A Relational Shape Abstract Domain

Hugo Illous[1,2], Matthieu Lemerre[1], and Xavier Rival[2]

[1] CEA, LIST, Software Reliability and Security Laboratory,
P.C. 174, Gif-sur-Yvette, 91191, France
[2] INRIA Paris/CNRS/École Normale Supérieure/PSL Research University

Abstract. Static analyses aim at inferring semantic properties of programs. While many analyses compute an over-approximation of reachable states, some analyses compute a description of the input-output relations of programs. In the case of numeric programs, several analyses have been proposed that utilize relational numerical abstract domains to describe relations. On the other hand, designing abstractions for relations over memory states and taking shapes into account is challenging. In this paper, we propose a set of novel logical connectives to describe such relations, which are inspired by separation logic. This logic can express that certain memory areas are unchanged, freshly allocated, or freed, or that only part of the memory was modified. Using these connectives, we build an abstract domain and design a static analysis that over-approximates relations over memory states containing inductive structures. We implement this analysis and report on the analysis of a basic library of list manipulating functions.

1 Introduction

Generally, static analyses aim at computing semantic properties of programs. Two common families of analyses are *reachability analyses*, that compute an over-approximation for the *set of reachable states* of programs, and *relational analyses*, that compute an over-approximation for the relations between input and output states. In general, sets of states are easier to abstract than state relations, which often makes reachability analyses simpler to design. On the other hand, abstracting relations brings several advantages:

– First, state relations allow to make the analyses modular [10,22,17,6,3] and compositional. Indeed, to analyze a sequence of two sub-programs, relational analyses can simply analyze each sub-program separately, and compose the resulting state relations. When sub-programs are functions, relational analyses may analyze each function separately, and compute one summary per function, so that the analysis of a function call does not require re-analyzing the body of the function, which is an advantage for scalability.

– Second, some properties can be expressed on state relations but not on sets of states, which makes relational analyses intrinsically more expressive. For example, contract languages [1,21] let functions be specified by formulas that may refer both to the input and to the output states. Such properties cannot be expressed using abstractions of sets of states, thus are beyond the scope of reachability analyses.

In general, the increased expressiveness of relational analyses requires more expressive abstractions. Let us discuss, as an example the case of numeric programs. A common way to express relations between input and output states consists in defining for each variable $\mathtt{x}$ a primed version $\mathtt{x}'$ that describes the value of $\mathtt{x}$ in the output state whereas the non primed version denotes the value of $\mathtt{x}$ in the input state. In this context, non-relational numerical abstract domain such as intervals [8] cannot capture any interesting relation between input and output states. On the other hand, relational numerical abstract domains such as convex polyhedra [7] can effectively capture relations between input and output states, as shown in [22]: for instance, when applied to a program that increments $\mathtt{x}$ by one, this analysis can infer the relation $\mathtt{x}' = \mathtt{x} + 1$.

In the context of programs manipulating complex data structures, relational analysis could allow to compute interesting classes of program properties. For instance, such analyses could express and verify that some memory areas were not physically modified by a program. Reachability analyses such as [24,15,5] cannot distinguish a program that inputs a list and leaves it unmodified from a program that inputs a list, copies it into an identical version and deallocates it, whereas a relational analysis could. More generally, it is often interesting to infer that a memory region is not modified by a program.

Separation logic [23] provides an elegant description for sets of states and is at the foundation of many reachability analyses for heap properties. In particular, the separating conjunction connective $*$ expresses that two regions are disjoint and allows local reasoning. On the other hand, it cannot describe state relations.

In this paper, we propose a logic inspired by separation logics and that can describe such properties. It provides connectives to describe that a memory region has been left unmodified by a program fragment, or that memory states can be split into disjoint sub-regions that undergo different transformations. We build an abstract domain upon this logic, and apply it to design an analysis for programs manipulating simple list or tree data structures. We make the following contributions:

- In Section 2, we demonstrate the abstraction of state relations using a specific family of heap predicates;
- In Section 4, we set up a logic to describe heap state relations and lift it into an abstract domain that describe concrete relations defined in Section 3;
- In Section 5, we design static analysis algorithms to infer heap state relations from abstract pre-condition;
- In Section 6, we report on experiments on basic linked data structures (lists and trees);
- Finally, we discuss related works in Section 7 and conclude in Section 8.

2 Overview and Motivating Example

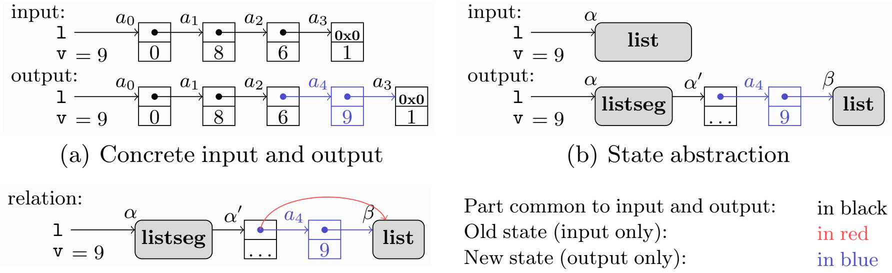
We consider the example code shown in Figure 1, which implements the insertion of an element inside a non empty singly linked list containing integer values. When applied to a pointer to an existing non empty list and an integer value, this function

```
typedef struct list { struct list * next; int data; } list;
void insert_non_empty( list *l, int v ){
  assume(l != NULL); list *c = l;
  while( c->next != NULL && ... ){
    c = c->next;
  }
  list *e = new( {next, data} ); // allocate 2 fields block
  e->next = c->next; c->next = e; e->data = v;
}
```

Fig. 1. A list insertion program

traverses it partially (based on a condition on the values stored in list elements —that is elided in the figure). It then allocates a new list element, inserts it at the selected position and copies the integer argument into the `data` field. For instance, Figure 2(a) shows an input list containing elements $0, 8, 6, 1$ and an output list where value 9 is inserted as a new element in the list. We observe that all elements of the input list are left physically unmodified except the element right before the insertion point. We now discuss abstractions of the behaviors of this program using abstractions for sets of states and abstractions for state relations.

Reachability analysis. First, we consider an abstraction based on separation logics with inductive predicates as used in [15,5]. We assume that the predicate **list**(α) describes heap regions that consist of a well-formed linked list starting at address α (α is a symbolic variable used in the abstraction to denote a concrete address). This predicate is intuitively defined by induction as follows: it means either the region is empty and α is the null pointer, or the region is not empty, and consists of a list element of address α and with a `next` field containing a value described by symbolic variable β and a region that can be described by **list**(β). Thus, the valid input states for the insertion function can be abstracted by the abstract state shown in the top of Figure 2(b). The analysis of the function needs to express that the insertion occurs somewhere in the middle of the list. This requires a list segment predicate **listseg**(α, α'), that is defined in a similar way as for **list**: it describes region that stores a sub list starting at address α and the last element of which has a `next` field pointing to address α' (note that the empty region can be described by **listseg**(α, α)). Using this predicate, we can now also express an abstraction for the output states of the insertion function: the abstract state shown in the bottom of Figure 2(b) describes the states where the new element was inserted in the middle of the structure (the list starts with a segment, then the predecessor of the inserted element, then the inserted element, and finally the list tail). We observe that this abstraction allows to express and to verify that the function is memory safe, and returns a well-formed list. Indeed, it captures the fact that no null or dangling pointer is ever dereferenced. Moreover, all states described by the abstract post-condition consist of a well-formed list, made of a segment, followed by two elements and a list tail. On the other hand, it does not say anything about


input:
l
v = 9
a_0
a_1
a_2
a_3
0
8
6
0x0
1
output:
a_4
9
α
α′
β
list
listseg
. . .
relation:
Part common to input and output: in black
Old state (input only): in red
New state (output only): in blue

(a) Concrete input and output

(b) State abstraction

(c) Relational abstraction

Fig. 2. Abstractions

the location of the list in the output state with respect to the list in the input state. More precisely, it cannot capture the fact that the elements of addresses a_0, a_1, a_3 are left unmodified physically. This is a consequence of the fact that each abstract state in Figure 2(b) independently describes a set of concrete heaps.

Relational analysis. To abstract *state relations* instead of sets of states, we now propose to define a new structure in Figure 2(c), that partially overlays the abstractions of input and output states. First, we observe that the tail of the list is not modified at all, thus, we describe it with a single predicate $\mathtt{Id}(\mathbf{list}(\beta))$, that denotes pairs made of input state and an output state, that are *physically equal* and can both be described by $\mathbf{list}(\beta)$. The same kind of predicate can be used to describe that the initial segment has not changed between the two states. Second, we need to define a counterpart for separating conjunction at the relation level. Indeed, the effect of the insertion function can be decomposed as its effect on the initial segment (which is left unchanged), its effect on the tail (which is also left unchanged) and its effect on the insertion point (where a new element is allocated and a `next` pointer is modified). This relation separating conjunction is noted $*_{\mathtt{R}}$. To avoid confusion, from now on, we write $*_{\mathtt{S}}$ for the usual separating conjunction. Last, the insertion function allocates a new element and modifies the value of the `next` field of an existing element. To account for this, we need a new connective $[\cdot \dashrightarrow \cdot]$ which is applied to two abstract states: if $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1$ are abstract heaps (described by formulas in the usual separation logic with inductive predicates), then $[\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1]$ describes the transformation of an input state described by $\mathrm{h}^\sharp_0$ into an output state described by $\mathrm{h}^\sharp_1$. This is presented with different colors in the figure. In Section 4, we formalize this logics and the abstraction that it defines. The analysis by forward abstract interpretation [8] starts with the identity relation at function entry, and computes relations between input and output states step by step. The analysis algorithms need to unfold inductive predicates to materialize cells (for instance to analyze the test at line 4), and to fold inductive predicates in order to analyze loops. In addition to this, it also needs to reason over $\mathtt{Id}$, $[\cdot \dashrightarrow \cdot]$ and $*_{\mathtt{R}}$ predicates, and perform operations similar to unfolding and folding on them. Section 5 describes the analysis algorithms.

3 Concrete Semantics

Before defining the abstraction, we fix notations for concrete states and programs.

We let $\mathbb{X}$ denote the set of program variables and $\mathbb{V}$ denote the set of values (that includes the set of numeric addresses). A field $\in \mathbb{F}$ (noted as $\texttt{next}, \texttt{data}, \ldots$) denotes both field names and offsets. A memory state $\sigma \in \mathbb{M}$ is a partial function from addresses to values. We write $\mathbf{dom}(\sigma)$ for the domain of σ, that is the set of addresses for which it is defined. Additionally, if σ_0, σ_1 are such that $\mathbf{dom}(\sigma_0) \cap \mathbf{dom}(\sigma_1) = \emptyset$, we let $\sigma_0 \circledast \sigma_1$ be the memory state obtained by merging σ_0 and σ_1 (its domain is $\mathbf{dom}(\sigma_0) \cup \mathbf{dom}(\sigma_1)$). If a_i is an address and v_i a value, we write $[a_0 \mapsto v_0; \ldots; a_n \mapsto v_n]$ the memory state where a_i contains v_i (with $0 \leq i \leq n$).

In the following, we consider simple imperative programs, that include basic assignments, allocation and deallocation statements and loops (although our analysis supports a larger language, notably with conditionals and unstructured control flow). Programs are described by the grammar below:

$$\begin{array}{lll}
\texttt{L} ::= \texttt{x} \quad (\texttt{x} \in \mathbb{X}) \mid \texttt{L}\,\texttt{->}\,\texttt{f} \quad (\texttt{f} \in \mathbb{F}) & & \text{l-values} \\
\texttt{E} ::= v \quad (v \in \mathbb{V}) \mid \texttt{L} \mid \texttt{E} \oplus \texttt{E} \quad (\oplus \in \{+, -, \leq, \ldots\}) & & \text{expressions} \\
\texttt{P} ::= \texttt{L} = \texttt{E}; \mid \texttt{L} = \mathbf{new}(\{\texttt{f}_0, \ldots\}); \mid \mathbf{free}(\texttt{L}); \mid \texttt{P}; \texttt{P} \mid \mathbf{while}(\texttt{E})\texttt{P} & & \text{programs}
\end{array}$$

We assume the semantics of a program $\texttt{P}$ is defined as a function $[\![\texttt{P}]\!]$ that maps a set of input states into a set of output states (thus $[\![\texttt{P}]\!] : \mathcal{P}(\mathbb{M}) \longrightarrow \mathcal{P}(\mathbb{M})$). We do not provide a full formal definition for $[\![\texttt{P}]\!]$ as it is classical. Given a program $\texttt{P}$, we define its *relational semantics* $[\![\texttt{P}]\!]_{\mathcal{R}} : \mathbb{M} \rightarrow \mathbb{M} \times \mathbb{M}$ by:

$$\forall M \subseteq \mathbb{M},\ [\![\texttt{P}]\!]_{\mathcal{R}}(M) = \{(\sigma_0, \sigma_1) \mid \sigma_0 \in M \wedge \sigma_1 \in [\![\texttt{P}]\!](\{\sigma_0\})\}$$

In the following, we define an analysis to compute an over-approximation for $[\![\texttt{P}]\!]_{\mathcal{R}}$.

4 Abstraction

In this section, we first define *abstract states*, that describe sets of memory states (as in [5]), and then we set up *abstract state relations*, that describe binary relations over memory states. Although our analysis and implementation support more general inductive predicates (such as trees and others), we consider only list inductive predicates in the body of the paper, for the sake of simplicity.

Abstract states. We assume a countable set $\mathbb{A} = \{\alpha, \beta, \ldots\}$ of *symbolic addresses* that abstract values and heap addresses. An abstract state $\sigma^\sharp$ consists of an abstract heap $\mathrm{h}^\sharp$ with a conjunction of numerical constraints such as equalities and disequalities. An abstract heap is a separating conjunction of region predicates that abstract *separate* memory regions [23] (as mentioned above, separating conjunction is denoted by $*_{\mathrm{S}}$). A node $\mathrm{n} \in \mathbb{N}$ is either a variable address $\&\texttt{x}$ or a symbolic address α. A region predicate is either **emp** describing an empty region, or a points-to predicate $\mathrm{n} \cdot \texttt{f} \mapsto \mathrm{n}'$ (that describes a heap memory cell at the base address n with the possibly null offset $\texttt{f}$ and with the content n'), or a summary predicate $\mathbf{list}(\mathrm{n})$ describing a list structure or $\mathbf{listseg}(\mathrm{n}, \mathrm{n}')$ for a (possibly empty) list segment from address n to n'. The **list** predicate is defined by induction as follows:

$$\begin{aligned}\mathbf{list}(\mathrm{n}) ::= {} & \mathbf{emp} \wedge \mathrm{n} = \mathbf{0x0} \\ \vee {} & \mathrm{n} \cdot \mathtt{next} \mapsto \alpha_n *_{\mathtt{S}} \mathrm{n} \cdot \mathtt{data} \mapsto \alpha_d *_{\mathtt{S}} \mathbf{list}(\alpha_n) \wedge \mathrm{n} \neq \mathbf{0x0}\end{aligned}$$

Segment predicate **listseg** stands for the segment version of **list** and describes a list without a tail; it can also be defined by induction. We write $\xrightarrow{\mathrm{unfold}}$ for the unfolding relation that syntactically transforms an instance of an inductive predicate into any of the disjuncts of that predicate.

Definition 1 (Abstract state). *Abstract heaps and abstract states are defined by the grammar below:*

$$\begin{array}{lll}\mathrm{c}^\sharp & ::= \mathrm{n} \odot \mathbf{0x0} \quad (\odot \in \{=, \neq\}) \;\mid\; \mathrm{n} = \mathrm{n}' \;\mid\; \mathrm{c}^\sharp \wedge \mathrm{c}^\sharp & \\ \mathrm{h}^\sharp (\in \mathbb{H}) & ::= \mathbf{emp} \mid \mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}' \mid \mathbf{list}(\mathrm{n}) \mid \mathbf{listseg}(\mathrm{n}, \mathrm{n}') \mid \mathrm{h}^\sharp *_{\mathtt{S}} \mathrm{h}^\sharp & \\ \sigma^\sharp (\in \varSigma) & ::= \mathrm{h}^\sharp \wedge \mathrm{c}^\sharp & \mathrm{n} (\in \mathbb{N}) ::= \alpha \quad (\alpha \in \mathbb{A}) \;\mid\; \&\mathtt{x} \quad (\mathtt{x} \in \mathbb{X})\end{array}$$

We now define the meaning of abstract heaps and abstract states using *concretization functions* [8], that associate to abstract elements the set of concrete elements they describe. To concretize an abstract heap, we also need to define how the nodes are bound into concrete values in concrete memories. We call *valuation* a function ν that maps nodes into concrete values and addresses.

Definition 2 (Concretization of abstract states). *The concretization function $\gamma_{\mathbb{C}}$ maps a numeric constraint into a set of valuations whereas $\gamma_{\mathbb{H}}$ and $\gamma_{\varSigma}$ respectively map an abstract heap and an abstract state into a set of pairs made of memory state and a valuation. They are defined by induction as follows:*

$$\begin{array}{l}\gamma_{\mathbb{C}}(\mathrm{n} \odot \mathbf{0x0}) = \{\nu \mid \nu(\mathrm{n}) \odot \mathbf{0x0}\} \\ \gamma_{\mathbb{C}}(\mathrm{n} = \mathrm{n}') = \{\nu \mid \nu(\mathrm{n}) = \nu(\mathrm{n}')\} \qquad \gamma_{\mathbb{C}}(\mathrm{c}_0^\sharp \wedge \mathrm{c}_1^\sharp) = \gamma_{\mathbb{C}}(\mathrm{c}_0^\sharp) \cap \gamma_{\mathbb{C}}(\mathrm{c}_1^\sharp) \\ \gamma_{\mathbb{H}}(\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}') = \{[\nu(\mathrm{n}) + \mathtt{f} \mapsto \nu(\mathrm{n}')], \nu)\} \qquad \gamma_{\mathbb{H}}(\mathbf{emp}) = \{([], \nu)\} \\ \gamma_{\mathbb{H}}(\mathbf{ind}) = \bigcup \{\gamma_{\varSigma}(\sigma^\sharp) \mid \mathbf{ind} \xrightarrow{\mathit{unfold}} \sigma^\sharp\} \quad \mathit{if}\ \mathbf{ind}\ \mathit{is}\ \mathbf{list}(\mathrm{n})\ \mathit{or}\ \mathbf{listseg}(\mathrm{n}, \mathrm{n}') \\ \gamma_{\mathbb{H}}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp) = \{(\sigma_0 \circledast \sigma_1, \nu) \mid (\sigma_0, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_0^\sharp) \wedge (\sigma_1, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_1^\sharp)\} \\ \gamma_{\varSigma}(\mathrm{h}^\sharp \wedge \mathrm{c}^\sharp) = \{(\sigma, \nu) \mid (\sigma, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}^\sharp) \wedge \nu \in \gamma_{\mathbb{C}}(\mathrm{c}^\sharp)\}\end{array}$$

Example 1 (Abstract state). The abstract pre-condition of the program of Figure 1 is $\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \mathbf{list}(\alpha) *_{\mathtt{S}} \&\mathtt{v} \mapsto \beta$.

Abstract relations. An *abstract heap relation* describes a set of pairs made of an *input* memory state σ_{i} and an *output* memory state σ_{o}. Abstract heap relations are defined by the following connectives:

- the *identity relation* $\mathtt{Id}(\mathrm{h}^\sharp)$ describes pairs of memory states that are equal and are both abstracted by $\mathrm{h}^\sharp$; this corresponds to the identity transformation;
- the *transformation relation* $[\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]$ describes pairs corresponding to the transformation of a memory state abstracted by $\mathrm{h}_{\mathrm{i}}^\sharp$ into a memory state abstracted by $\mathrm{h}_{\mathrm{o}}^\sharp$;

- the *relation separating conjunction* $\mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$ of two heap relations $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp$ denotes a transformation that can be described by combining independently the transformations described by $\mathrm{r}_0^\sharp$ and $\mathrm{r}_1^\sharp$ on disjoint memory regions.

Definition 3 (Abstract relations). *The syntax of* abstract heap relations *and* abstract state relations *are defined by the grammar below:*

$$\mathrm{r}^\sharp (\in \mathbb{R}) ::= \mathtt{Id}(\mathrm{h}^\sharp) \mid [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp] \mid \mathrm{r}^\sharp *_{\mathtt{R}} \mathrm{r}^\sharp \qquad\qquad \rho^\sharp (\in \mathit{\Pi}) ::= \mathrm{r}^\sharp \wedge \mathrm{c}^\sharp$$

The concretization of relations also requires using valuations as it also needs to define the concrete values that nodes denote. It thus returns triples made of two memory states and a valuation.

Definition 4 (Concretization of abstract relations). *The concretization functions $\gamma_{\mathbb{R}}, \gamma_{\mathit{\Pi}}$ respectively map an abstract heap relation and an abstract state relation into elements of $\mathbb{M} \times \mathbb{M} \times (\mathbb{N} \longrightarrow \mathbb{V})$. They are defined by:*

$$\begin{aligned}
\gamma_{\mathbb{R}}(\mathtt{Id}(\mathrm{h}^\sharp)) &= \{(\sigma, \sigma, \nu) \mid (\sigma, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}^\sharp)\} \\
\gamma_{\mathbb{R}}([\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]) &= \{(\sigma_{\mathrm{i}}, \sigma_{\mathrm{o}}, \nu) \mid (\sigma_{\mathrm{i}}, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_{\mathrm{i}}^\sharp) \wedge (\sigma_{\mathrm{o}}, \nu) \in \gamma_{\mathbb{H}}(\mathrm{h}_{\mathrm{o}}^\sharp)\} \\
\gamma_{\mathbb{R}}(\mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp) &= \{(\sigma_{\mathrm{i},0} \circledast \sigma_{\mathrm{i},1}, \sigma_{\mathrm{o},0} \circledast \sigma_{\mathrm{o},1}, \nu) \mid \\
&\qquad\qquad (\sigma_{\mathrm{i},0}, \sigma_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{R}}(\mathrm{r}_0^\sharp) \wedge \mathbf{dom}(\sigma_{\mathrm{i},0}) \cap \mathbf{dom}(\sigma_{\mathrm{o},1}) = \emptyset \\
&\qquad\quad \wedge (\sigma_{\mathrm{i},1}, \sigma_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{R}}(\mathrm{r}_1^\sharp) \wedge \mathbf{dom}(\sigma_{\mathrm{i},1}) \cap \mathbf{dom}(\sigma_{\mathrm{o},0}) = \emptyset\} \\
\gamma_{\mathit{\Pi}}(\mathrm{r}^\sharp \wedge \mathrm{c}^\sharp) &= \{(\sigma_{\mathrm{i}}, \sigma_{\mathrm{o}}, \nu) \mid (\sigma_{\mathrm{i}}, \sigma_{\mathrm{o}}, \nu) \in \gamma_{\mathbb{R}}(\mathrm{r}^\sharp) \wedge \nu \in \gamma_{\mathbb{C}}(\mathrm{c}^\sharp)\}
\end{aligned}$$

We remark that $*_{\mathtt{R}}$ is commutative and associative.

Example 2 (Expressiveness). Let $\mathrm{r}_0^\sharp = \mathtt{Id}(\mathbf{list}(\mathrm{n}))$ and $\mathrm{r}_1^\sharp = [\mathbf{list}(\mathrm{n}) \dashrightarrow \mathbf{list}(\mathrm{n})]$. We observe that $\mathrm{r}_0^\sharp$ describes only the identity transformation applied to a precondition where n is the address of a well-formed list, whereas $\mathrm{r}_1^\sharp$ describes any transformation that inputs such a list and also outputs such a list, but may modify its content, add or remove elements, or may modify the order of list elements (except for the first one which remains at address n). This means that $\gamma_{\mathbb{R}}(\mathrm{r}_0^\sharp) \subset \gamma_{\mathbb{R}}(\mathrm{r}_1^\sharp)$.

More generally, we have the following properties:

Theorem 1 (Properties). *Let $\mathrm{h}^\sharp, \mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp, \mathrm{h}_{\mathrm{i},0}^\sharp, \mathrm{h}_{\mathrm{i},1}^\sharp, \mathrm{h}_{\mathrm{o},0}^\sharp, \mathrm{h}_{\mathrm{o},1}^\sharp$ be abstract heaps. Then, we have the following properties*

1. $\gamma_{\mathbb{R}}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$
2. $\gamma_{\mathbb{R}}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$ *(the opposite inclusion may not hold, as observed in Example 2);*
3. $\gamma_{\mathbb{R}}([\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp] *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]) \subseteq \gamma_{\mathbb{R}}([(\mathrm{h}_{\mathrm{i},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{i},1}^\sharp) \dashrightarrow (\mathrm{h}_{\mathrm{o},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{o},1}^\sharp)])$ *(the opposite inclusion may not hold).*

Example 3 (Abstract state relation). The effect of the insertion function of Figure 1 can be described by the abstract state relation $\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} [\mathrm{h}_1^\sharp \dashrightarrow \mathrm{h}_2^\sharp] *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \mathrm{h}_3^\sharp]$, where $\mathrm{h}_0^\sharp = \&\mathtt{l} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{v} \mapsto \beta *_{\mathtt{S}} \mathbf{listseg}(\alpha_0, \alpha_1) *_{\mathtt{S}} \mathbf{list}(\alpha_2) *_{\mathtt{S}} \alpha_1 \cdot \mathtt{data} \mapsto \beta_2)$ (preserved region), $\mathrm{h}_1^\sharp = \alpha_1 \cdot \mathtt{next} \mapsto \alpha_2$, $\mathrm{h}_2^\sharp = \alpha_1 \cdot \mathtt{next} \mapsto \alpha$ (modified region) and $\mathrm{h}_3^\sharp = \alpha \cdot \mathtt{next} \mapsto \alpha_2 *_{\mathtt{S}} \alpha \cdot \mathtt{data} \mapsto \beta$ (new region).

5 Analysis Algorithms

We now propose a static analysis to compute abstract state relations as described in Definition 3. It proceeds by forward abstract interpretation [8], starting from the abstract relation $\mathtt{Id}(\mathrm{h}^\sharp)$ where $\mathrm{h}^\sharp$ is a pre-condition, supplied by the user.

More generally, the analysis of a program $\mathtt{P}$ is a function $\llbracket \mathtt{P} \rrbracket^\sharp_{\mathcal{R}}$ that inputs an abstract state relation describing a previous transformation $\mathcal{T}$ done on the input *before* running $\mathtt{P}$ and returns a relation describing that transformation $\mathcal{T}$ followed by the execution of $\mathtt{P}$. Thus, $\llbracket \mathtt{P} \rrbracket^\sharp_{\mathcal{R}}$ should meet the following soundness condition:

$$\begin{aligned}&\forall \rho^\sharp \in \Pi,\ \forall (\sigma_0, \sigma_1) \in \gamma_\Pi(\rho^\sharp),\ \forall \sigma_2 \in \mathbb{M},\\ &\qquad (\sigma_1, \sigma_2) \in \llbracket \mathtt{P} \rrbracket_{\mathcal{R}} \Longrightarrow (\sigma_0, \sigma_2) \in \gamma_\Pi(\llbracket \mathtt{P} \rrbracket^\sharp_{\mathcal{R}}(\rho^\sharp))\end{aligned}$$

5.1 Basic abstract post-conditions

We start with the computation of abstract post-condition for assignments, allocation and deallocation, on abstract relations that do not contain inductive predicates. As an example, we consider the analysis of an assignment $\mathtt{L} = \mathtt{E}$, starting from an abstract pre-condition relation $\mathrm{r}^\sharp$. To compute the effect of this assignment on $\mathrm{r}^\sharp$, the analysis should update it so as to reflect the modification of $\mathtt{L}$ in the output states of the pairs denoted by $\mathrm{r}^\sharp$. We first consider the case where $\mathrm{r}^\sharp$ is a transformation relation.

Case of a transformation relation. We assume $\mathrm{r}^\sharp = [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1]$. Then, if $\mathrm{h}^\sharp_2$ is an abstract state that describes the memory states after the assignment $\mathtt{L} = \mathtt{E}$, when it is executed on a state that is in $\gamma_{\mathbb{H}}(\mathrm{h}^\sharp_1)$, then a valid definition for $\llbracket \mathtt{L} = \mathtt{E} \rrbracket^\sharp_{\mathcal{R}}(\mathrm{r}^\sharp)$ is $[\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_2]$. An algorithm for computing such a $\mathrm{h}^\sharp_2$ can be found in [5]. It first evaluates $\mathtt{L}$ into a points-to predicate $\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}'$ describing the cell that $\mathtt{L}$ represents, then evaluates $\mathtt{E}$ into a node n'' describing the value of the right hand side and finally replaces $\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}'$ with $\mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}''$. As a consequence, we have the following definitions for the two main cases of assignments :

$$\begin{aligned}&\llbracket \mathtt{x} = \mathtt{y}\ \mathtt{->}\ \mathtt{f} \rrbracket^\sharp_{\mathcal{R}}([\mathrm{h}^\sharp_0 \dashrightarrow (\mathrm{h}^\sharp_1 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{f} \mapsto \alpha_2)])\\ &\quad = [\mathrm{h}^\sharp_0 \dashrightarrow (\mathrm{h}^\sharp_1 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_2 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{f} \mapsto \alpha_2)]\\ &\llbracket \mathtt{x}\ \mathtt{->}\ \mathtt{f} = \mathtt{y} \rrbracket^\sharp_{\mathcal{R}}([\mathrm{h}^\sharp_0 \dashrightarrow (\mathrm{h}^\sharp_1 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \cdot \mathtt{f} \mapsto \alpha_1 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_2)])\\ &\quad = [\mathrm{h}^\sharp_0 \dashrightarrow (\mathrm{h}^\sharp_1 *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \cdot \mathtt{f} \mapsto \alpha_2 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_2)]\end{aligned}$$

Case of a separating conjunction relation. We now assume that $\mathrm{r}^\sharp = \mathrm{r}^\sharp_0 *_{\mathtt{R}} \mathrm{r}^\sharp_1$. If the assignment can be fully analyzed on $\mathrm{r}^\sharp_0$ (i.e., it does not read or modify $\mathrm{r}^\sharp_1$), then the following definition provides a sound transfer function, that relies on the same principle as the Frame rule [23] for separation logic:

$$\text{if } \llbracket \mathtt{L} = \mathtt{E} \rrbracket^\sharp_{\mathcal{R}}(\mathrm{r}^\sharp_0) \text{ is defined, then } \llbracket \mathtt{L} = \mathtt{E} \rrbracket^\sharp_{\mathcal{R}}(\mathrm{r}^\sharp_0 *_{\mathtt{R}} \mathrm{r}^\sharp_1) = \llbracket \mathtt{L} = \mathtt{E} \rrbracket^\sharp_{\mathcal{R}}(\mathrm{r}^\sharp_0) *_{\mathtt{R}} \mathrm{r}^\sharp_1$$

When $\mathtt{L} = \mathtt{E}$ writes in $\mathrm{r}^\sharp_0$ and reads in $\mathrm{r}^\sharp_1$, we get a similar definition as above. For instance:

$$\begin{aligned}&\llbracket \mathtt{x} = \mathtt{y}\ \mathtt{->}\ \mathtt{f} \rrbracket^\sharp_{\mathcal{R}}([\mathrm{h}^\sharp_0 \dashrightarrow (\&\mathtt{x} \mapsto \alpha_0)] *_{\mathtt{R}} [\mathrm{h}^\sharp_1 \dashrightarrow (\&\mathtt{y} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{f} \mapsto \alpha_2)])\\ &\quad [\mathrm{h}^\sharp_0 \dashrightarrow (\&\mathtt{x} \mapsto \alpha_2)] *_{\mathtt{R}} [\mathrm{h}^\sharp_1 \dashrightarrow (\&\mathtt{y} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{f} \mapsto \alpha_2)]\end{aligned}$$

Case of an identity relation. We now assume that $\mathrm{r}^\sharp = \mathtt{Id}(\mathrm{h}^\sharp)$. As observed in Theorem 1, $\gamma_{\Pi}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\Pi}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$. We derive from the previous two paragraphs and from this principle the following definitions:

$$
\begin{aligned}
&\llbracket \mathtt{x} = \mathtt{y}\,\mathtt{->}\,\mathtt{f} \rrbracket^\sharp_{\mathcal{R}}(\mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{f} \mapsto \alpha_2))\\
&\quad = \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{f} \mapsto \alpha_2) *_{\mathtt{R}} [(\&\mathtt{x} \mapsto \alpha_0) \dashrightarrow (\&\mathtt{x} \mapsto \alpha_2)]\\
&\llbracket \mathtt{x}\,\mathtt{->}\,\mathtt{f} = \mathtt{y} \rrbracket^\sharp_{\mathcal{R}}(\mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \alpha_0 \cdot \mathtt{f} \mapsto \alpha_1 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_2))\\
&\quad = \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \&\mathtt{x} \mapsto \alpha_0 *_{\mathtt{S}} \&\mathtt{y} \mapsto \alpha_2) *_{\mathtt{R}} [(\alpha_0 \cdot \mathtt{f} \mapsto \alpha_1) \dashrightarrow (\alpha_0 \cdot \mathtt{f} \mapsto \alpha_2)]
\end{aligned}
$$

Other transfer functions. Condition tests boil down to numeric constraints intersections. The analysis of allocation needs to account for the creation of cells in the right side of relations whereas deallocation needs to account for the deletion of cells that were present before. Thus, for instance:

$$
\begin{aligned}
&\llbracket \mathtt{x} = \mathbf{new}(\{\mathtt{f}_0, \ldots, \mathtt{f}_n\}) \rrbracket^\sharp_{\mathcal{R}}(\mathrm{r}^\sharp *_{\mathtt{R}} [\mathrm{h}^\sharp \dashrightarrow (\&\mathtt{x} \mapsto \alpha)])\\
&\quad = \mathrm{r}^\sharp *_{\mathtt{R}} [\mathrm{h}^\sharp \dashrightarrow (\&\mathtt{x} \mapsto \beta)] *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{f}_0 \mapsto \beta_0 *_{\mathtt{S}} \ldots *_{\mathtt{S}} \beta \cdot \mathtt{f}_n \mapsto \beta_n)]\\
&\qquad\qquad \text{where } \beta, \beta_0, \ldots, \beta_n \text{ are fresh}\\
&\llbracket \mathbf{free}(\mathtt{x}) \rrbracket^\sharp_{\mathcal{R}}(\mathrm{r}^\sharp *_{\mathtt{R}} \mathtt{Id}(\&\mathtt{x} \mapsto \alpha *_{\mathtt{S}} \alpha \cdot \mathtt{f}_0 \mapsto \alpha_0) *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow (\mathrm{h}^\sharp_{\mathrm{o}} *_{\mathtt{S}} \alpha \cdot \mathtt{f}_1 \mapsto \alpha_1)])\\
&\quad = \mathrm{r}^\sharp *_{\mathtt{R}} \mathtt{Id}(\&\mathtt{x} \mapsto \alpha) *_{\mathtt{R}} [(\alpha \cdot \mathtt{f}_0 \mapsto \alpha_0) \dashrightarrow \mathbf{emp}] *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]
\end{aligned}
$$

5.2 Materialization and general abstract post-conditions

In Section 5.1, we considered only abstract states without inductive predicates, to first provide a simpler definition of abstract post-conditions. We now lift this restriction. For example, the analysis of the program in Figure 1 starts with $\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \mathbf{list}(\alpha) *_{\mathtt{S}} \&\mathtt{v} \mapsto \beta)$, and then has to analyze a reading of `l->next`.

If we consider an abstract state relation of the form $[\mathrm{h}^\sharp \dashrightarrow \mathbf{list}(\mathrm{n})]$, and an assignment that reads or writes a field at base address n, the inductive predicate $\mathbf{list}(\mathrm{n})$ should first be *unfolded* [5]: before the post-condition operators of Section 5.1 can be applied, this predicate first needs to be substituted with the disjunction of cases it is made of, as defined in Section 4. This process is known in reachability shape analyses as a technique to materialize cells [24,15,5]. It results in disjunctive abstract states. For instance, the concretization of the abstract state relation $[\mathrm{h}^\sharp \dashrightarrow \mathbf{list}(\mathrm{n})]$ is included in the union of the concretizations of $[\mathrm{h}^\sharp \dashrightarrow \mathbf{emp}] \wedge \mathrm{n} = \mathbf{0x0}$ and $[\mathrm{h}^\sharp \dashrightarrow (\mathrm{n}{\cdot}\mathtt{next} \mapsto \alpha_n *_{\mathtt{S}} \mathrm{n}{\cdot}\mathtt{data} \mapsto \alpha_d *_{\mathtt{S}} \mathbf{list}(\alpha_n))] \wedge \mathrm{n} \neq \mathbf{0x0})$. This disjunctive abstract states allows to analyze a read or write into a field at address n.

However, this naive extension of unfolding may be imprecise here. Let us consider the unfolding at node n in the abstract state relation $[\mathrm{n} \cdot \mathtt{next} \mapsto \alpha *_{\mathtt{S}} \mathrm{n} \cdot \mathtt{data} \mapsto \beta \dashrightarrow \mathbf{list}(\mathrm{n})]$. The above technique will generate two disjuncts, including one where $\mathrm{n} = \mathbf{0x0}$. However, n cannot be equal to the null pointer here, since n is the base address of a regular list element in the left side of the $[. \dashrightarrow .]$ abstract relation. Therefore, unfolding should take into account information in both sides of abstract relations for the sake of analysis precision.

In the following, we let $\mathbf{unfold}_{\boldsymbol{\Sigma}}(\mathrm{n}, \sigma^\sharp)$ denote the set of disjuncts produced by unfolding an inductive predicate at node n in abstract state $\sigma^\sharp$, if any. For instance, $\mathbf{unfold}_{\boldsymbol{\Sigma}}(\mathrm{n}, \mathbf{list}(\mathrm{n}))$ is $\{(\mathbf{emp} \wedge \mathrm{n} = \mathbf{0x0}), (\mathrm{n} \cdot \mathtt{next} \mapsto \alpha_n *_{\mathtt{S}} \mathrm{n} \cdot \mathtt{data} \mapsto \alpha_d *_{\mathtt{S}}$

$\mathbf{list}(\alpha_n) \wedge \mathrm{n} \neq \mathbf{0x0})\}$. If there is no inductive predicate attached to node n in $\sigma^\sharp$, we let $\mathbf{unfold}_{\boldsymbol{\Sigma}}(\alpha, \sigma^\sharp) = \{\sigma^\sharp\}$. This operator is sound in the sense that, $\gamma_\Sigma(\sigma^\sharp)$ is included in $\cup\{\gamma_\Sigma(\sigma^\sharp_{\mathrm{u}}) \mid \sigma^\sharp_{\mathrm{u}} \in \mathbf{unfold}_{\boldsymbol{\Sigma}}(\mathrm{n}, \sigma^\sharp)\}$.

Using $\mathbf{unfold}_{\boldsymbol{\Sigma}}$, we define the function $\mathbf{unfold}_{\boldsymbol{\Pi}}$ that performs unfolding at a given node and in an abstract state relation as follows:

- $\mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, \mathtt{Id}(\mathrm{h}^\sharp)) = \{\mathtt{Id}(\mathrm{h}^\sharp_{\mathrm{u}}) \wedge \mathrm{c}^\sharp_{\mathrm{u}} \mid (\mathrm{h}^\sharp_{\mathrm{u}} \wedge \mathrm{c}^\sharp_{\mathrm{u}}) \in \mathbf{unfold}_{\boldsymbol{\Sigma}}(\mathrm{n}, \mathrm{h}^\sharp)\}$;
- if the node n carries inductive predicate in $\mathrm{r}^\sharp_0$ then $\mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, \mathrm{r}^\sharp_0 *_{\mathtt{R}} \mathrm{r}^\sharp_1) = \{(\mathrm{r}^\sharp_{0,\mathrm{u}} *_{\mathtt{R}} \mathrm{r}^\sharp_1) \wedge \mathrm{c}^\sharp_{0,\mathrm{u}} \mid (\mathrm{r}^\sharp_{0,\mathrm{u}} \wedge \mathrm{c}^\sharp_{0,\mathrm{u}}) \in \mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, \mathrm{r}^\sharp_0)\}$;
- $\mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]) = \{[\mathrm{h}^\sharp_{\mathrm{i,u}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o,u}}] \wedge (\mathrm{c}^\sharp_{\mathrm{i,u}} \wedge \mathrm{c}^\sharp_{\mathrm{o,u}}) \mid (\mathrm{h}^\sharp_{\mathrm{i,u}} \wedge \mathrm{c}^\sharp_{\mathrm{i,u}}) \in \mathbf{unfold}_{\boldsymbol{\Sigma}}(\mathrm{n}, \mathrm{h}^\sharp_{\mathrm{i}}) \wedge (\mathrm{h}^\sharp_{\mathrm{o,u}} \wedge \mathrm{c}^\sharp_{\mathrm{o,u}}) \in \mathbf{unfold}_{\boldsymbol{\Sigma}}(\mathrm{n}, \mathrm{h}^\sharp_{\mathrm{o}})\}$;
- $\mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, \mathrm{r}^\sharp \wedge \mathrm{c}^\sharp) = \{\mathrm{r}^\sharp_{\mathrm{u}} \wedge (\mathrm{c}^\sharp \wedge \mathrm{c}^\sharp_{\mathrm{u}}) \mid (\mathrm{r}^\sharp_{\mathrm{u}} \wedge \mathrm{c}^\sharp_{\mathrm{u}}) \in \mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, \mathrm{r}^\sharp)\}$.

We note that conjunctions of numerical constraints over node may yield to unfeasible elements being discarded in the last two cases: for instance, in the $[\cdot \dashrightarrow \cdot]$ case, unfolding will only retain disjuncts where both sides of the arrow express compatible conditions over n.

We can prove by case analysis that this unfolding operator is sound:

$$\gamma_\Pi(\rho^\sharp) \subseteq \bigcup\{\gamma_\Pi(\rho^\sharp_{\mathrm{u}}) \mid \rho^\sharp_{\mathrm{u}} \in \mathbf{unfold}_{\boldsymbol{\Pi}}(\mathrm{n}, \rho^\sharp)\}$$

Example 4 (Abstract state relation unfolding and post-condition). Let us consider the analysis of the insertion function of Figure 1. This function should be applied to states where `l` is a non null list pointer (the list should have at least one element), thus, the analysis should start from $\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \mathbf{list}(\alpha)) \wedge \alpha \neq \mathbf{0x0}$ (in this example, we omit `v` for the sake of concision). Before the loop entry, the analysis computes the abstract state relation $\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \mathbf{list}(\alpha)) *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\&\mathtt{c} \mapsto \alpha)] \wedge \alpha \neq \mathbf{0x0}$. To deal with the test $\mathtt{c}{-}{>}\mathtt{next} \mathrel{!}= \mathtt{NULL}$ (and the assignment $\mathtt{c} = \mathtt{c}{-}{>}\mathtt{next}$), the analysis should materialize the cell at node α. This unfolding is performed under the `Id` connective, and produces:

$$\begin{aligned}&\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \alpha \cdot \mathtt{next} \mapsto \alpha_0 *_{\mathtt{S}} \alpha \cdot \mathtt{data} \mapsto \beta_0 *_{\mathtt{S}} \mathbf{list}(\alpha_0))\\ &\qquad *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\&\mathtt{c} \mapsto \alpha)] \wedge \alpha \neq \mathbf{0x0}\end{aligned}$$

In turn, the effect of the condition test and of the assignment in the loop body can be precisely analyzed from this abstract state relation.

5.3 Folding and lattice operations

Like classical shape analyses [15,5], our analysis needs to *fold* inductive predicates so as to (conservatively) decide inclusion and join abstract states. We present folding algorithms in the following paragraphs.

Conservative inclusion checking. Inclusion checking is used to verify logical entailment, to check the convergence of loop iterates, and to support the join / widening algorithm. It consists of a conservative function $\mathbf{isle}_{\mathbb{H}}$ over abstract states and a conservative function $\mathbf{isle}_{\mathbb{R}}$ over abstract state relations, that either return $\mathbf{true}$

$$\frac{\mathrm{h}^\sharp \text{ is of the form } \mathrm{n} \cdot \mathtt{f} \mapsto \mathrm{n}' \text{ or } \mathbf{list}(\mathrm{n}) \text{ or } \mathbf{listseg}(\mathrm{n}, \mathrm{n}')}{\mathrm{h}^\sharp \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp}\ (\sqsubseteq_{=})$$

$$\frac{\mathrm{h}^\sharp \sqsubseteq_{\mathbb{H}} \mathbf{list}(\mathrm{n}')}{\mathbf{listseg}(\mathrm{n}, \mathrm{n}') \ast_{\mathtt{S}} \mathrm{h}^\sharp \sqsubseteq_{\mathbb{H}} \mathbf{list}(\mathrm{n})}\ (\sqsubseteq_{\mathrm{seg}}) \qquad \frac{\mathrm{h}^\sharp_{0,0} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{1,0} \qquad \mathrm{h}^\sharp_{0,1} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{1,1}}{\mathrm{h}^\sharp_{0,0} \ast_{\mathtt{S}} \mathrm{h}^\sharp_{0,1} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{1,0} \ast_{\mathtt{S}} \mathrm{h}^\sharp_{1,1}}\ (\sqsubseteq_{\ast_{\mathtt{S}}})$$

$$\frac{\mathrm{r}^\sharp_{\mathrm{u}} \in \mathbf{unfold}_{\mathbf{\Pi}}(\mathrm{n}, \mathrm{r}^\sharp_1) \qquad \mathrm{r}^\sharp_0 \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_{\mathrm{u}} \qquad \mathrm{r}^\sharp_1 \text{ contains } \mathbf{list}(\mathrm{n}) \text{ or } \mathbf{listseg}(\mathrm{n}, \mathrm{n}')}{\mathrm{r}^\sharp_0 \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_1}\ (\sqsubseteq_{\mathrm{unfold}})$$

$$\frac{\mathrm{h}^\sharp_0 \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_1}{\mathtt{Id}(\mathrm{h}^\sharp_0) \sqsubseteq_{\mathbb{R}} \mathtt{Id}(\mathrm{h}^\sharp_1)}\ (\sqsubseteq_{\mathtt{Id}}) \qquad \frac{\mathrm{h}^\sharp_{\mathrm{i},0} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{\mathrm{i},1} \qquad \mathrm{h}^\sharp_{\mathrm{o},0} \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_{\mathrm{o},1}}{[\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}] \sqsubseteq_{\mathbb{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]}\ (\sqsubseteq_{\dashrightarrow-\mathrm{intro}})$$

$$\frac{\mathrm{r}^\sharp \ast_{\mathtt{R}} [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp] \sqsubseteq_{\mathbb{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]}{\mathrm{r}^\sharp \ast_{\mathtt{R}} \mathtt{Id}(\mathrm{h}^\sharp) \sqsubseteq_{\mathbb{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]}\ (\sqsubseteq_{\mathtt{Id}-\mathrm{weak}}) \qquad \frac{\mathrm{r}^\sharp_{0,0} \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_{1,0} \qquad \mathrm{r}^\sharp_{0,1} \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_{1,1}}{\mathrm{r}^\sharp_{0,0} \ast_{\mathtt{R}} \mathrm{r}^\sharp_{0,1} \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_{1,0} \ast_{\mathtt{R}} \mathrm{r}^\sharp_{1,1}}\ (\sqsubseteq_{\ast_{\mathtt{R}}})$$

$$\frac{\mathrm{r}^\sharp \ast_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},0} \ast_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0} \ast_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1}] \sqsubseteq_{\mathbb{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]}{\mathrm{r}^\sharp \ast_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}] \ast_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}] \sqsubseteq_{\mathbb{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]}\ (\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$$

Fig. 3. Inclusion checking rules

(meaning that the inclusion of concretizations holds) or **false** (meaning that the analysis cannot conclude whether inclusion holds).

Their definition relies on a conservative algorithm, that implements a proof search, based on the rules shown in Figure 3 (for clarity, we omit the numerical constraints inclusion checking). In this system of rules, if $\mathrm{h}^\sharp_0 \sqsubseteq_{\mathbb{H}} \mathrm{h}^\sharp_1$ (resp., $\mathrm{r}^\sharp_0 \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_1$), then $\gamma_{\mathbb{H}}(\mathrm{h}^\sharp_0) \subseteq \gamma_{\mathbb{H}}(\mathrm{h}^\sharp_1)$ (resp., $\gamma_{\mathbb{R}}(\mathrm{r}^\sharp_0) \subseteq \gamma_{\mathbb{R}}(\mathrm{r}^\sharp_1)$). The rules $(\sqsubseteq_{=})$, $(\sqsubseteq_{\mathrm{seg}})$ and $(\sqsubseteq_{\ast_{\mathtt{S}}})$ are specific to reasoning of abstract states, and are directly inspired from [5] (they allow to reason over equal abstract regions, over segments, and over separating conjunction). The rule $(\sqsubseteq_{\mathrm{unfold}})$ allows to reason by unfolding of inductive predicates, at the level of relations. Finally, the rules $(\sqsubseteq_{\mathtt{Id}})$, $(\sqsubseteq_{\dashrightarrow-\mathrm{intro}})$, $(\sqsubseteq_{\mathtt{Id}-\mathrm{weak}})$, $(\sqsubseteq_{\ast_{\mathtt{R}}})$ and $(\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$ allow to derive inclusion over abstract state relations, and implement the properties observed in Theorem 1. The proof search algorithm starts from the goal to prove and attempt to apply these rules so as to complete an inclusion derivation. We observe that abstract states are equivalent up to a renaming of the internal nodes (the nodes that are not of the form &$\mathtt{x}$), thus, the implementation also takes care of this renaming, although the rules of Figure 3 do not show it, as this issue is orthogonal to the reasoning over abstract state relations which is the goal of this paper (indeed, this requires complex renaming functions that are made fully explicit in [5]). The rules can be proved sound one by one, thus they define a sound inclusion checking procedure:

Theorem 2 (Soundness of inclusion checking). *If* $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1 \in \mathbb{H}$ *and* $\mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1 \in \mathbb{R}$ *then:*

$$\begin{array}{c} \mathbf{isle}_{\mathbb{H}}(\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1) = \mathbf{true} \Longrightarrow \gamma_{\mathbb{H}}(\mathrm{h}^\sharp_0) \subseteq \gamma_{\mathbb{H}}(\mathrm{h}^\sharp_1) \\ \mathbf{isle}_{\mathbb{R}}(\mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1) = \mathbf{true} \Longrightarrow \gamma_{\mathbb{R}}(\mathrm{r}^\sharp_0) \subseteq \gamma_{\mathbb{R}}(\mathrm{r}^\sharp_1) \end{array}$$

Example 5 (Inclusion checking). Let us consider the following abstract state relations, and discuss the computation of $\mathbf{isle}_{\mathbb{R}}(\mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1)$:

$$\begin{aligned}
\mathrm{r}^\sharp_0 &= \mathtt{Id}(\mathrm{n} \cdot \mathtt{next} \mapsto \alpha_0 *_{\mathtt{S}} \mathbf{list}(\alpha_0)) *_{\mathtt{R}} [\mathrm{n} \cdot \mathtt{data} \mapsto \alpha_1 \dashrightarrow \mathrm{n} \cdot \mathtt{data} \mapsto \alpha_2] \\
\mathrm{r}^\sharp_1 &= [\mathbf{list}(\mathrm{n}) \dashrightarrow \mathbf{list}(\mathrm{n})]
\end{aligned}$$

Using first rule $(\sqsubseteq_{\mathtt{Id}-\mathrm{weak}})$ then rule $(\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$, this goal gets reduced into checking the inclusion $[\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1] \sqsubseteq_{\mathbb{R}} \mathrm{r}^\sharp_1$, where $\mathrm{h}^\sharp_0 = \mathrm{n} \cdot \mathtt{next} \mapsto \alpha_0 *_{\mathtt{S}} \mathbf{list}(\alpha_0) *_{\mathtt{S}} \mathrm{n} \cdot \mathtt{data} \mapsto \alpha_1$ and $\mathrm{h}^\sharp_1 = \mathrm{n} \cdot \mathtt{next} \mapsto \alpha_0 *_{\mathtt{S}} \mathbf{list}(\alpha_0) *_{\mathtt{S}} \mathrm{n} \cdot \mathtt{data} \mapsto \alpha_2$. In turn, this inclusion follows from rule $(\sqsubseteq_{\mathrm{unfold}})$.

Join / widening operators. In the following, we define abstract operators $\mathbf{wid}_{\mathbb{H}}$, $\mathbf{wid}_{\mathbb{R}}$ that respectively operate over abstract states and abstract state relations, and compute an over-approximation for concrete unions. They also ensure termination and serve as widening. The algorithm to compute these two functions heavily relies on the inclusion checking that was discussed in the previous paragraph. Indeed, the widening functions compute results that are more approximate than their arguments. To achieve this, they search for syntactic patterns in their arguments and produce outputs that inclusion checking proves more general. This process is performed region by region on both arguments of the widening, as formalized in [5, Figure 7]. We discuss in the following a list of such widening rules:

- when both arguments of widening are equal to a same base predicate, widening is trivial, and returns the same base predicate, thus for instance:

$$\begin{aligned}
\mathbf{wid}_{\mathbb{H}}(\mathrm{n} \cdot \mathtt{f} \mapsto \alpha, \mathrm{n} \cdot \mathtt{f} \mapsto \alpha) &= \mathrm{n} \cdot \mathtt{f} \mapsto \alpha \\
\mathbf{wid}_{\mathbb{H}}(\mathbf{list}(\alpha), \mathbf{list}(\alpha)) &= \mathbf{list}(\alpha)
\end{aligned}$$

- when applied to two abstract relations that consist of the same connective, the widening functions simply calls themselves recursively on the sub-components:

$$\begin{aligned}
\mathbf{wid}_{\mathbb{R}}(\mathtt{Id}(\mathrm{h}^\sharp_0), \mathtt{Id}(\mathrm{h}^\sharp_1)) &= \mathtt{Id}(\mathbf{wid}_{\mathbb{H}}(\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1)) \\
\mathbf{wid}_{\mathbb{R}}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}], [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]) &= [\mathbf{wid}_{\mathbb{H}}(\mathrm{h}^\sharp_{\mathrm{i},0}, \mathrm{h}^\sharp_{\mathrm{i},1}) \dashrightarrow \mathbf{wid}_{\mathbb{H}}(\mathrm{h}^\sharp_{\mathrm{o},0}, \mathrm{h}^\sharp_{\mathrm{o},1})] \\
\mathbf{wid}_{\mathbb{R}}(\mathrm{r}^\sharp_{0,0} *_{\mathtt{R}} \mathrm{r}^\sharp_{0,1}, \mathrm{r}^\sharp_{1,0} *_{\mathtt{R}} \mathrm{r}^\sharp_{1,1}) &= \mathbf{wid}_{\mathbb{R}}(\mathrm{r}^\sharp_{0,0}, \mathrm{r}^\sharp_{1,0}) *_{\mathtt{R}} \mathbf{wid}_{\mathbb{R}}(\mathrm{r}^\sharp_{0,1}, \mathrm{r}^\sharp_{1,1})
\end{aligned}$$

- when applied to an $\mathtt{Id}(\cdot)$ predicate and another abstract relation, widening first tries to maintain the $\mathtt{Id}(\cdot)$ predicate, and, if this fails, tries to weaken it into an $[\cdot \dashrightarrow \cdot]$ predicate:

$$\begin{aligned}
&\text{if } \mathbf{isle}_{\mathbb{H}}(\mathrm{h}^\sharp_0, \mathrm{h}^\sharp) = \mathbf{true} \text{ then,} \\
&\mathbf{wid}_{\mathbb{R}}(\mathtt{Id}(\mathrm{h}^\sharp_0), \mathrm{r}^\sharp) = \begin{cases} \mathtt{Id}(\mathrm{h}^\sharp) & \text{if } \mathbf{isle}_{\mathbb{R}}(\mathrm{r}^\sharp, \mathtt{Id}(\mathrm{h}^\sharp)) = \mathbf{true} \\ [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp] & \text{otherwise, if } \mathbf{isle}_{\mathbb{R}}(\mathrm{r}^\sharp, [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]) = \mathbf{true} \end{cases}
\end{aligned}$$

- when applied to an $[\cdot \dashrightarrow \cdot]$ predicate, the widening tries to weaken the other argument accordingly:

$$\begin{aligned}
&\text{if } \mathbf{isle}_{\mathbb{H}}(\mathrm{h}^\sharp_{\mathrm{i},0}, \mathrm{h}^\sharp_{\mathrm{i}}) = \mathbf{true} \text{ and } \mathbf{isle}_{\mathbb{H}}(\mathrm{h}^\sharp_{\mathrm{o},0}, \mathrm{h}^\sharp_{\mathrm{o}}) = \mathbf{true} \\
&\text{and } \mathbf{isle}_{\mathbb{R}}(\mathrm{r}^\sharp, [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]) = \mathbf{true} \text{ then,} \\
&\qquad \mathbf{wid}_{\mathbb{R}}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}], \mathrm{r}^\sharp) = [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]
\end{aligned}$$

Each of these operations is sound, and the results computed by widening are also sound:

Theorem 3 (Soundness of widening). *If* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp \in \mathbb{H}$ *and* $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp \in \mathbb{R}$ *then:*

$$\gamma_{\mathbb{H}}(\mathrm{h}_0^\sharp) \cup \gamma_{\mathbb{H}}(\mathrm{h}_1^\sharp) \subseteq \gamma_{\mathbb{H}}(\mathbf{wid}_{\mathbb{H}}(\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp)) \qquad \gamma_{\mathbb{R}}(\mathrm{r}_0^\sharp) \cup \gamma_{\mathbb{R}}(\mathrm{r}_1^\sharp) \subseteq \gamma_{\mathbb{R}}(\mathbf{wid}_{\mathbb{R}}(\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp))$$

Furthermore, termination of widening follows from an argument similar to [5].

Example 6 (Widening). We consider the analysis of the program of Figure 1, and more specifically, the widening after the first abstract iteration over the loop:

$$\begin{aligned}
&\mathbf{wid}_{\mathbb{R}}(\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \mathbf{list}(\alpha) *_{\mathtt{S}} \&\mathtt{v} \mapsto \beta) *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \&\mathtt{c} \mapsto \alpha],\\
&\quad \mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \alpha \cdot \mathtt{data} \mapsto \alpha_{\mathrm{d}} *_{\mathtt{S}} \alpha \cdot \mathtt{next} \mapsto \alpha_{\mathrm{n}} *_{\mathtt{S}} \mathbf{list}(\alpha_{\mathrm{n}}) *_{\mathtt{S}} \&\mathtt{v} \mapsto \beta)\\
&\qquad\quad *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \&\mathtt{c} \mapsto \alpha_{\mathrm{n}}])\\
&= \mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_{\mathtt{S}} \mathbf{listseg}(\alpha, \alpha') *_{\mathtt{S}} \mathbf{list}(\alpha') *_{\mathtt{S}} \&\mathtt{v} \mapsto \beta) *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \&\mathtt{c} \mapsto \alpha']
\end{aligned}$$

This abstract widening performs some generalization and introduces a list segment inductive predicate, that over-approximates an empty segment in the left argument, and a segment of length one. It also involves some renaming of symbolic nodes (as observed in the previous paragraph, the concretization of an abstract states is unchanged under symbolic nodes renaming).

5.4 Analysis

The abstract semantics $[\![.]\!]^\sharp_{\mathcal{R}}$ relies on the abstract operations defined in Section 5.1, on the unfolding of Section 5.2 to analyze basic statements, and on the folding operations defined in Section 5.3 to cope with control flow joins and loop invariants computation. Soundness follows from the soundness of the basic operations.

Theorem 4 (Soundness). *The analysis is sound in the sense that, for all program* P *and for all abstract state relation* $\rho^\sharp$*:*

$$\forall(\sigma_0, \sigma_1) \in \gamma_{\Pi}(\rho^\sharp),\ \forall \sigma_2 \in \mathbb{M}, (\sigma_1, \sigma_2) \in [\![P]\!]_{\mathcal{R}} \Longrightarrow (\sigma_0, \sigma_2) \in \gamma_{\Pi}([\![P]\!]^\sharp_{\mathcal{R}}(\rho^\sharp))$$

6 Experimental Evaluation

In this section, we report on the implementation of our analysis and try to evaluate:

1. whether it can prove precise and useful relational properties, and
2. how it compares with a more classical reachability shape analysis.

Our implementation supports built-in inductive predicates to describe singly linked lists and binary trees. It provides both the analysis described in this paper, and a basic reachability shape analysis in the style of [5], and supporting the same inductive predicates. It was implemented as a Frama-C [19] plugin consisting of roughly 7800 lines of OCaml. We have ran both the reachability shape analysis and relational shape analysis on series of small programs manipulating lists and trees listed in Table 1. These tests are selected to test specifically the relational domain (and not a full analysis). This allows us to not only assess the results of

Structure	Function	Time (in ms)		Loop iterations	Relational
		Reach	Relat.		Property
sll	allocation	0.53	1.27	2	yes
sll	deallocation	0.34	0.99	2	yes
sll	traversal	0.53	0.83	2	yes
sll	insertion (head)	0.32	0.33	0	yes
sll	insertion (random pos)	1.98	2.75	2	yes
sll	insertion (random)	2.33	3.94	2	yes
sll	reverse	0.52	2.36	2	partial
sll	map	0.66	1.17	2	partial
tree	allocation	0.94	2.21	2	yes
tree	search	1.06	1.76	2	yes

Table 1. Experiment results (sll: singly linked lists; tree: binary trees; time in milliseconds averaged over 1000 runs on a laptop with Intel Core i7 running at 2.3 GHz, with 16 Gb RAM, for the reachability and relational analyses; the last column states whether the relational shape analysis computed the expected abstract relation)

the analysis computing abstract state relations, but also to compare them with an analysis that infers abstract states.

First, we discuss whether the analysis computing abstract state relations computes the expected relations, that describes the most precisely the transformation implemented by the analyzed function. As an example, in the case of an insertion at the head of a list, we expect the abstract relation below, that expresses that the body of the list was not modified:

$$[\&\texttt{l} \mapsto \alpha \dashrightarrow \&\texttt{l} \mapsto \beta] *_{\texttt{R}} [\mathbf{emp} \dashrightarrow \beta \cdot \texttt{next} \mapsto \alpha *_{\texttt{S}} \beta \cdot \texttt{data} \mapsto \delta] *_{\texttt{R}} \texttt{Id}(\mathbf{list}(\alpha))$$

We observe that the state relation computed in all test cases except the list reverse and map are the most precise. For example, with the function map that traverses a list and modifies only its `data` fields, the relation obtained is:

$$\texttt{Id}(\&\texttt{l} \mapsto \alpha) *_{\texttt{R}} [(\mathbf{listseg}(\alpha, \beta)) \dashrightarrow (\mathbf{listseg}(\alpha, \beta))]$$

This relation shows that both input and output lists start at the address α and end at the address β. This is not enough to prove that the lists contain the same addresses linked in the same order.

Second, we compare the runtime of the relational analysis and of the reachability analysis. We observe that the slow-down is at most 4x (reverse), and is about 2x in most cases. An exception is the list head insertion, which incurs no slowdown. This is due to the fact this analysis does not require computing an abstract join. While these test cases are not large, these results show that the analysis computing abstract state relations has a reasonable overhead compared to a classical analysis, yet it computes stronger properties. Furthermore, it would be more adapted to a modular interprocedural analysis.

7 Related Works

Our analysis computes an abstraction of the relational semantics of programs so as to capture the effect of a function or other blocks of code using an element of some specifically designed abstract domain. This technique has been applied to other abstractions in the past, and often applied to design *modular* static analyses [10], where program components can be analyzed once and separately. For numerical domains, it simply requires duplicating each variable into two instances respectively describing the old and the new value, and using a relational domain to the inputs and outputs. For instance, [22] implements this idea using convex polyhedra and so as to infer abstract state relations for numerical programs. It has also been applied to shape analyses based on Three Valued Logic [24] in [17]. This work is probably the closest to ours, but it relies on a very different abstraction using a TVLA whereas we use a set of abstract predicates based on separation logic. It uses the same variable duplication trick as mentioned above. Our analysis also has a notion of overlaid old / new predicates, but these are described heap regions, inside separation logic formulas. Desynchronized separation [11] also introduces a notion of overlaid state in separation logic, but does not support inductive predicates as our analysis does. Instead, it allows to reason on abstractions of JavaScript open objects seen as dictionaries. Also, [13,14] can express relations between heaps in different states using temporal logic extensions and automatas. In the context of functional languages, [18] allows to write down relations between function inputs and outputs, and relies on a solver to verify that constraints hold and [25] computes shape specifiations by learning. Modular analyses that compute invariants by separate analysis of program components [6,12,4] use various sorts of abstractions for the behavior of program components. A common pattern is to use tables of couples made of an abstract pre-condition and a corresponding abstract post-condition, effectively defining a sort of cardinal power abstraction [9]. This technique has been used in several shape analyses based on separation logic [3,16,20,2]. We believe this tabular approach could benefit from abstractions of relations such as ours to infer stronger properties, and more concise summaries.

8 Conclusion

In this paper, we have introduced a set of logical connectives inspired by separation logic, to describe state relations rather than states. We have built upon this logic an abstract domain, and a static analysis based on abstract interpretation that computes conservative state relations. Experiments prove it effective for the analysis of basic data structure library functions.

Acknowledgements. We thank Arlen Cox for fruitful discussions, and Francois Berenger, Huisong Li, Jiangchao Liu and the anonymous reviewers for their comments on an earlier version of this paper. This work has received funding from the European Research Council under the EU's seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD, and from Bpifrance, grant agreement P3423-189738, FUI Project P-RC2.

References

1. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi c specification language, 2008.
2. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Footprint analysis : A shape analysis that discovers preconditions. In *Static Analysis Symposium (SAS)*, pages 402–418. Springer, 2007.
3. 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.
4. Ghila Castelnuovo, Mayur Naik, Noam Rinetzky, Mooly Sagiv, and Hongseok Yang. Modularity in lattices: A case study on the correspondence between top-down and bottom-up analysis. In *Static Analysis Symposium (SAS)*, pages 252–274. Springer, 2015.
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. Ramkrishna Chatterjee, Barbara G Ryder, and William A Landi. Relevant context inference. In *Symposium on Principles of Programming Languages (POPL)*, pages 133–146. ACM, 1999.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM, 1978.
8. 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.
9. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1979.
10. Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Conference on Compiler Construction*, pages 159–179. Springer, 2002.
11. Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Desynchronized multi-state abstractions for open programs in dynamic languages. In *European Symposium on Programming (ESOP)*, pages 483–509. Springer, 2015.
12. Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 567–577. ACM, 2011.
13. Dino Distefano, Joost-Pieter Katoen, and Arend Rensik. Who is pointing when to whom? In *Foundations of Software Technology and Theoretical (FSTTCS)*, pages 250–262. Springer, 2004.
14. Dino Distefano, Joost-Pieter Katoen, and Arend Rensik. Safety and liveness in concurrent pointer programs. In *Formal Methods for Components and Objects (FMCO)*, pages 280–312. Springer, 2005.
15. 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.
16. Bhargav S Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V Nori. Bottom-up shape analysis. In *Static Analysis Symposium (SAS)*, pages 188–204. Springer, 2009.

17. Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):5, 2010.
18. Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. In *International Colloquium on Function Programming*, pages 311–324. ACM, 2014.
19. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
20. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Conference on Computer Aided Verification (CAV)*, pages 52–68. Springer, 2014.
21. Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420, 1998.
22. Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345. Springer, 2006.
23. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics In Computer Science (LICS)*, pages 55–74. IEEE, 2002.
24. 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.
25. He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 491–507. ACM, 2016.