Shape Analysis for Unstructured Sharing *

Huisong Li[1], Xavier Rival[1], and Bor-Yuh Evan Chang[2]

[1] INRIA, ENS, CNRS, PSL*, Paris, France
[2] University of Colorado, Boulder, Colorado, USA

Abstract. Shape analysis aims to infer precise structural properties of imperative memory states and has been applied heavily to verify safety properties on imperative code over pointer-based data structures. Recent advances in shape analysis based on separation logic has leveraged summarization predicates that describe unbounded heap regions like lists or trees using inductive definitions. Unfortunately, data structures with *unstructured sharing*, such as graphs, are challenging to describe and reason about in such frameworks. In particular, when the sharing is unstructured, it cannot be described inductively in a local manner. In this paper, we propose a global abstraction of sharing based on set-valued variables that when integrated with inductive definitions enables the specification and shape analysis of structures with unstructured sharing.

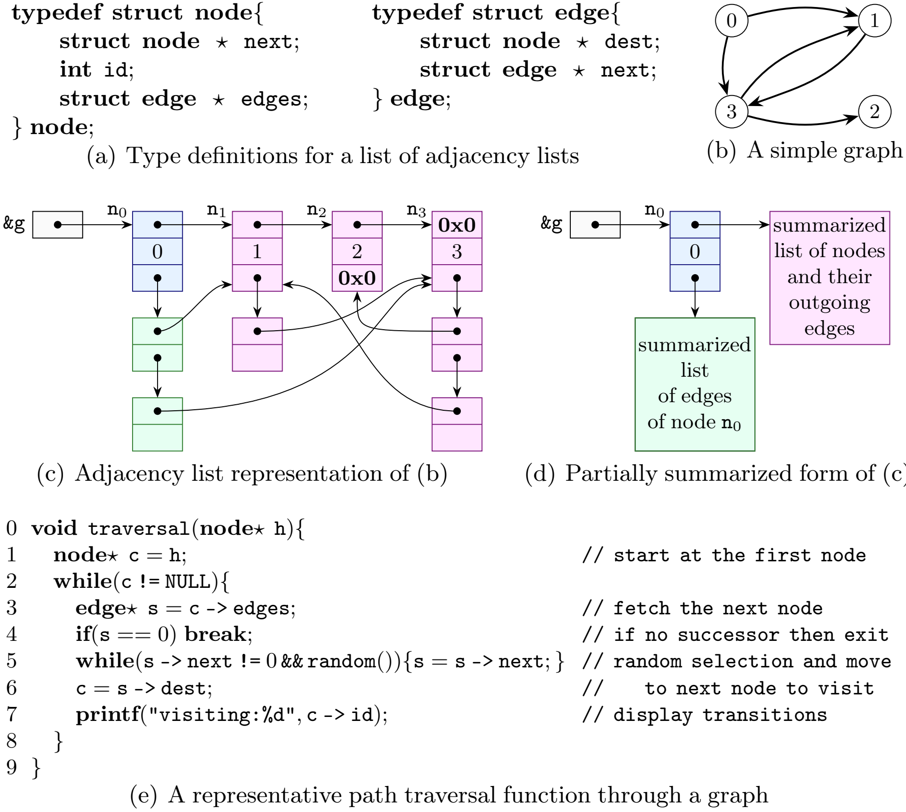
1 Introduction

Many recent advances in shape analysis have been made by building on separation logic [24] with inductive definitions. Such frameworks (e.g., [14,1,7]) leverage (1) separating conjunction $*$ to enable local reasoning and strong updates by combining properties holding over disjoint memory regions and (2) inductive definitions to summarize recursive structures of unbounded size. While this approach has been effective for many applications, a significant limitation has been its inability to effectively handle *unstructured sharing*.

We say that a data structure has *sharing* whenever a given cell in the data structure may be pointed to by several other cells. Singly-linked lists and trees are *unshared data structures*, while other important structures, such as directed-acyclic graphs (DAGs) and graphs in general, are *shared data structures*. Certain shared data structures have regular sharing patterns that can be described using a bounded number of constraints on each cell. For example, doubly-linked lists can be summarized using the following inductive definition:

$$\alpha \cdot \mathbf{dll}(\delta) ::= (\mathbf{emp} \wedge \alpha = 0) \vee (\alpha.\mathtt{prev} \mapsto \delta * \alpha.\mathtt{next} \mapsto \beta * \beta \cdot \mathbf{dll}(\alpha) \wedge \alpha \neq 0)$$

* The research leading to these results has received funding from the European Research Council under the European Union's seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD, and from the ARTEMIS Joint Undertaking no 269335 (see Article II.9 of the JU Grant Agreement) and the United States National Science Foundation under grant CCF-1055066.



(a) Type definitions for a list of adjacency lists

(b) A simple graph

(c) Adjacency list representation of (b)

(d) Partially summarized form of (c)

(e) A representative path traversal function through a graph

Fig. 1. Graph represented by adjacency lists and traversal function.

This definition states that α is a doubly-linked list pointer if and only if it is either null (0) or a pointer to a list element; in the latter case, the `prev` field of α points to δ, and the tail β should be a doubly-linked list such that the `prev` field of its first element should point back to α itself. We say that doubly-linked lists have *structured sharing*—sharing occurs because each cell points back to its predecessor, but since each cell has *exactly one* predecessor (except for the first cell), it can be specified by the parameter δ. A skip list is another such example [7].

On the other hand, the case of structures with unstructured sharing, such as general graphs, is much more challenging since the number of predecessors of a node is unbounded and since the predecessors could be anywhere in the structure. To make the challenges more concrete, consider the representation of graphs shown in Figure 1. Figure 1(a) shows a type definition for representing graphs as *adjacency lists*: a graph is a list of nodes, each node has a list of edges, and an edge is a pointer to its destination node. Figure 1(c) shows a representation of the graph of Figure 1(b), where node i is described at address $\mathtt{n}_i$. To extend shape analysis techniques to this structure (and prove memory safety or functional properties of algorithms manipulating it), we need an effective method to summarize

instances of this structure and to manipulate such summaries. A natural approach to summarization is to exploit the list-of-lists inductive skeleton of adjacency lists, as hinted at in Figure 1(d): at node $\mathtt{n}_0$, the list of its adjacent nodes is inductively summarized in the green region, while the adjacency list of the other nodes in the graph are summarized in the purple region. What is implicit in this informal diagram is that these summarized regions must, for precision, capture complex, unstructured, cross pointer relations (i.e., the curving lines in Figure 1(c)).

To capture this unstructured sharing precisely, we observe that the correctness of the structure stems from the fact that each edge pointer points to an address in $\mathscr{E} = \{\mathtt{n}_0, \mathtt{n}_1, \mathtt{n}_2, \mathtt{n}_3\}$. Thus, the absence of dangling edges can be captured by adjoining a *set property* to a conventional list predicate. We need to capture that all edge pointers point to nodes belonging to the set $\mathscr{E}$ of valid nodes in the graph for each node's adjacency list. To give an inductive definition for the outer list of nodes, we need to ensure that this list of nodes is consistent with the set $\mathscr{E}$ of valid nodes, and thus we require a second set variable $\mathscr{F}$ that captures the nodes summarized in this list region. For example, in the node list summary of Figure 1(d) (shown in purple), this variable $\mathscr{F}$ should be the set $\{\mathtt{n}_1, \mathtt{n}_2, \mathtt{n}_3\}$.

While we have hinted at an approach to summarize adjacency lists using a combination of an inductive skeleton and relations over set-valued variables, using such summaries poses significant algorithmic challenges, including both unfolding from and folding into such summaries. To be more concrete, consider the traversal algorithm shown in Figure 1(e) that is representative of graph operations that manipulate paths. Following graph edges amounts to traversing the cross pointers. This traversing of cross pointers makes the shape analysis of such programs tricky, since this step does not follow the inductive skeleton of the adjacency list—instead, it "jumps" to some other node in the structure.

In this paper, we propose a shape analysis that tracks set properties to infer precise invariants about data structures with unstructured sharing. Our contributions are as follows:

- The formalization of inductive predicates with set-valued parameters (Section 3). Such predicates enable a definition for the adjacency lists representation of graphs described here.
- A shape abstraction using such inductive definitions that is parameterized by a *set abstract domain* to track and infer relations over set-valued variables (Section 4).
- Static analysis algorithms to infer invariants over data structures with unstructured sharing (Section 5). These algorithms rely on novel notions of *non-local unfolding* to address the issue of "jumps" and *inductive set parameter synthesis* to enable folding into summaries with set-valued parameters. We then report on a preliminary empirical evaluation of these algorithms in Section 6.

2 Overview

Graph inductive predicate. The first step towards an analysis to verify graph algorithms is to set up inductive predicates to summarize the structure of Figure 1. Such

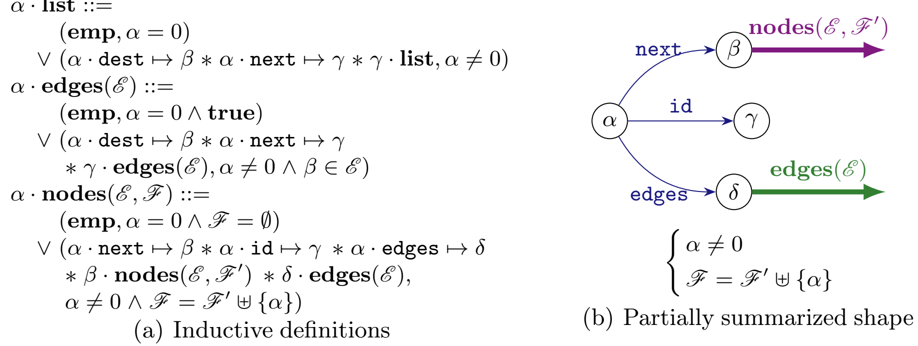


Fig. 2. Summarizing graph data-structures using inductive predicates

predicates are based on generic inductive definitions, which describe a disjunction of cases, each of which consists of a *memory formula* (a separating conjunction $F_0 * \ldots * F_n$ of points-to predicates of the form $\alpha \cdot \mathtt{f} \mapsto \beta$ and inductive predicates of the form $\alpha \cdot \iota$, where ι is another inductive definition) and a *pure formula* (a conjunction of value properties, such as pointer equalities). The set of outgoing edges of a node consists of a list of records, thus the predicate to summarize such a region can be based on a classical list inductive definition, such as $\alpha \cdot \mathbf{list}$, as shown in the top of Figure 2(a). This inductive definition states that α is either a null pointer (the list then spans over an empty region), or a non-null pointer to a record made of two fields but it does not further characterize the `dest` field.

However, this definition does not express that all instances of field `dest` contain a pointer to a node of the graph as the value β of that field is unconstrained. To resolve this issue, we simply need to add the constraint $\beta \in \mathscr{E}$, where $\mathscr{E}$ should denote the set of all node addresses in the graph. The abstract domain should also keep track of those predicates through folding and unfolding steps. Therefore, we obtain inductive definition **edges** shown in Figure 2(a), which takes the additional parameter $\mathscr{E}$, and where the value predicate of the non-empty case has been strengthened with set predicate $\beta \in \mathscr{E}$.

Moreover, the inductive definition of a graph needs to capture two set properties: (1) the destination of all edges are in set $\mathscr{E}$ (as described by inductive definition **edges**) and (2) the set of nodes in the adjacency list should correspond *exactly* to $\mathscr{E}$. Thus, inductive definition **nodes** shown in Figure 2(a) takes two set parameters: (1) $\mathscr{E}$ is constant over the whole induction and (2) $\mathscr{F}$ stands for the set of nodes described in the graph fragment described by an **nodes** instance. We note that the set predicates $\mathscr{F} = \emptyset$ (base case) and $\mathscr{F} = \mathscr{F}' \uplus \{\alpha\}$ (inductive case) guarantee that $\mathscr{F}$ is exactly the set of nodes described by predicate $\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F})$.

Abstraction of memory states. Using these definitions, a complete graph with set of nodes $\mathscr{E}$ can be fully summarized by inductive predicate $\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{E})$, where symbolic variable $\mathscr{E}$ denotes all the nodes of the concrete graph. Similarly, Figure 2(b) displays a partially summarized abstraction, following the splitting of Figure 1(d), where thin edges denote points-to predicates and bold edges stand for

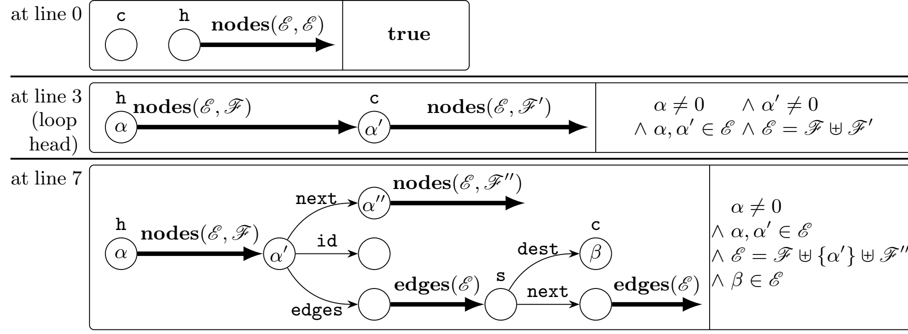


Fig. 3. Abstract pre-condition and local invariants

inductive predicates, annotated by inductive definition instances, with arguments denoting sets of concrete values. Colors are consistent with Figure 1(d) to highlight the memory region each edge describes. Additional set predicates ($\mathscr{E} = \mathscr{F}' \uplus \{\alpha\} \ldots$) are represented in a *set abstract domain* [11]. This means, that a concrete state represented by this state should bind α to an address and $\mathscr{E}, \mathscr{F}'$ to sets of addresses satisfying the aforementioned property, in the same way as the numerical constraint $\alpha \neq 0$ specifies α should be bound to a non-null pointer value.

Shape analysis extension. We now discuss automatic shape analysis algorithms to verify programs manipulating such graphs. Properties of interest include memory safety (absence of null or dangling pointer dereferences) and the preservation of structural invariants. As a benchmark property, we consider the verification of the memory safety of the random traversal algorithm of Figure 1(e). In particular, to establish no dangling pointer is dereferenced, the analysis should precisely track the fact that `c` should point to a valid node of the graph, or be the null pointer (which causes the program to exit) at all times. Figure 3 shows the main local abstract invariant involved in the verification of this property, described as shapes (complete invariants are shown in Appendix A). For the sake of concision, only parts of the shapes that play a role in the analysis are shown, and we discuss mainly the novel parts of the analysis. We use the same conventions as in Figure 2(b). The pre-condition shown at line 0 specifies that the function starts with a correct graph, with set of nodes $\mathscr{E}$. At line 1, cursor `c` is initialized to `h`. The analysis of the loop body requires the analysis to unfold [5] summaries to perform mutation over summarized regions, at lines 6 and 7, and to utilize a widening [5] operator for the convergence of the abstract iterates over both nested loops. The invariant at the head of the main loop, at line 3, shows a node list segment between nodes α and α' that denote the respective values of `h` and `c`. This segment describes a memory region encompassing a set of nodes of the graph together with their adjacency list. The segment predicate parameters are most interesting: the first specifies that all edges from nodes allocated in that region point to an address in $\mathscr{E}$ (global graph correctness property) whereas the second states that the set of the addresses of the nodes represented in that region is exactly $\mathscr{F}$. The side property $\mathscr{E} = \mathscr{F} \uplus \mathscr{F}'$ states that the splitting of the graph into the two summaries partitions its nodes.

The abstract state at line 7 is significantly more complex, and `c` does not immediately appear to point in the **nodes** inductive backbone anymore. Yet, the dereference of `c -> id` requires the materialization of an edge from that node, although no edge (points-to or summary) starts from node β. However, the analysis infers that $\beta \in \mathscr{E}$ (i.e., β is the address of an element of the graph adjacency list), and $\mathscr{E} = \mathscr{F} \uplus \{\alpha'\} \uplus \mathscr{F}''$. Thus, either $\beta \in \mathscr{F}$, or $\beta = \alpha'$, or $\beta \in \mathscr{F}''$. If $\beta \in \mathscr{F}$, then it is the address of a node in the segment from α to α', and it can be *materialized* as such, by splitting the segment (the case $\beta \in \mathscr{F}''$ is similar, and means that `c` points in the tail of the structure). This form of unfolding is much more complex than more conventional forms of inductive predicates unfolding [5] as it needs to utilize the set properties to localize β. To achieve this, the analysis needs to track all set predicates shown in Figure 3, through unfolding, updates and widening steps.

3 Inductive definitions with set predicates

The analysis presented in this paper is *parameterized* by a set of inductive definitions, which means the abstract domain is generic, and can deal with a wide family of data-structures. We extend the relational inductive definitions of [5] with set predicates. A concrete memory $m \in \mathbb{M} = \mathbb{V}_{\text{addr}} \longrightarrow \mathbb{V}$ maps addresses into values. Structure fields are considered numerical offsets, so that $a + \mathtt{f}$ denotes the address at base address a and offset `f`. In the abstract level, symbolic variables ($\alpha, \beta, \ldots \in \mathbb{V}^\sharp$) abstract numerical values. A *valuation* ν is a function that maps each numerical variable $\alpha \in \mathbb{V}^\sharp$ (resp., set variable $\mathscr{E} \in \mathbb{T}^\sharp$) to a numerical value $\nu(\alpha)$ (resp., set of numerical values $\nu(\mathscr{E})$). We write $\mathbb{Val}$ for the set of valuations.

$$
\begin{array}{rcll}
r & ::= & (F_{\text{Heap}}, F_{\text{Pure}}) & \\
F_{\text{Heap}} & ::= & \mathbf{emp} & \\
& | & \alpha \cdot \mathtt{f} \mapsto \beta & \\
& | & \alpha \cdot \iota(\mathscr{E}_0, \ldots, \mathscr{E}_{n-1}) & \\
& | & F_{\text{Heap}} * F_{\text{Heap}} & \\
F_{\text{Pure}} & ::= & \alpha = c & (c \in \mathbb{V}) \\
& | & \alpha \neq c & (c \in \mathbb{V}) \\
& | & \alpha \in \mathscr{E} & \\
& | & \mathscr{E} = \{\alpha\} \uplus \mathscr{F} & \\
& | & \mathscr{E} = \{\alpha\} \cup \mathscr{F} & \\
& | & \ldots &
\end{array}
$$

Inductive definitions. An *inductive definition* $\alpha \cdot \iota(\mathscr{E}_1, \ldots, \mathscr{E}_n) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ takes a pointer parameter α and a list of set parameters $\mathscr{E}_1, \ldots, \mathscr{E}_n$ and defines a scheme to summarize heap regions that satisfy some inductive property, specified as a disjunction of *rules*, which comprise a *heap* part and a *pure* part, as described in the inset. The heap part is a separating conjunction of memory cells (predicates of the form $\alpha \cdot \mathtt{f} \mapsto \beta$) and recursive calls to inductive definitions. The pure part comprises not only numerical constraints, but also set constraints, over the symbolic variables exposed in the heap part, and the set parameters $\mathscr{E}_1, \ldots, \mathscr{E}_n$, as shown in F_{Pure}. The intuitive meaning of a set constraint such as $\alpha \in \mathscr{E}$ is that the concretization will map α into a numerical value that belongs to the concretization of $\mathscr{E}$. As a simple example, the inductive definition below characterizes the singly linked list starting at address α, such that the set of addresses of list elements is exactly $\mathscr{E}$:

$$
\begin{array}{rl}
\alpha \cdot \mathbf{l}_{\text{s}}(\mathscr{E}) ::= & (\mathbf{emp}, \alpha = 0 \wedge \mathscr{E} = \emptyset) \\
& \vee\ (\alpha \cdot \mathtt{n} \mapsto \beta_0 * \alpha \cdot \mathtt{v} \mapsto \beta_1 * \beta_0 \cdot \mathbf{l}_{\text{s}}(\mathscr{E}'), \alpha \neq 0 \wedge \mathscr{E} = \{\alpha\} \uplus \mathscr{E}')
\end{array}
$$

Inductive definitions **edges** and **nodes** (Figure 2(a)) capture set constraints over the nodes and edges of graphs in a similar way: **nodes** collects *exactly* the set of

all nodes of the graph, whereas **edges** asserts all edges should point to one of the nodes of the graph.

Properties of set parameters. Analysis algorithms (Section 5) need to infer instances of inductive definitions, including their set parameters. Computing set parameters accurately is a very hard task, as it depends on complex properties of the data-structures shapes and contents. Yet, properties of set parameters may make their computation simpler. We define the following set parameters *kinds*:

– **Constant parameters.** In definition **edges** (Figure 2(a)), the set parameter is propagated with no modification to the recursive call of the inductive definition. We call it a *constant parameter*, that is a parameter $\mathscr{E}$ of inductive definition ι such that any recursive call $\alpha' \cdot \iota(\mathscr{E}')$ in the definition of $\alpha \cdot \iota(\mathscr{E})$ is such that $\mathscr{E} = \mathscr{E}'$. We note that the first parameter ($\mathscr{E}$) of **nodes** also satisfies this property.

– **Head parameters:** The second parameter of **nodes** is clearly not constant, but it satisfies another interesting property: it collects the set of head nodes in all recursive inductive calls, and can be computed exactly from the values of the same parameters in the recursive calls, since $\mathscr{F} = \emptyset$ in the empty rule and $\mathscr{F} = \{\alpha\} \uplus \mathscr{F}'$ in the second rule, where α is the address of the head of the structure and $\mathscr{F}'$ is the parameter of the tail. We call such a parameter a *head parameter*. This definition generalizes to non-linear structures (i.e., with several recursive calls, corresponding to distinct sub-structures). Parameter $\mathscr{E}$ of the $\mathbf{l}_{\mathrm{s}}$ definition also satisfies this property.

These set parameter kinds are computed by a very simple analysis of inductive definitions, shown in Appendix B. In the following, we write $\iota \vdash \mathscr{E} : \mathbf{cst}$ (resp., $\iota \vdash \mathscr{E} : \mathbf{head}$) to denote that parameter $\mathscr{E}$ of inductive definition ι is constant (resp., head).

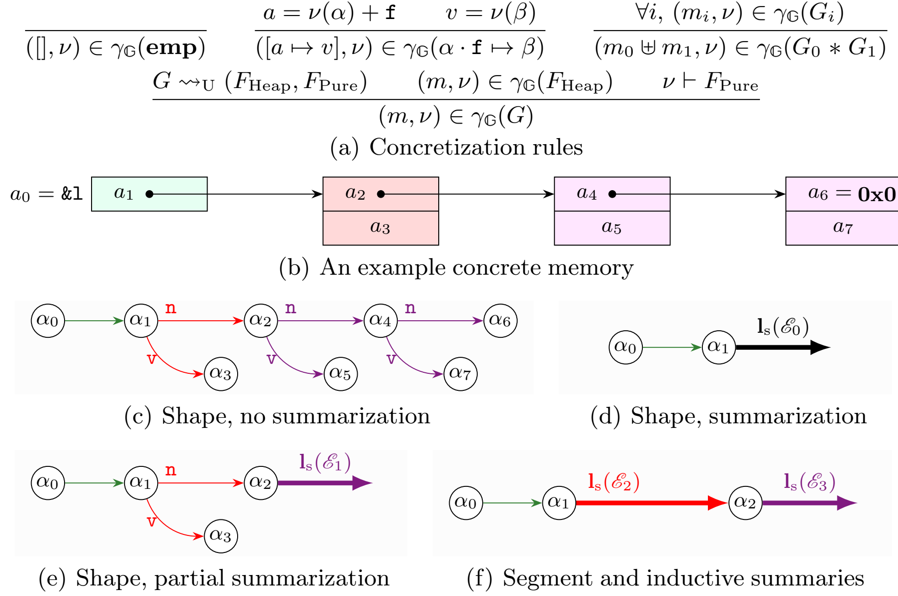
In this paper, we provide static analysis algorithms that are sound whatever the properties of the set parameters. However, precise folding and materialization will only be supported for *constant* and *head* parameters. Note that other kinds of set parameters may be proposed though, so as to recover precise static analyses even when the above properties are not satisfied.

4 Composite memory abstraction with set predicates

We now formalize our shape abstract domain $\mathbb{M}^\sharp$, that is parameterized by the inductive predicates of Section 3, and an abstract domain $\mathbb{S}^\sharp$ for constraints over value and set symbolic variables.

$$
\begin{array}{rl}
G ::= & \mathbf{emp} \\
| & \alpha \cdot \mathtt{f} \mapsto \beta \\
| & \alpha \cdot \iota(\vec{\mathscr{E}}) \\
| & \alpha \cdot \iota(\vec{\mathscr{E}}) \mathrel{*\!\!=} \beta \cdot \iota(\vec{\mathscr{E}}) \\
| & G * G
\end{array}
$$

Abstract states. An *abstract state* is a pair (G, S) made of a *shape* $G \in \mathbb{G}^\sharp$ and an element $S \in \mathbb{S}^\sharp$. The syntax of shapes is shown in the inset: a shape is either empty, or a single points-to edge $\alpha \cdot \mathtt{f} \mapsto \beta$, or an inductive predicate $\alpha \cdot \iota(\vec{\mathscr{E}})$ (instantiating an inductive definition ι defined as in Section 3), or a segment $\alpha \cdot \iota(\vec{\mathscr{E}}) \mathrel{*\!\!=} \beta \cdot \iota(\vec{\mathscr{E}})$ describing an incomplete inductive structure, or a separating product of such predicates. Intuitively, a



(a) Concretization rules

(b) An example concrete memory

(c) Shape, no summarization

(d) Shape, summarization

(e) Shape, partial summarization

(f) Segment and inductive summaries

Fig. 4. Shapes and their concretizations

segment describes incomplete induction, with a missing sub-structure, hence a segment of an **list** structure effectively describes a conventional list segment (similarly, a tree segment would describe a "tree minus a subtree"). Inductive and segment predicates comprise a number of parameters that matches their definition (Section 3). Our analysis supports inductive predicates with any number of set parameters (although, in the rest of the paper, we sometimes write properties in the case of definitions using a single parameter, for the sake of readability).

Concretization. We assume that elements of $\mathbb{S}^\sharp$ concretize into sets of valuations, satisfying both sets and value constraints, thus $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathcal{P}(\mathbb{Val})$. Similarly, the concretization of a shape is a set of memory states together with value and set valuations, thus $\gamma_{\mathbb{G}} : \mathbb{G}^\sharp \to \mathcal{P}(\mathbb{M} \times \mathbb{Val})$. Figure 4(a) displays the concretization rules for shapes. The first three rules describe the usual concretization for empty shapes, single points-to edges and separating conjunction [24]. The last rule defines the concretization for inductive and segment predicates using the standard notion of *syntactic unfolding*: the unfolding of an inductive or segment predicate selects a rule r in the corresponding inductive definition, and replaces the predicate with the heap part of r and constrains value and set valuations with the pure part of r. This construction is standard and is fully described in [7].

Examples. Figure 4 shows a few abstractions of the concrete memory state m shown in Figure 4(b), where `l` stores a pointer to a list of length 3, and where $a_0, a_1, \ldots, a_7$ denote numerical values / addresses. The shape of Figure 4(c) abstracts this state without any summarization (it contains no inductive predicate). Its concretization

into m results in $\forall i,\ \nu(\alpha_i) = a_i$ (in particular, $\nu(\alpha_6) = a_6 = \mathbf{0x0}$), and can be fully expressed using the points-to and separating product rules of Figure 4(a). The shape of Figure 4(d) summarizes the list completely into a single inductive predicate $\alpha_1 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_0)$. In this case, the concretization also needs to bind $\mathscr{E}_0$ to a set of concrete addresses: by the definition of $\mathbf{l}_\mathrm{s}$ (Section 3), this boils down to $\nu(\mathscr{E}_0) = \{a_1, a_2, a_4\}$. Moreover, this concretization needs to trigger the unfolding rule (last rule in Figure 4(a)) in order to produce the shape of Figure 4(c). The shape of Figure 4(e) summarizes only the last two elements of the list (in purple) while the first element (in red) is preserved in its unfolded form. Similarly to the previous case, the unfolding of this shape needs to trigger three times the unfolding rule in order to get the shape of Figure 4(c) and to map $\mathscr{E}_1$ to $\nu(\mathscr{E}_1) = \{a_2, a_4\}$.

Finally, the shape of Figure 4(f) summarizes two list elements with inductive predicate $\alpha_2 \cdot \mathbf{l}_\mathrm{s}(\mathscr{E}_3)$ (in purple) and the rest of the list with a segment predicate (in red). This segment predicate should take a form $\alpha_1 \cdot \mathbf{l}_\mathrm{s}(\mathscr{F}_1) \ast\!= \alpha_2 \cdot \mathbf{l}_\mathrm{s}(\mathscr{F}_2)$, where $\mathscr{F}_1, \mathscr{F}_2$ describe two sets of addresses such that the set of addresses of the list elements summarized in the segment is exactly $\nu(\mathscr{F}_1) \setminus \nu(\mathscr{F}_2)$, by the definition of $\mathbf{l}_\mathrm{s}$. The fact that only the difference of these two sets matters is actually a direct consequence of the fact that the parameter of $\mathbf{l}_\mathrm{s}$ is *head* (Section 3). Therefore, when an inductive definition parameter is head, segment predicates are decorated with only one parameter. In the example of Figure 4(f), the segment predicates writes down $\alpha_1 \cdot \mathbf{l}_\mathrm{s} \ast\!=_{(\mathscr{E}_2)} \alpha_2 \cdot \mathbf{l}_\mathrm{s}$; the graphical notation in the figure condenses this slightly, into a single $\mathbf{l}_\mathrm{s}(\mathscr{E}_2)$ parameter. The concretization of this shape produces the memory of Figure 4(b) with $\nu(\mathscr{E}_2) = \{a_1\}$ and $\nu(\mathscr{E}_3) = \{a_2, a_4\}$.

Properties of constant and head parameters. The parameters kinds introduced in Section 3 allow to prove properties allowing to fold segment and inductive predicates, such as the following concretization inclusions / implications:

- if $\iota \vdash \mathscr{E} : \mathbf{cst}$, then $\gamma_\mathbb{G}(\alpha_0 \cdot \iota \ast\!=_{(\mathscr{E})} \alpha_1 \cdot \iota \ast \alpha_1 \cdot \iota(\mathscr{E})) \subseteq \gamma_\mathbb{G}(\alpha_0 \cdot \iota(\mathscr{E}))$, and $\gamma_\mathbb{G}(\alpha_0 \cdot \iota \ast\!=_{(\mathscr{E})} \alpha_1 \cdot \iota \ast \alpha_1 \cdot \iota \ast\!=_{(\mathscr{E})} \alpha_2 \cdot \iota) \subseteq \gamma_\mathbb{G}(\alpha_0 \cdot \iota \ast\!=_{(\mathscr{E})} \alpha_2 \cdot \iota)$;
- if $\iota \vdash \mathscr{E} : \mathbf{head}$, $(m, \nu) \in \gamma_\mathbb{G}(\alpha_0 \cdot \iota \ast\!=_{(\mathscr{E}_0)} \alpha_1 \cdot \iota \ast \alpha_1 \cdot \iota(\mathscr{E}_1))$ and $\nu \vdash \mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1$, then $(m, \nu) \in \gamma_\mathbb{G}(\alpha_0 \cdot \iota(\mathscr{E}))$.

As an example, the last of these rules allows to show that the shape in Figure 4(d) over-approximates that of Figure 4(f) under the condition that $\mathscr{E}_0 = \mathscr{E}_2 \uplus \mathscr{E}_3$.

Abstraction of constraints over sets and addresses. Abstract domain $\mathbb{S}^\sharp$ should provide an abstraction for constraints over set variables and symbolic variables. For instance, the constraints of the invariant of Figure 3 corresponding to line 3 in the program of Figure 1(e) collects value constraints $\alpha \neq 0$, $\alpha' \neq 0$ and set constraints $\alpha, \alpha' \in \mathscr{E}$, $\mathscr{E} = \mathscr{F} \uplus \mathscr{F}'$, thus abstract domain $\mathbb{S}^\sharp$ should be able to express these constraints. A suitable such abstract domain can be obtained as a reduced product [10] of the interval abstract domain [9], of a domain representing inequalities and of a set abstract domain (two set domains were used in the evaluation—see Section 6 for discussion). For more expressiveness, more powerful numerical / set abstract domains can be used instead.

Combined abstract domain. The concretization of an abstract memory state $M = (G, S) \in \mathbb{M}^\sharp$ is defined as the set of memory states for which a pair of valuations can be found, that satisfy all constraints from G and S:

$$\gamma_{\mathbb{M}}(G, S) = \{m \in \mathbb{M} \mid \exists \nu \in \gamma_{\mathbb{S}}(S),\ (m, \nu) \in \gamma_{\mathbb{G}}(G)\}$$

While this concretization looks similar to that of a reduced product [9], the composite abstract domain actually has the structure of a *cofibered abstract domain* [29], since the set of symbolic variables present in the S component are exactly the nodes in shape G, and the analysis should maintain this consistency at all times.

5 Static analysis algorithms

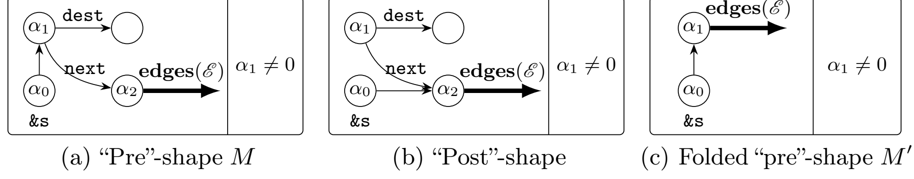
We now describe algorithms to infer invariants involving inductive predicates with set parameters. Extending [5], it inputs an abstract pre-condition, and performs a forward abstract interpretation to compute a *sound abstract post-condition* satisfied by any execution starting from the pre-condition. We emphasize the novel aspects of this analysis, namely non-local unfolding (in Section 5.1) and set parameters synthesis during folding (in Section 5.2) and defer full algorithm definitions to Appendix C. Our analysis assumes abstract domain $\mathbb{S}^\sharp$ provide *sound* abstract join $\sqcup_{\mathbf{S}}$, abstract inclusion test $\sqsubseteq_{\mathbf{S}}$, widening $\triangledown_{\mathbf{S}}$, supremum $\top$ and *sound transfer functions*: $\mathfrak{guard}_{\mathbb{S}}$ inputs an abstract value S and a set constraint C, and returns an abstract value refined with C and $\mathfrak{prove}_{\mathbb{S}}$ inputs an abstract value S and a set constraint and returns **true** when it successfully establishes that S entails C.

5.1 Transfer functions, local and non-local unfolding

Given a concrete post-condition function $\mathbf{f} : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$, the corresponding *abstract transfer function* $\mathbf{f}^\sharp : \mathbb{M}^\sharp \to \mathbb{M}^\sharp$ should over-approximate the effect of $\mathbf{f}$, in the sense that $\mathbf{f} \circ \gamma_{\mathbb{M}} \subseteq \gamma_{\mathbb{M}} \circ \mathbf{f}^\sharp$. In this section, we consider the case of a pointer assignment and let f be $[\![\mathtt{l} := \mathtt{e}]\!]$ since condition tests, allocation, and deallocation follow similar principles. We assume pre-condition $M = (G, S)$. If l-value $\mathtt{l}$ evaluates to points-to edge $\alpha \cdot \mathtt{f} \mapsto \beta$ in M, r-value $\mathtt{e}$ evaluates to node β' in M and $G = \alpha \cdot \mathtt{f} \mapsto \beta * G'$, then the abstract assignment should simply replace the old edge with a new one and produce $[\![\mathtt{l} := \mathtt{e}]\!](M) = (\alpha \cdot \mathtt{f} \mapsto \beta' * G', S)$. The local reasoning principle [24,18] ensures the soundness of this mechanism.

As an example, Figure 5(a) and 5(b) display the pre and post-condition of assignment $\mathtt{s} = \mathtt{s} \mathbin{\texttt{->}} \mathtt{next}$ in the program of Figure 1(e), at line 5 (for the sake of clarity, shapes are simplified to the relevant memory regions). In pre-condition M (Figure 5(a)), the modified memory cell corresponds to edge $\alpha_0 \mapsto \alpha_1$, and the r-value describes node α_2. Thus, abstract transfer function $[\![\mathtt{s} := \mathtt{s} \mathbin{\texttt{->}} \mathtt{next}]\!]^\sharp$ simply returns the shape of Figure 5(b).

The unfolding transformation. However, the above scheme cannot work when the over-written memory cell or the memory read in the r-value is part of an



(a) "Pre"-shape M (b) "Post"-shape (c) Folded "pre"-shape M'

Fig. 5. Assignment and (local) unfolding

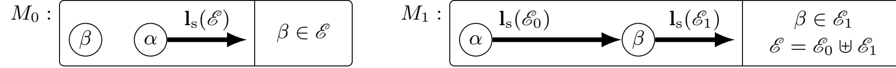


Fig. 6. Non-local unfolding

inductive predicate. This case actually happens in the analysis of the program of Figure 1(e), since the actual pre-condition is shown in Figure 5(c) and does not allow to evaluate `s -> next` into a node. *Unfolding* [14,1,5] resolves this issue, by replacing the inductive predicate $\alpha_1 \cdot \mathbf{edges}(\mathscr{E})$ with its inductive definition and producing a disjunction of two cases (one per inductive rule).

Theorem 1 (Unfolding soundness). *Given inductive definition* $\alpha \cdot \iota(\mathscr{E}) ::= \bigvee\{(F_{\mathrm{Heap},i}, F_{\mathrm{Pure},i}) \mid 1 \leq i \leq k\}$*:*

$$\gamma_{\mathbb{M}}(\alpha \cdot \iota(\mathscr{E}) * G, S) \subseteq \bigcup_{i=1}^{k} \gamma_{\mathbb{M}}(F_{\mathrm{Heap},i} * G, \mathfrak{guard}_{\mathrm{S}}(F_{\mathrm{Pure},i}, S))$$

In the example of Figure 5, the inductive rule corresponding to the empty list of edges is ruled out, since M' contains constraint $\alpha_1 \neq 0$ in the $\mathbb{S}^\sharp$ component, thus unfolding produces M (Figure 5(a)) as a single disjunct. Thus, when applied to M' abstract transfer function $[\![\mathtt{s} := \mathtt{s} \text{ -> } \mathtt{next}]\!]^\sharp$ first invokes the unfolding procedure, and then proceeds as explained above. Theorem 1 ensures the soundness of the resulting abstract operations. We write $G \rightsquigarrow_{\mathrm{U}} (G_u, F_{\mathrm{Pure}})$ when a predicate of G can be unfolded so as to produce G_u with side constraints F_{Pure}.

Non-local unfolding. The unfolding case studied so far is quite straightforward as the node at which inductive predicate should be unfolded is well specified, by the transfer function (α_1 in Figure 5). The analysis of an instruction reading `c -> id` from the abstract state corresponding to line 7 appears in Figure 3: in this state `c` points to β in the abstract level, but node β is neither the origin of a points-to predicate nor that of an inductive predicate that could be unfolded. Intuitively, β could be any node in the graph as shown by the set property $\beta \in \mathscr{F} \uplus \{\alpha'\} \uplus \mathscr{F}''$, thus we expect the abstract memory state to reflect this. This intuition is formalized as follows: the second parameter of **nodes** is a *head parameter* (Section 3), and the side predicates carry out the fact that $\beta \in \mathscr{F} \uplus \{\alpha'\} \uplus \mathscr{F}''$, where $\mathscr{F}$ and $\mathscr{F}''$ appear as a second parameter for both **nodes** predicates in the shape, which allows to localize β. This principle is a direct consequence of a property of head parameters:

Theorem 2 (Non-local unfolding principle). *Let* ι *be a single parameter inductive, such that* $\alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$. *Let* $(m, \nu) \in \gamma_{\mathbb{G}}(\alpha \cdot \iota(\mathscr{E}))$ *such that* $\nu(\beta) \in \nu(\mathscr{E})$. *Given* $\mathscr{E}_0, \mathscr{E}_1$ *fresh set variables,* ν *can be extended into* ν', *such that* $(m, \nu') \in \gamma_{\mathbb{G}}(\alpha \cdot \iota \ast\!\!=_{(\mathscr{E}_0)} \beta \cdot \iota \ast \beta \cdot \iota(\mathscr{E}_1))$, $\nu'(\mathscr{E}) = \nu'(\mathscr{E}_0) \uplus \nu'(\mathscr{E}_1)$ *and* $\nu'(\beta) \in \nu'(\mathscr{E}_1)$.

The proof follows directly from the definition of head parameters. Figure 6 illustrates this *non-local* unfolding principle. While theorem 2 states the result for inductive definitions with a single set parameter, the result generalizes directly to the case of definitions with several parameters (only the parameter supporting non-local unfolding is then required to be a head parameter). It also generalizes to segments.

To conclude, the analysis of an assignment proceeds along the following steps:

1. it attempts to evaluate all l-values to edges and r-values to symbolic variables;
2. when step 1 fails as no points-to edge can be found at offset $\alpha \cdot \mathtt{f}$, it searches for a local unfolding at α, that is either a segment or an inductive predicate starting from α;
3. when no such local unfolding can be found, it searches for predicates of the form $\alpha \in \{\alpha_0, \ldots, \alpha_k\} \uplus \mathscr{E}_0 \uplus \ldots \uplus \mathscr{E}_l$, where $\mathscr{E}_0, \ldots, \mathscr{E}_l$ appear as head parameters; when it finds such a predicate, the analysis produces a disjunction of cases, where either $\alpha = \alpha_i$ (and it goes back to step 1), or where $\alpha \in \mathscr{E}_i$ and it performs non-local unfolding of the corresponding predicate (Theorem 1);
4. last, it performs the abstract operation on the unfolded disjuncts

Note that failure to fully materialize all required nodes would produce imprecise results; thus, in absence of information about parameters, the analysis may fail to produce a precise post-condition.

5.2 Folding of inductive summaries: inclusion test, join and widening

While transfer functions *unfold* inductive predicates, inclusion checking, join and widening operators need to discover valid set parameters so as to *fold* them back.

Inclusion checking. The inclusion checking abstract operation $\sqsubseteq$ inputs two abstract memory states M_l, M_r and returns **true** if it successfully establishes that $\gamma_{\mathbb{M}}(M_l) \subseteq \gamma_{\mathbb{M}}(M_r)$ (it is conservative) using inclusion testing functions $\sqsubseteq_{\mathbf{G}}, \sqsubseteq_{\mathbf{S}}$. It attempts to construct a proof of inclusion, following a set of logical rules an excerpt of which is shown in Figure 7(a). Some rules were already introduced in [5]. For instance, the bottom left rule states that any shape is included in itself; the inclusion checking algorithm actually applies it to single predicates (points-to, inductives or segments). Inclusion checking splits shapes according to the separation principle. It may unfold the right hand side shape and try to match the left hand side with one of the disjuncts. Last, it returns **true** when both the comparison of shapes and of side predicates return **true**.

However, the matching of segments and inductive predicates with set parameters requires some specific rules. Figure 7(a) shows two such rules, that apply when trying to compare a segment on the left and an inductive predicate on the right, that correspond to the same definition and the same origin:

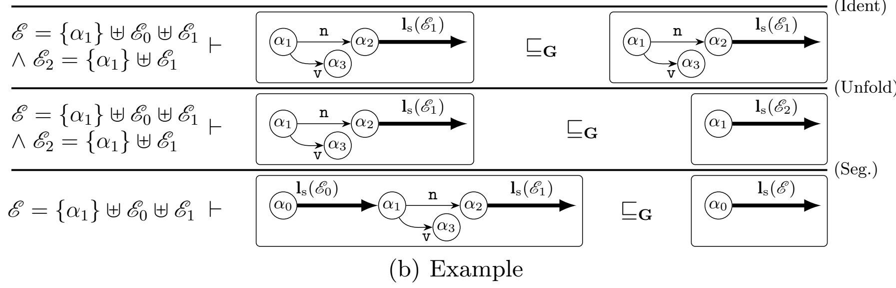
- The bottom middle rule applies to the case of a *constant* set parameter and simply requires inductive and segment to share the same set parameter.

$$\frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r}{S_l \vdash G_l * G \sqsubseteq_{\mathbf{G}} G_r * G} \qquad \frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_u \qquad G_r \rightsquigarrow_{\mathrm{U}} (G_u, F_{\mathrm{Pure}}) \qquad \mathfrak{prove}_{\mathrm{S}}(S_l, F_{\mathrm{Pure}}) = \mathbf{true}}{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r}$$

$$\frac{\mathfrak{prove}_{\mathrm{S}}(S_l, \mathscr{E}_0 \subseteq \mathscr{E}) \qquad \mathscr{E}_1 \text{ fresh (denotes } \mathscr{E} \setminus \mathscr{E}_0) \qquad S_l \vdash G_l \sqsubseteq_{\mathbf{G}} \beta \cdot \iota(\mathscr{E}_1) \qquad \alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}}{S_l \vdash \alpha \cdot \iota *\!=_{(\mathscr{E}_0)} \beta \cdot \iota * G_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota(\mathscr{E})}$$

$$\frac{}{S_l \vdash G \sqsubseteq_{\mathbf{G}} G} \qquad \frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} \beta \cdot \iota(\mathscr{E}) \qquad \alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{cst}}{S_l \vdash \alpha \cdot \iota *\!=_{(\mathscr{E})} \beta \cdot \iota * G_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota(\mathscr{E})} \qquad \frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r \qquad S_l \sqsubseteq_{\mathbf{S}} S_r}{\vdash (G_l, S_l) \sqsubseteq (G_r, S_r)}$$

(a) Logical rules for inclusion checking over shapes ($\sqsubseteq_{\mathbf{G}}$) and over memory states ($\sqsubseteq$)



(b) Example

Fig. 7. Inclusion checking

- The middle rule applies to the case of a *head* set parameter and enforces its additiveness by checking $\mathscr{E}_0 \subseteq \mathscr{E}$ and choosing fresh $\mathscr{E}_1$ so that $\mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_1$, following the properties of head parameters shown in Section 4.

Similar rules (described in Appendix C.1) apply to the comparison of segments in both sides. Soundness follows from the soundness of each rule:

Theorem 3 (Inclusion checking soundness). *If* $S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r$, $(m, \nu) \in \gamma_{\mathbb{G}}(G_l)$, *and* $\nu \in \gamma_{\mathbb{S}}(S_l)$, *then* $(m, \nu) \in \gamma_{\mathbb{G}}(G_r)$. *Moreover, if* $\vdash (G_l, S_l) \sqsubseteq (G_r, S_r)$, *then* $\gamma_{\mathbb{M}}(G_l, S_l) \subseteq \gamma_{\mathbb{M}}(G_r, S_r)$.

Figure 7(b) illustrates this algorithm on an example based on the $\mathbf{l}_{\mathrm{s}}$ definition, which behaves similarly to **nodes** in its second parameter (it resembles inclusion tests ran in the analysis of the program of Figure 1(e)). The inclusion proof search starts from the bottom shapes, where α_0 is the origin of a segment in the left, and of an inductive predicate in the right. Since $\mathbf{l}_{\mathrm{s}}$ has a *head* parameter, the rule specific to this case applies, and the inclusion test "consumes" the segment, effectively removing it from the left argument, and adding a fresh $\mathscr{E}_2$ variable, such that $\mathscr{E} = \mathscr{E}_0 \uplus \mathscr{E}_2$ (thus, constraint $\mathscr{E}_2 = \{\alpha_1\} \uplus \mathscr{E}_1$ is added). Then, the algorithm derives inclusion holds after unfolding and matching three pairs of identical predicate.

Join and widening. In the shape domain $\mathbb{G}^\sharp$, *join* and *widening* rely on the same algorithm. A basic version of this algorithm is formalized in [5], and we extend it here so as to handle set parameters. To over-approximate $M_l = (G_l, S_l)$ and $M_r = (G_r, S_r)$, the join algorithm starts from a configuration $[M_l \sqcup_{\mathbf{G}} M_r | \mathbf{emp}]$, and incrementally selects components of both inputs that can be over-approximated likewise, and moves their common over-approximation to the third (output) element. The two fundamental rules are shown in Figure 8(a). The first rule states that, if

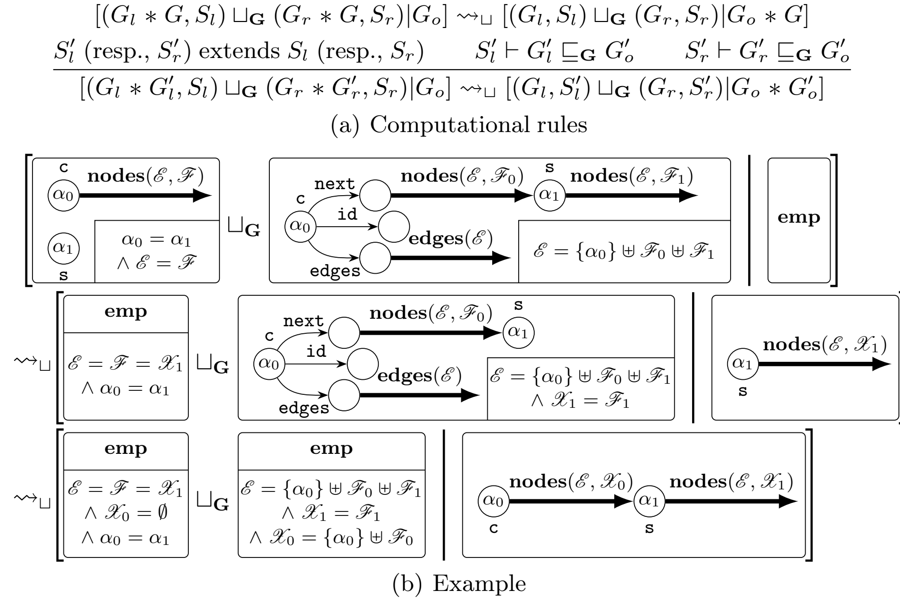


Fig. 8. Join and widening algorithms

both inputs contain a same region G, then these regions can be joined immediately (it is applied incrementally for points-to, inductive and segment predicates). The second rule states that, if a common over-approximation G_o for G_l, G_r can be found and checked with $\sqsubseteq_{\mathbf{G}}$, abstract join can rewrite these into G_o. This algorithm is a widening: it ensures termination of sequences of abstract iterates.

Novel rules are needed to *introduce* segments and inductive predicates. Let ι be an inductive definition with a single set parameter. Then, a segment predicate $\alpha \cdot \iota *\!\!=_{(\mathscr{E})} \beta \cdot \iota$ (where $\mathscr{E}$ is fresh) can be introduced when $G_l = \mathbf{emp}$, $S \vdash \alpha = \beta$, $S_r \vdash G_r \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota *\!\!=_{(\mathscr{E})} \beta \cdot \iota$, and when:

- either $\mathscr{E}$ is *constant* ($\alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{cst}$);
- or $\mathscr{E}$ is a *head parameter* ($\alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head}$), and $S'_l = \mathfrak{guard}_{\mathbb{S}}(S_l, \mathscr{E} = \emptyset)$;

The inclusion checking algorithm may then discover new constraints between fresh variable $\mathscr{E}$ and the other set variables, and enrich S_r accordingly. These constraints indirectly stem from the constant or head kind of the set parameters. The full list of rule instances is shown in Appendix C.2, and allows to introduce inductive predicates, and extend segment or inductive predicates in a similar way, as the above segment weakening. Soundness follows from step by step preservation of concretization (the convergence property of the shape join is proved in [5]):

Theorem 4 (Soundness of join). *If* $[(G_l, S_l) \sqcup_{\mathbf{G}} (G_r, S_r) | G_o] \rightsquigarrow_{\sqcup} [(G'_l, S'_l) \sqcup_{\mathbf{G}} (G'_r, S'_r) | G'_o]$*, then,* $\forall i \in \{l, r\}$*,* $\gamma_{\mathbb{M}}(G_i * G_o, S_i) \subseteq \gamma_{\mathbb{M}}(G'_i * G'_o, S'_i)$*.*

Therefore, if $[M_l \sqcup_{\mathbf{G}} M_r | \mathbf{emp}] \rightsquigarrow_{\sqcup}^{\star} [(\mathbf{emp}, S'_l) \sqcup_{\mathbf{G}} (\mathbf{emp}, S'_r) | G'_o]$*, then* $(G'_o, S'_l \sqcup_{\mathbf{S}} S'_r)$ *(resp.,* $(G'_o, S'_l \triangledown_{\mathbf{S}} S'_r)$*) provides a sound join (resp., widening) for* M_l, M_r*.*

Description	LOCs	Nested loops	"BDD" time (ms) Total	Shape	Set	"BDD" Property	"LIN" time (ms) Total	Shape	Set	"LIN" Property
Node: add	27	0	44	0.3	11	**yes**	28	0.3	0.2	**yes**
Edge: add	26	0	31	0.2	4	**yes**	27	0.2	0.1	**yes**
Edge: delete	22	0	45	0.4	16	**yes**	30	0.3	0.2	**yes**
Node list traversal	25	1	117	1.5	87	**yes**	28	0.5	0.3	**yes**
Edge list iteration + dest. read	34	1	332	2.7	293	**yes**	36	3.5	2.4	**yes**
Graph path: deterministic	31	2	360	2.7	323	**yes**	35	2.4	2	**yes**
Graph path: random	43	2	765	7.1	711	**yes**	41	4.1	3	**yes**

Table 1. Analysis of a set of fundamental graph manipulation functions. Analysis times (in milliseconds) are measured on one core of an Intel Xeon at 3.20GHz with 16GB of RAM running Ubuntu 14.04.2 LTS (we show overall time including front-end and iterator shape domain and set domain), with "BDD" and "LIN" set domains. "Property" columns: inference of structural properties.

Figure 8(b) shows a simplified instance of a join taken from the analysis of the program of Figure 1(e). Initially, both inputs contain very similar **nodes** inductive predicates at α_0, thus the first step moves these predicates into the output component. The first parameter is constant, and is equal to $\mathscr{E}$ everywhere. The second parameter is a head parameter, so a new variable $\mathscr{X}_1$ is introduced, and S_l (resp., S_r) is enriched with constraint $\mathscr{X}_1 = \mathscr{F}$ (resp., $\mathscr{X}_1 = \mathscr{F}_1$). In the second step, a segment is introduced. Again, the constant parameter is equal to $\mathscr{E}$ everywhere. In the left, constraint $\mathscr{X}_0 = \emptyset$ is added. In the right, inclusion check discovers constraint $\mathscr{X}_0 = \{\alpha_0\} \uplus \mathscr{F}_0$. This configuration is final, and allows to compute the set constraints $\mathscr{E} = \mathscr{X}_0 \uplus \mathscr{X}_1$.

6 Empirical evaluation

We implemented inductive definitions with set predicates into the MemCAD static analyzer [26,27] and integrated set constraints as part of the numerical domain [6], so as to assess (1) whether it achieves the verification of structure preservation in the presence of sharing, and (2) if the memory abstract domain efficiency is preserved. The analysis takes a set abstract domain as a parameter to represent set constraints. We have considered two set abstract domains:

- the first one is based on an encoding of set constraints into BDDs, and utilizes a BDD library [17], following an idea of [11] (this domain is called "BDD" in the results table);
- the second one relies on a compact representation of constraints of the form $\mathscr{E}_i = \{\alpha_0, \ldots, \alpha_n\} \uplus \mathscr{F}_0 \uplus \ldots \uplus \mathscr{F}_m$ as well set equalities, inclusion and membership constraints (this domain is called "LIN" in the results table, since the main set constraints expressed here are of "linear" form).

Both abstract domains were implemented in OCaml, and integrated into the MemCAD static analyzer.

The analysis inputs inductive definition parameters, a C program, and a precondition, and computes and verifies abstract post-conditions. We ran the analysis on a basic graph library, chosen to assess specifically the handling of shared

structures (addition or removal operations, structure traversals, and traversals following paths including the program of Figure 1(e)). Results are shown in Table 1. In all cases, the analysis successfully establishes memory safety (absence of null / dangling pointer dereference), structural preservation for the graph modifying functions, and precise cursor localization for the traversal functions, with both set domains. Note that, in the case of path traversals, memory safety requires the analysis to localize the cursor as a valid graph node, at all times (the strongest set property, captured by the graph inductive definitions of Figure 2). The analysis time spent in the shape domain is in line with those usually observed in the analyzer [26,27], yet the BDD-based set domain proves inefficient in this situation and accounts for most of the analysis time for two reasons: (1) it is far too expressive and keeps properties that are not relevant to the analysis and (2) set variables renaming (required after joins) necessitate full recomputation of BDDs. By contrast, the "LIN" set domain is tailored for the predicates required in the analysis, and produces very quick analysis run-times. In several cases, the time spent in the shape domain is even reduced compared to "BDD", due to shorter iteration sequences.

7 Related work

A wide family of shape analysis techniques have been proposed so as to deal with inductive structures, often based on 3-valued logic [25,21] or on separation logic [2,14,1,23,4,5]. Such analyses often deal very well with list and tree like structures, but are often challenged with *unbounded* sharing. In this paper, we augmented a separation logic based analysis [7,5] with set predicates to account for unbounded sharing, both in summaries and in unfolded regions, and retain the parameterizability of this analysis. A shape analysis tracking properties of structure contents was presented in [28], although with a less general set abstraction interface, and without support for unfolding guided by set parameters. Another related approach was proposed in [8], that utilizes a set of data-structure specific analysis rules and encodes sharing information into instrumentalized variables in order to analyze programs manipulating trees and graphs. By contrast, our analysis does no such instrumentation and requires no built-in inductive definitions. Recently, a set of works [15,19,20,26] targeted *overlaid* structures, which feature some form of *structured* sharing, such as a tree overlaid on a list. Typically, these analyses combine several abstractions specific to the different layer. We believe that problem is orthogonal to ours, since we consider a form of sharing that is not structured, and need to achieve *non-local* materialization. Another line of work that is slightly related to ours are the hybrid analyses that aim at discovering relations between the structures and their contents [5,16,3]. Our set predicates actually fit in the domain product formalized in [5], and can also indirectly capture through sets relations between structures and their contents. Abstractions of set properties have recently been used in order to capture relations between sets of keys of dictionaries [13,12] or groups of array cells [22]. A noticeable result is

that our analysis tracks very similar predicates, although for a radically different application.

8 Conclusion

In this paper, we have set up a shape analysis able to cope with unbounded sharing. This analysis combines separation logic based shape abstractions and a *set abstract domain*, that tracks pointer sharing properties. Reduction across domains is done lazily at non-local materialization and join. This abstraction was implemented into the MemCAD static analyzer and could cope with graphs described with adjacency lists. Future works will experiment with other set abstract domains and combine this abstraction with other memory abstractions [26,27].

Acknowledgments. We would like to thank Arlen Cox for suggestions about the implementation of the set domain, and François Bérenger for providing very helpful tool support. We also thank Tie Cheng, Antoine Toubhans and the anonymous referees for comments helping us improve this paper.

References

1. Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Conference on Computer Aided Verification (CAV)*, pages 178–192. Springer, 2007.
2. Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Symbolic execution with separation logic. In *Asian Conference on Programming Languages And Systems (APLAS)*, pages 52–68. Springer, 2005.
3. Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 1–22. Springer, 2012.
4. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Symposium on Principles of Programming Languages (POPL)*, pages 289–300. ACM, 2009.
5. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 247–260. ACM, 2008.
6. Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. *Festschrift for Dave Schmidt, ENTCS*, pages 161–185, 2013.
7. Bor-Yuh Evan Chang, Xavier Rival, and George Necula. Shape analysis with structural invariant checkers. In *Static Analysis Symposium (SAS)*. Springer, 2007.
8. Renato Cherini, Lucas Rearte, and Javier Blanco. A shape analysis for non-linear data structures. In *Static Analysis Symposium (SAS)*, pages 201–217. Springer, 2010.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, 1977.
10. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, 1979.

11. Arlen Cox. Binary-Decision-Diagrams for Set Abstraction. *ArXiv e-prints*, March 2015.
12. Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis Symposium (SAS)*, pages 134–150, 2014.
13. Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Symposium on Principles of Programming Languages (POPL)*, pages 187–200. ACM, 2011.
14. Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302. Springer, 2006.
15. Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Local shape analysis for overlaid data structures. In *Static Analysis Symposium (SAS)*, pages 150–171, 2013.
16. Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. Tval+: Tvla and value analyses together. In *Software Engineering and Formal Methods*, pages 63–77. Springer, 2012.
17. Jean-Christophe Filliatre. Bdd ocaml library. https://www.lri.fr/ filliatr/ftp/ocaml/bdd/.
18. Samin S. Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001.
19. Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. Shape analysis of low-level c with overlapping structures. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 214–230. Springer, 2010.
20. Oukseh Lee, Hongseok Yang, and Rasmus Petersen. Program analysis for overlaid data structures. In *Conference on Computer Aided Verification (CAV)*, pages 592–608, 2011.
21. Tal Lev-Ami and Mooly Sagiv. Tvla: A system for implementing static analyses. In *Static Analysis Symposium (SAS)*, pages 280–301. Springer, 2000.
22. Jiangchao Liu and Xavier Rival. Abstraction of arrays based on non contiguous partitions. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 282–299. Springer, 2015.
23. Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 251–266. Springer, 2007.
24. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics In Computer Science (LICS)*, pages 55–74. IEEE, 2002.
25. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
26. Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 375–395. Springer, 2013.
27. Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. An abstract domain combinator for separately conjoining memory abstractions. In *Static Analysis Symposium (SAS)*, pages 285–301. Springer, 2014.
28. Victor Vafeiadis. Shape-value abstraction for verifying linearizability. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–348. Springer, 2009.

29. Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Static Analysis Symposium (SAS)*, pages 366–382. Springer, 1996.

A Invariants of the running example

We show the invariants computed during the analysis of the random graph traversal program of Figure 1(e) in Figure 9 (invariants computed during the first iteration) and Figure 10 (invariants obtained after reaching an abstract post-fixpoint).

B Inductive definition set parameter kinds

In this section, we provide the derivation rules that allow to derive when a set parameter of an inductive definition is *head* or *constant*. These rules form the basis of a static analysis of inductive definitions, which infers the status of each set parameter.

Constant parameters. Let ι be an inductive definition defined by $\alpha \cdot \iota(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$ (for the sake of clarity, we assume ι has a single parameter). The following rules define the cases where $\mathscr{E}$ is said to be a *constant parameter*:

– parameter $\mathscr{E}$ is constant if and only if it is so for each heap part of a rule of ι:

$$\frac{\forall 0 \leq i \leq k,\ \ r_i \vdash_\iota \mathscr{E} : \mathbf{cst}}{\iota \vdash \mathscr{E} : \mathbf{cst}} \qquad \frac{F_{\text{Heap}} \vdash_\iota \mathscr{E} : \mathbf{cst}}{(F_{\text{Heap}}, F_{\text{Pure}}) \vdash_\iota \mathscr{E} : \mathbf{cst}}$$

– when considering a part of F_{Heap} that consists only of a call to ι, then the parameter is constant if and only if it is applied to the *same* set parameter as the current call:

$$\frac{}{\beta \cdot \iota(\mathscr{E}) \vdash_\iota \mathscr{E} : \mathbf{cst}}$$

– when considering a part of F_{Heap} that consists only of a call to another inductive definition ι', and provided that definition does not call ι (even indirectly), then the parameter is constant:

$$\frac{\iota' \text{ does not call } \iota \text{ directly or indirectly}}{\beta \cdot \iota'(\mathscr{F}) \vdash_\iota \mathscr{E} : \mathbf{cst}}$$

– finally, the property "is constant" propagates naturally through separating product and points-to predicate:

$$\frac{F_{\text{Heap}} \vdash_\iota \mathscr{E} : \mathbf{cst} \qquad F'_{\text{Heap}} \vdash_\iota \mathscr{E} : \mathbf{cst}}{F_{\text{Heap}} * F'_{\text{Heap}} \vdash_\iota \mathscr{E} : \mathbf{cst}} \qquad \frac{}{\alpha \cdot \mathtt{f} \mapsto \beta \vdash_\iota \mathscr{E} : \mathbf{cst}}$$

Let us assume that parameter $\mathscr{E}$ of inductive definition ι is *constant*, as defined by the rules shown above. The the following concretization inclusions hold:

$$\gamma_{\mathbb{G}}(\alpha_0 \cdot \iota *\!=_{(\mathscr{E})} \alpha_1 \cdot \iota * \alpha_1 \cdot \iota(\mathscr{E})) \subseteq \gamma_{\mathbb{G}}(\alpha_0 \cdot \iota(\mathscr{E}))$$
$$\gamma_{\mathbb{G}}(\alpha_0 \cdot \iota *\!=_{(\mathscr{E})} \alpha_1 \cdot \iota * \alpha_1 \cdot \iota *\!=_{(\mathscr{E})} \alpha_2 \cdot \iota) \subseteq \gamma_{\mathbb{G}}(\alpha_0 \cdot \iota *\!=_{(\mathscr{E})} \alpha_2 \cdot \iota)$$

These properties can be proved by structural induction on the unfolding based concretization of shapes, and then on the structure of inductive definitions.

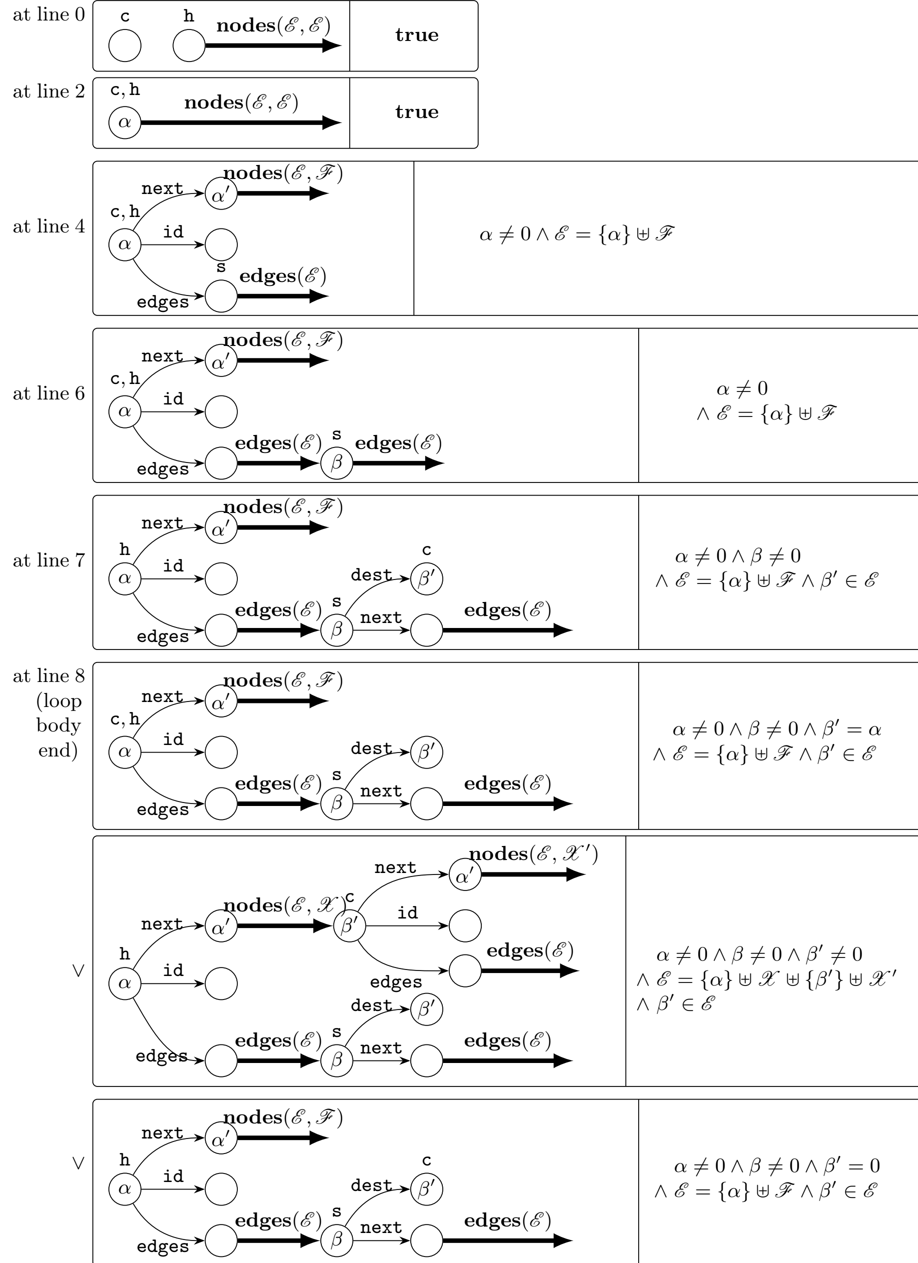


Fig. 9. Local invariants computed over the first iteration

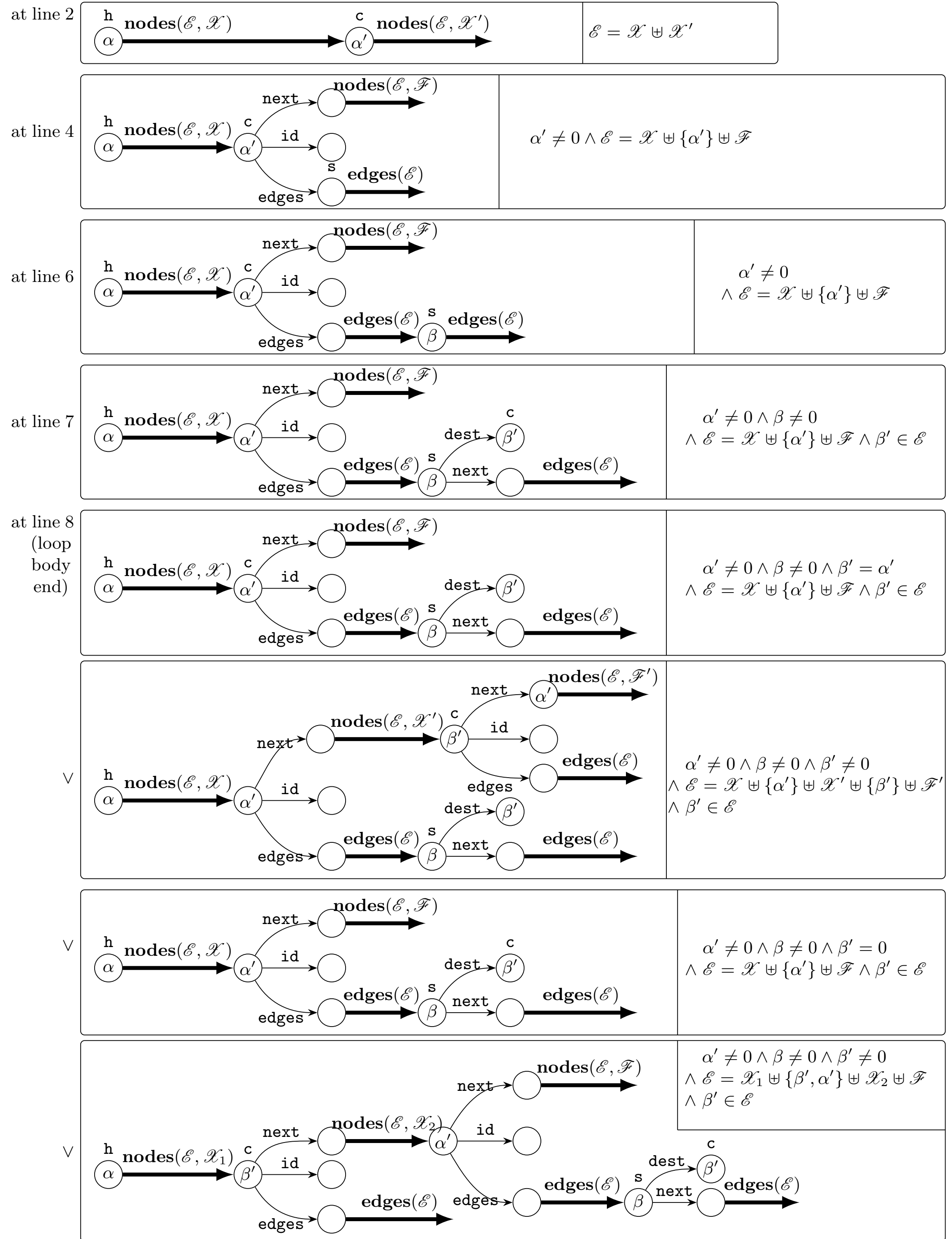


Fig. 10. Local invariants computed after reaching a post-fixpoint

Head parameters. We adopt similar notations as above, and assume ι is an inductive definition defined by $\alpha \cdot \iota(\mathscr{E}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$, with a single set parameter. The following rules define the cases where $\mathscr{E}$ is said to be a *head parameter*:

- parameter $\mathscr{E}$ is head for ι if and only if it is so for each inductive rule:

$$\frac{\forall 0 \leq i \leq k,\ \ r_i \vdash_\iota \mathscr{E} : \mathbf{head}}{\iota \vdash \mathscr{E} : \mathbf{head}}$$

- parameter $\mathscr{E}$ is head for a given rule if and only if it is provably equal to the disjoint union of the arguments of the recursive sub-calls to ι, that is if it can be partitioned into a set of sets corresponding to the argument of each call:

$$\frac{F_{\text{Heap}} \vdash_\iota \{\mathscr{E}_i \mid 0 \leq i \leq n\} \qquad F_{\text{Pure}} \vDash \biguplus_{0=i}^{n} \mathscr{E}_i \uplus \{\alpha\} = \mathscr{E}}{(F_{\text{Heap}}, F_{\text{Pure}}) \vdash_\iota \mathscr{E} : \mathbf{head}}$$

- rules for separating conjunction, empty and points-to predicates express the linearity of head parameters (where, T, T' denote sets of set variables):

$$\frac{F_{\text{Heap}} \vdash_\iota T \quad F'_{\text{Heap}} \vdash_\iota T'}{F_{\text{Heap}} * F'_{\text{Heap}} \vdash_\iota T \uplus T'} \qquad \frac{F_{\text{Pure}} \vDash \mathscr{E} = \emptyset}{(\mathbf{emp}, F_{\text{Pure}}) \vdash_\iota \mathscr{E} : \mathbf{head}} \qquad \frac{}{\alpha \cdot \mathtt{f} \mapsto \beta \vdash_\iota \emptyset}$$

- an inductive call to ι accounts for a single set variable corresponding to its parameter (the head addresses in the structure are exactly the head addresses of the sub-call):

$$\frac{}{\beta \cdot \iota(\mathscr{E}_i) \vdash_\iota \{\mathscr{E}_i\}}$$

- finally, calls to other inductive definitions contribute an empty set of head addresses in a structure:

$$\frac{\iota' \text{ does not call } \iota \text{ directly or indirectly}}{\beta \cdot \iota'(\mathscr{F}) \vdash_\iota \emptyset}$$

Let us assume that parameter $\mathscr{E}$ of inductive definition ι is *head*, as defined by the rules shown above. The the following implications hold:

$$\begin{array}{c} \forall (m, \nu) \in \gamma_{\mathbb{G}}(\alpha_0 \cdot \iota *\!\!=_{(\mathscr{E}_0)} \alpha_1 \cdot \iota * \alpha_1 \cdot \iota(\mathscr{E}_1)), \\ \qquad \nu \vdash \mathscr{E}_0 \uplus \mathscr{E}_1 \Longrightarrow (m, \nu) \in \gamma_{\mathbb{G}}(\alpha_0 \cdot \iota(\mathscr{E})) \\ \forall (m, \nu) \in \gamma_{\mathbb{G}}(\alpha_0 \cdot \iota *\!\!=_{(\mathscr{E}_0)} \alpha_1 \cdot \iota * \alpha_1 \cdot \iota *\!\!=_{(\mathscr{E}_1)} \alpha_2 \cdot \iota), \\ \qquad \nu \vdash \mathscr{E}_0 \uplus \mathscr{E}_1 \Longrightarrow (m, \nu) \in \gamma_{\mathbb{G}}(\alpha_0 \cdot \iota *\!\!=_{(\mathscr{E})} \alpha_2 \cdot \iota) \end{array}$$

These properties can be proved by structural induction on the unfolding based concretization of shapes, and then on the structure of inductive definitions.

C Analysis operations

In this section, we describe all the rules underlying the inclusion checking and join algorithms.

C.1 Inclusion checking rules

The inclusion checking algorithm is based on a set of logical rules, only a subset of which was described in the paper. For reference, the full list of logical rules used in the inclusion checking algorithm is provided below:

– To derive inclusion in the combined domain, inclusion checking needs to establish it in **both shape and set predicate abstract domains**:

$$\frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r \qquad S_l \sqsubseteq_{\mathbf{S}} S_r}{\vdash (G_l, S_l) \sqsubseteq_{\mathbf{G}} (G_r, S_r)}$$

– To handle **separating conjunction**, inclusion checking treats shapes region by region:

$$\frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r}{S_l \vdash G_l * G \sqsubseteq_{\mathbf{G}} G_r * G} \qquad \frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r}{S_l \vdash G * G_l \sqsubseteq_{\mathbf{G}} G * G_r}$$

– Inclusion checking derives inclusion holds when applied to two **identical graphs**:

$$\frac{}{S_l \vdash G \sqsubseteq_{\mathbf{G}} G}$$

– Inclusion checking may **unfold** any inductive predicate in its right argument:

$$\frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_u \qquad G_r \rightsquigarrow_{\mathrm{U}} (G_u, F_{\mathrm{Pure}}) \qquad \mathfrak{prove}_{\mathrm{S}}(S_l, F_{\mathrm{Pure}}) = \mathbf{true}}{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} G_r}$$

– When applied to a segment and an inductive predicate at the same origin corresponding to **a same inductive definition, with a same constant parameter**, inclusion checking can "consume" the segment in the left, and "reduce" the inductive predicate in the right:

$$\frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} \beta \cdot \iota(\mathscr{E}) \qquad \alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{cst}}{S_l \vdash \alpha \cdot \iota *\!\!=_{(\mathscr{E})} \beta \cdot \iota * G_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota(\mathscr{E})}$$

– Similarly, when applied to two segment predicates at the same origin, corresponding to **a same inductive definition, with a same constant parameter**, inclusion checking can "consume" the segment in the left, and "reduce" the segment predicate in the right:

$$\frac{S_l \vdash G_l \sqsubseteq_{\mathbf{G}} \beta \cdot \iota *\!\!=_{(\mathscr{E})} \delta \cdot \iota \qquad \alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{cst}}{S_l \vdash \alpha \cdot \iota *\!\!=_{(\mathscr{E})} \beta \cdot \iota * G_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota *\!\!=_{(\mathscr{E})} \delta \cdot \iota}$$

– When applied to a segment and an inductive predicate at the same origin corresponding to **a same inductive definition, with a head parameter**, and provided $\mathfrak{prove}_{\mathrm{S}}$ can establish inclusion of the right head parameter in the left one, inclusion checking can "consume" the segment in the left, "reduce" the inductive predicate in the right after synthesizing a *fresh* set parameter accounting for the set difference of the previous two set parameters:

$$\frac{\begin{array}{cc} \mathfrak{prove}_{\mathrm{S}}(S_l, \mathscr{E}_0 \subseteq \mathscr{E}) & \mathscr{E}_1 \text{ fresh (denotes } \mathscr{E} \setminus \mathscr{E}_0) \\ S_l \vdash G_l \sqsubseteq_{\mathbf{G}} \beta \cdot \iota(\mathscr{E}_1) & \alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \mathbf{head} \end{array}}{S_l \vdash \alpha \cdot \iota *\!\!=_{(\mathscr{E}_0)} \beta \cdot \iota * G_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota(\mathscr{E})}$$

– A similar rule applies when both arguments consist of a **segment predicate**, corresponding to the same inductive definition, and with **head parameter satisfying a similar inclusion condition**:

$$\frac{\mathfrak{prove}_{\mathrm{S}}(S_l, \mathscr{E}_0 \subseteq \mathscr{E}) \qquad \mathscr{E}_1 \text{ fresh (denotes } \mathscr{E} \setminus \mathscr{E}_0) \quad S_l \vdash G_l \sqsubseteq_{\mathbf{G}} \beta \cdot \iota \ast\!\!=_{(\mathscr{E}_1)} \delta \cdot \iota \qquad \alpha \cdot \iota(\mathscr{E}) \vdash \mathscr{E} : \textbf{head}}{S_l \vdash \alpha \cdot \iota \ast\!\!=_{(\mathscr{E}_0)} \beta \cdot \iota \ast G_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \delta \cdot \iota}$$

During the search for a proof of inclusion, the algorithm attempts to apply those rules in a particular order, so as to maximize chances of a successful result, without exploring the whole proof search space. Strategies are discussed in [5].

C.2 Join rules

The join algorithm relies on two generic rules, shown in Figure 8. We provide the full list of instances of these two rules below:

– The first rule applies when both inputs contain *syntactically equal* atomic predicates, and immediately return the same shape as a join:

$$\frac{G = \alpha \cdot \mathtt{f} \mapsto \beta \ \text{ or } \ G = \alpha \cdot \iota(\mathscr{E}) \ \text{ or } \ G = \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota}{[(G_l \ast G, S_l) \sqcup_{\mathbf{G}} (G_r \ast G, S_r) | G_o] \rightsquigarrow_{\sqcup} [(G_l, S_l) \sqcup_{\mathbf{G}} (G_r, S_r) | G_o \ast G]}$$

– The weakening rule below attempts to identify inductive predicates that subsumes regions of both input shapes:

$$\frac{S'_l \text{ (resp., } S'_r) \text{ extends } S_l \text{ (resp., } S_r) \qquad S'_l \vdash G'_l \sqsubseteq_{\mathbf{G}} G'_o \qquad S'_r \vdash G'_r \sqsubseteq_{\mathbf{G}} G'_o}{[(G_l \ast G'_l, S_l) \sqcup_{\mathbf{G}} (G_r \ast G'_r, S_r) | G_o] \rightsquigarrow_{\sqcup} [(G_l, S'_l) \sqcup_{\mathbf{G}} (G_r, S'_r) | G_o \ast G'_o]}$$

We note that this rule may *extend* the set predicates, as fresh set variables are introduced by the inclusion check algorithm. Nevertheless, this process is non-trivial, since the join algorithm needs to *infer* relations over the new set predicates, from the results of inclusion check.

The join algorithm attempts to perform this weakening in the following cases:

- when either input contains an inductive predicate that subsumes a region of the other input:

$$\begin{cases} G'_l = \alpha \cdot \iota(\mathscr{E}) \\ \wedge\ S'_r \vdash G'_r \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota(\mathscr{E}) \end{cases} \quad \text{or} \quad \begin{cases} G'_r = \alpha \cdot \iota(\mathscr{E}) \\ \wedge\ S'_l \vdash G'_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota(\mathscr{E}) \end{cases}$$

- when either input contains a region subsumed by a segment between α and β, and $\alpha = \beta$ in the other input (this rule effectively *introduces* an empty segment):

$$\begin{cases} \mathfrak{prove}_{\mathrm{S}}(S_l, \alpha = \beta) = \textbf{true} \\ \wedge\ S'_r \vdash G'_r \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota \\ \wedge\ S'_l = S_l \end{cases} \quad \text{or} \quad \begin{cases} \mathfrak{prove}_{\mathrm{S}}(S_r, \alpha = \beta) = \textbf{true} \\ \wedge\ S'_l \vdash G'_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota \\ \wedge\ S'_r = S_r \end{cases}$$

  Since set parameter $\mathscr{E}$ is fresh, the join algorithm needs to rely on its properties, and on the inclusion checking algorithm to discover what $\mathscr{E}$ actually represents:

  - if $\iota \vdash \mathscr{E}$: **cst**, no new constraint is needed;
  - if $\iota \vdash \mathscr{E}$: **head** and the empty segment is introduced in the left side, then $S'_l = S_l \wedge \mathscr{E} = \emptyset$ and $S'_r = S'_r \wedge \mathscr{E} = H$ where H is the set of nodes grabbed as head nodes in the inclusion check (the case of an introduction in the right side is symmetric).
- when either input contains a segment, and the other input contains a region that can be subsumed by a similar segment:

$$\begin{cases} \quad G_l = \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota \\ \wedge\, S'_r \vdash G'_r \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota \\ \wedge\, S'_l = S_l \end{cases} \quad \text{or} \quad \begin{cases} \quad G_r = \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota \\ \wedge\, S'_l \vdash G'_l \sqsubseteq_{\mathbf{G}} \alpha \cdot \iota \ast\!\!=_{(\mathscr{E})} \beta \cdot \iota \\ \wedge\, S'_r = S_r \end{cases}$$

  As in the previous case, constraints over $\mathscr{E}$ may be added after the inclusion checking (in particular, in the case of a head set parameter, these constraints should capture the linearity over the head set parameter).

Similarly to the inclusion checking, these rules need to be apply in a carefully chosen order, to avoid a loss of precision due to the algorithm getting "stuck" in a position where no rule applies. Strategies are discussed in [5].