer Ψ_r is never constrained so that $\Psi_r(\gamma_0^{\mathsf{lo}}) = \Psi_r(\gamma_1^{\mathsf{lo}}) = \beta_0^{\mathsf{lo}}$. This folding would be sound, but it would not allow folding at the next step, due to being too general. Instead, with these fields, we break the dependence on synthesizing the appropriate "less general" parameters. Example 6 does not require such fields because of the tight constraints on the parameters. We note that this kind of technique is also rather common in verification (e.g., McPeak and Necula (2005)), which we apply here to separate the analysis concerns from the modeling ones.

6. Experimental Evaluation

We have applied a prototype implementation of our shape analysis for C code to a set of data structure manipulation benchmarks. Table 1 presents analysis statistics executed on a 2.0GHz Intel Xeon with 2GB of RAM. In each case, we verified that the pointer manipulation preserved the structural invariants of the data structures (e.g., back-pointer property, acyclicity, non-sharing, treeness). We did not verify any numerical properties on the node data, as we do not yet have an effective interface to implementations of numerical base domains. In the table, when the operation exists for the non-back pointer analogue (i.e., singly-linked list vs. doubly-linked list and tree vs. tree with parent pointers), we show the analysis time

Benchmark	With Back Pointers			Without
	Time (sec)	Disj. (num)	Iter. (num)	Time (sec)
list reverse	0.0014	1	3	0.0006
list copy	0.0053	2	3	0.0037
list insert	0.0038	2	4	0.0049
list insert*	0.0042	2	4	-
list remove*	0.0065	5	4	-
list remove and back	0.0068	5	4	-
search tree insert	0.0083	5	5	0.0148
search tree insert and back	0.0470	5	5	-

Table 1. Benchmark results for verifying shape preservation. We show the analysis time, the maximum number of disjuncts at any program point (Disj.), and the maximum number of iterations at any point (Iter.). Where applicable, we also show the analysis time for the analogous operation in the structure without back pointers.

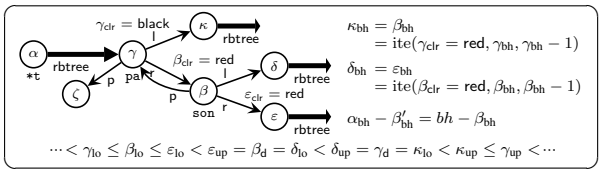
of the non-back pointer variant as a point of comparison. The insert and remove cases marked with * are variants where the search for the location to do the operation is done with only one cursor, so back pointers are required to perform the operation. The "list remove and back" example finds an element if it exists, removes it, and walks back modifying the previous nodes (e.g., updating a length field); the "tree insert and back" is similar. In all the test cases, the analysis times are negligible, but more importantly, the maximum number of disjuncts (i.e., the number of shape graphs), we need to keep at any program point seems to be small.

7. Example: Red-Black Trees

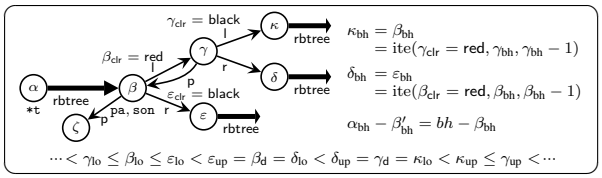
We now return to the red-black tree insertion example from Figure 1, Section 2 to discuss how the invariants in the rebalancing loop after an insertion can be obtained. In Figure 8, we present one of the rebalancing cases in detail with the fixed-point invariants shown at key points. At program point 21, we show a loop invariant for the rebalancing loop that is sufficient to show that after the loop, α (pointed to by `*t`) is a red-black tree according to checker `rbtree`. The shape portion of the loop invariant indicates that δ and ε are red-black trees (with certain parameters) but perhaps not locally around β. In the data portion, we have the ordering property on the data (shown at the bottom), which is obtained in the search loop (lines 2 to 8) prior to the insertion. Note that this ordering invariant is obtained by the analysis algorithm as described in the example widening on a binary search tree traversal (Example 7 of Section 5.2.2) and then preserved in this loop. The other data constraints describe the invariant on the black height parameters of the checkers (e.g., β_{bh}). The top constraint gives the relation between the black height at β with those at δ and ε, which first comes from the unfolding of the `rbtree` in the search loop and then notably preserved on insertion (line 9). The middle constraint is the relation between the black height at α with the subtree at β (where bh is the initial black height of the entire tree and β'_{bh} is the black height checker parameter at the end of segment to β—as opposed to β_{bh} that is the black height parameter from β). This invariant is also obtained in the search loop and in the same manner as the example widening on lists of given length (Example 6). However, the base domain should be richer to handle the additional boolean structure (on whether a node is red or black) by using, for example, binary decision diagrams (BDDs) with linear equalities at the leaves.

Observe that we have no constraints on the red-ok parameters (e.g., $\varepsilon_{\text{redok}}$) meaning that any of δ, ε, and β may be red and thus locally violating the color aspect of the red-black tree invariant for the entire structure. The shown rebalancing case addresses

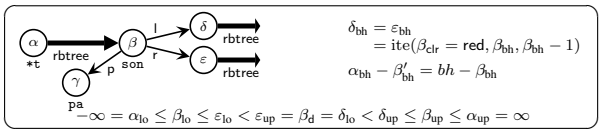
```
10 while (pa != null) {
11   if (pa->r && pa->r->r
         && pa->r->clr == RED && pa->r->r->clr == RED) {
12     son = pa->r;
```




```
13     son->r->clr = BLACK;
14     son->p = pa->p; set pa->p's l/r field to replace pa with son;
15     pa->r = son->l; if (son->l) { son->l->p = pa; }
16     son->l = pa;    pa->p = son;
17     pa = son;
```




```
18   }
19   ... other rebalancing cases ...
20   son = pa; pa = pa->p;
```




```
21 }
```

Figure 8. Rebalancing in the red-black tree insertion example.

this violation by performing a left rotation and coloring. From program point 21 to 13, the analysis does a backward unfolding to materialize the fields of `pa`. This unfolding (along with $\alpha_{\text{bh}} - \beta'_{\text{bh}} = bh - \beta_{\text{bh}}$) yields the additional black height constraint on κ_{bh} and β_{bh}. Then the condition that $\beta_{\text{clr}} = \text{red}$ (and an unfolding constraint on $\varepsilon_{\text{redok}}$) tells us that $\gamma_{\text{clr}} = \text{black}$. For compactness, we do not show the unfolding of ε, which is needed only to access its color field. Aside from the coloring of ε, the rotation only affects the graph (as shown at point 18). Now, compare this after-rotation state with the loop invariant at 21. We see that the after-rotation state is contained in the loop invariant (after advancing the cursor `pa`) by folding the region from γ into a `rbtree`, which is computed by the join as described in Section 5.2.2. In the data constraints, the key observation is that the coloring gives us that $\kappa_{\text{bh}} = \delta_{\text{bh}} = \varepsilon_{\text{bh}}$. Also, while the new black height at β and ε increase by one, this is summarized by the difference equality constraint.

8. Related Work

In the past few years, we note a growing interest in shape analyses based on inductive definitions in separation logic. Distefano et al. (2006) build into the analysis a singly-linked list segment predicate with specialized folding rules (in a unary canonicalization operation). Berdine et al. (2007) have extended this framework to apply to doubly-linked lists polymorphically. In contrast, our analysis algorithm is parameterized by inductive checker definitions that support data constraints and folds using a generic widening operator (i.e., uses iteration history). Magill et al. (2007) propose an analysis for length-specified lists (like `listn`) that is staged as opposed to a reduced product: the shape analysis is performed prior to applying the numerical analysis. This design has clear engineering advantages but may make it harder to achieve the desired precision.

Also, like the works mentioned previously, the length-specified list predicate is built into the analysis, which enables the domain operations to be tuned for the relational aspects. Guo et al. (2007) describe a global shape analysis that synthesizes inductive structural invariants (i.e., shape only) from construction patterns present in the code. In contrast, our approach is to focus the shape analysis based on developer intent, which often includes intertwined data constraints. They also describe a notion similar to segments ("truncation points"), though it is unclear how unfolding is applied.

TVLA (Sagiv et al. 2002) is a well-known, very powerful and generic framework based on three-valued logic for setting up shape analyses. It has been widely used to verify complex structures and has proven able to tackle deep properties, such as sortedness (Lev-Ami et al. 2000). A large amount of ongoing work on this topic addresses scalability (e.g., Arnold (2006)). Our parametric framework seems to offer an interesting balance between expressivity and efficiency (Chang et al. 2007); the present contribution significantly extends it to accommodate data invariants and relational shapes.

The abstract interpretation-based analysis proposed by Gulwani and Tiwari (2007) is based on an encoding of shape invariants into quite expressive $\exists\forall$ quantified formulas that essentially correspond to our checker edges. The existential defines a segment endpoint (i.e., a bound on the recursion depth), while the universal gives the induction. Also, it does not make explicit use of separation and thus requires may/must alias information to be recomputed on the fly.

Shape analysis-based on reference counting and region inference techniques has proven quite successful in addressing certain relational properties, such as the balance invariant in AVL trees (Rugina 2004) and doubly-linked lists (Cherem and Rugina 2007) at a very reasonable cost. However, it is not clear how to extend the analysis to more global properties, such as the search tree ordering invariant, amenable to the shape analyses mentioned previously. There is also a large body of work on using "verification procedures" for shape properties (e.g., Nguyen et al. (2007); Chatterjee et al. (2007); McPeak and Necula (2005); Møller and Schwartzbach (2001)). These techniques typically address expressive sets of predicates but require the user to supply loop invariants. Another relevant line of work addresses array properties. Whereas no folding/unfolding of specialized structures need to be considered, partitioning and summarization of array cells are required. In particular, Gopan et al. (2005) utilize canonical abstraction and numerical abstract domains to achieve this, while Cousot (2003) uses parametric predicate abstraction.

9. Conclusion

We have described a generic framework for relational inductive shape analysis with user-supplied invariant checkers. We have highlighted the difficulties with relational checkers, which include invertible structural invariants, such as in doubly-linked lists, as well as intertwined data and shape invariants, such as in binary search trees. The key mechanisms we have introduced to enable relational shape analysis are the notion of (generic) inductive segments and a two-phased widening operator over the combination of shape and data domains. Moreover, we have introduced a typing of checker parameters with an inference algorithm that is then utilized to control the firing of unfoldings in the abstract interpretation. Finally, we have shown the applicability of our analysis algorithm to the correctness verification of insertion for red-black trees (including both the ordering and balance invariants).

Acknowledgments

We thank George Necula for valuable discussions and his support of this project, as well as the anonymous referees for providing helpful comments on drafts of this paper. This research was supported in part by NSF under grants CCR-0326577, CCF-0524784, and CNS-0509544; and an NSF Graduate Research Fellowship.

References

Gilad Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *Static Analysis (SAS)*, 2006.

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

Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Static Analysis (SAS)*, 2007.

Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

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

Patrick Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*, 2003.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, 1977.

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

Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Principles of Programming Languages (POPL)*, 2005.

Sumit Gulwani and Ashish Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *Computer-Aided Verification (CAV)*, 2007.

Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *Programming Language Design and Implementation (PLDI)*, 2007.

Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6, 1976.

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

Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Software Testing and Analysis (ISSTA)*, 2000.

Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for separation logic based shape analyses. In *Static Analysis (SAS)*, 2007.

Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *Computer-Aided Verification (CAV)*, 2005.

Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI)*, 2001.

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002.

Radu Rugina. Quantitative shape analysis. In *Static Analysis (SAS)*, 2004.

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

Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Static Analysis (SAS)*, 1996.