A relational shape abstract domain

Hugo Illous, Matthieu Lemerre, Xavier Rival

▶ To cite this version:

Hugo Illous, Matthieu Lemerre, Xavier Rival. A relational shape abstract domain. Formal Methods in System Design, Springer Verlag, 2020, 10.1007/s10703-021-00366-4 . cea-03218896

HAL Id: cea-03218896

https://hal-cea.archives-ouvertes.fr/cea-03218896

Submitted on 6 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Methods in System Design manuscript No.
(will be inserted by the editor)

A Relational Shape Abstract Domain

Hugo Illous · **Matthieu Lemerre** · **Xavier Rival**

Abstract Static analyses aim at inferring semantic properties of programs. We distinguish two important classes of static analyses: state analyses and relational analyses. While state analyses aim at computing an over-approximation of reachable states of programs, relational analyses aim at computing functional properties over the input-output states of programs. Several advantages of relational analyses are their ability to analyze incomplete programs, such as libraries or classes, but also to make the analysis modular, using input-output relations as composable summaries for procedures. In the case of numerical programs, several analyses have been proposed that utilize relational numerical abstract domains to describe relations. On the other hand, designing abstractions for relations over input-output 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 basic libraries of programs manipulating lists and trees.

Keywords Static analysis · Abstract interpretation · Shape analysis · Separation Logic · Relational properties

1 Introduction

Generally, static analyses aim at automatically inferring, or computing, semantic properties from programs. Two common families of analyses are *state analyses* and *relational analyses*.

H. Illous · M. Lemerre
CEA, LIST, Software Reliability and Security Laboratory, P.C. 174, Gif-sur-Yvette, 91191, France
Tel.: +331-69-08-26-28
E-mail: matthieu.lemerre@cea.fr
Orcid: 0000-0002-1081-0467

H. Illous · X. Rival
INRIA Paris/CNRS/École Normale Supérieure/PSL Research University
E-mail: hugo.illous@ens.fr
E-mail: xavier.rival@ens.fr

While state analyses aim at computing an over-approximation of the set of the *reachable* states of programs, relational analyses aim at computing an over-approximation for the relations between the *input and output* states of programs.

Benefits of Relational Analyses. In general, sets of states are easier to abstract than state relations, which often makes states analyses simpler to design. On the other hand, abstracting relations brings several advantages.

First, state relations allow to make the analyses modular [19, 38, 30, 13, 8] 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 [2, 34, 36, 29] 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 state analyses cannot be used to infer precise function contracts.

The Need of Relational Abstractions. In general, the increased expressiveness of relational analyses requires more expressive abstractions. Let us discuss, as an example, the case of numerical 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 domains such as intervals [17] cannot capture any interesting relation between input and output states. Conversely, relational numerical abstract domains such as convex polyhedra [15] can effectively capture relations between input and output states, as shown in [38]: for instance, when applied to a program that increments $\mathtt{x}$ by one, this analysis can infer the relation $\mathtt{x}' = \mathtt{x} + 1$.

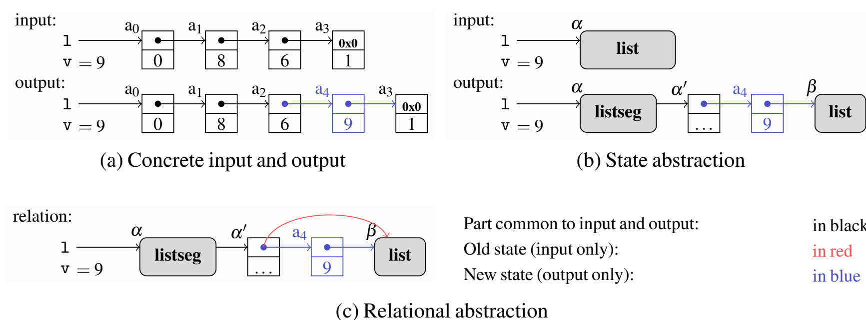
In the context of programs manipulating complex data structures, relational analysis could allow to compute interesting classes of program properties. For instance, such analyses could express and verify that some memory areas were not physically modified by a program. State analyses such as [40, 25, 10] cannot distinguish a program p_1 from a different program p_2 that has a similar behavior. For example, if p_1 inputs a list and leaves it unmodified and p_2 inputs a list, copies it into an identical version and deallocates the input list, only a relational analysis could distinguish p_1 from p_2. More generally, it is often interesting to infer that a memory region is not modified by a program.

Separation logic [39] provides an elegant description for sets of memory states and is at the foundation of many state 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 relations.

Our Approach and Contributions. In this paper, we propose a logic inspired by separation logics and that can describe such relational 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. It also enriches these relational connectives with sets of predicates that can describe specific transformations. We build an abstract domain upon this logic, and apply it to design an analysis for procedures

```
1  typedef struct list { struct list * next; int data; } list;
2
3  void insert_non_empty( list *l, int v ) {
4    list *c = l;
5    while( c->next != NULL && ... ){
6      c = c->next;
7    }
8    list *e = malloc( sizeof( list ) );
9    e->next = c->next; c->next = e; e->data = v;
10 }
```

Fig. 1: A list insertion program



(a) Concrete input and output

(b) State abstraction

(c) Relational abstraction

Fig. 2: State and relational abstractions for the program of Figure 1.

manipulating simple list or tree data structures, thereby automatically inferring relational function contracts for these procedures. We make the following contributions:

- In Section 2, we demonstrate the abstraction of heap relations using a specific family of heap predicates. We also propose an extension of this abstraction, that improves the expressiveness of heap relations;
- In Section 4, we set up a logic to describe heap relations and lift it into an abstract domain that describes concrete relations defined in Section 3;
- In Section 5, we design static analysis algorithms to infer memory state relations from abstract pre-conditions;
- 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

In this section, we first compare the use of states and relational abstractions for a relational analysis. We then present an extension of the relational abstraction that improves the precision of the analysis.

2.1 Relational Analysis With State Abstraction

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 traverses the list 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 2a 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 a relational analysis of this program that does not use relational abstraction, but only states abstraction. We consider an abstraction based on separation logics with inductive predicates as used in [25, 10]. We assume that the predicate $\mathbf{list}(\alpha)$ describes heap regions that consist of a well-formed linked list starting at address α (α is a symbolic value 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 a symbolic variable β and a region that can be described by $\mathbf{list}(\beta)$. Thus, the valid input states for the insertion function can be abstracted by the abstract state shown on the top of Figure 2b. 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 $\mathbf{listseg}(\alpha, \alpha')$, that is defined in a similar way as for **list**: it describes a 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 $\mathbf{listseg}(\alpha, \alpha)$). 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 2b 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 (when applied to non-empty linked lists), 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 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. Finally, it does not express that the element of address a_4 has been freshly allocated by the function (this abstraction allows this element to be an element of the input list). This is a consequence of the fact that each abstract state in Figure 2b independently describes a set of concrete heaps.

2.2 Simple Heap Relations Abstraction

We now discuss a relational analysis of the code of Figure 1 using a relational abstraction based on heap abstractions.

To abstract heap relations instead of sets of heaps, we now propose to define a new structure in Figure 2c, that is based on new predicates inspired by separation logic and that partially overlays the abstractions of input and output heaps.

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 an input heap and an output heap, that

```
1  list *sort(list *l) {
2    list *res = NULL;
3    while(1) {
4      list *p_max = l;
5      int v_max = l->data;
6      list *c = l;
7      while(c->next) {
8        if(c->next->data > v_max) {
9          v_max = c->next->data;
10         p_max = c;
11       }
12       c = c->next;
13     }
14     list *tmp;
15     if(p_max == l && v_max == l->data) {
16       //the maximum is the head
17       tmp = l;
18       l = l->next;
19     } else {
20       tmp = p_max->next;
21       p_max->next = p_max->next->next;
22     }
23      tmp->next = res;
24      res = tmp;
25   }
26   return res;
27 }
```

Fig. 3: A list sort program

are *physically equal* and can both be described by $\mathbf{list}(\beta)$. The same predicate can be used to describe that the initial segment has not changed between the two heaps: $\mathtt{Id}(\mathbf{listseg}(\alpha, \alpha'))$.

Second, we need to define a counterpart for separating conjunction at the relation level. Indeed, the effect of the insertion function can be decomposed into 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. Thus, we need to describe relations between states where a region has been modified or allocated. To account for this, we need a new connective $[\cdot \dashrightarrow \cdot]$ which is applied to two abstract heaps: if $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp$ are abstract heaps (described by formulas in the usual separation logic with inductive predicates), then $[\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_1^\sharp]$ describes the transformation of an input heap described by $\mathrm{h}_0^\sharp$ into an output heap described by $\mathrm{h}_1^\sharp$. This is presented with different colors in Figure 2.

2.3 Heap Relations Abstraction Extension

We now consider the code shown in Figure 3, that implements a sort of a linked list. We observe this is an in-place sort: it works directly on the input list, without allocating or deleting cells.

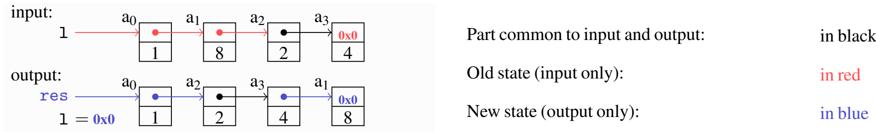


Fig. 4: Example of concrete input and output states for the program of Figure 3

Figure 4 illustrates this property using different colors. We can see that exactly all the memory cells in the input list (at addresses a_0, a_1, a_2 and a_3) are physically present in the output list. We also notice that each value of the `data` field of each list element is unchanged (the `data` field at address a_0 is 1, at address a_1 is 8, ...), but that only the `next` field of some list elements has been modified. For instance, the `next` field of the list element at address a_3 pointed to the null pointer, now it points to the address a_1. We can say that the list is *partially modified*.

With a relational analysis using only states abstraction, we can only show that if this function inputs a well formed linked list $\textbf{list}(\alpha)$, then it outputs a well form linked list $\textbf{list}(\beta)$. Here again, we cannot deduce any information about how the output list is obtained. However, using the heap relations abstraction described above, we could not infer more interesting properties than an analysis using states abstraction. Indeed, we could only express the transformation of a well formed linked list into an other, that is potentially the same. The first version of our heap relation abstraction is expressive enough for programs that strictly do not modify a heap region or only modify a finite number of memory cells. It reaches its limit of precision when a program modifies partially an unbounded number of memory cells.

In this paper, we propose an extension of our heap relations abstraction, that improves a lot its precision, in a generic way. We consider first the basic version of our heap relations abstraction, where the relation $[h_0^\sharp \dashrightarrow h_1^\sharp]$ expresses that the input heap described by $h_0^\sharp$ has been transformed into the output heap described by $h_1^\sharp$. Intuitively, for the program in Figure 3, we would like to add to this relation that the output heap contains exactly the same physical memory cells as the input list, and that the value of each of these cells is left unchanged. To achieve this, we adjoin to the $[\cdot \dashrightarrow \cdot]$ connective transformation predicates that can capture various properties of the transformation. More precisely, given a set of transformation predicates $\mathbb{T}^\sharp$, and an element $t^\sharp$ of $\mathbb{T}^\sharp$, we let the relation $[h_0^\sharp \dashrightarrow h_1^\sharp]_{t^\sharp}$ denote pairs of heaps where the input heap is described by $h_0^\sharp$ has been transformed into an output heap described by $h_1^\sharp$, and such that this pair of heaps meets the condition expressed by $t^\sharp$. In order to keep the construction general so that other transformation predicates can easily be integrated to the analysis, we do not fix a unique $\mathbb{T}^\sharp$. Instead, we define a generic transformation predicate interface that comprises a set $\mathbb{T}^\sharp$ and some analysis operations that are subject to soundness conditions. Moreover, we present several instances of this generic interface, including the one described above.

In Section 4 we formalize the heap states abstraction based on Separation logic [39], and the new relational connectives and the relational abstraction that they define. We also formalize the extension of these connectives for any set $\mathbb{T}^\sharp$ of transformation predicates and give examples of such predicates that describe the properties for the sort function of Figure 3.

In Section 5, we define the analysis algorithm that uses these new connectives to compute a sound over-approximation of the input-output memory states relations. It proceeds by forward abstract interpretation [17]. It starts with the identity relation of a given pre-condition at the function's entry, and computes input-output heap relations step by step. The analysis algorithms need to unfold inductive predicates to materialize cells (for instance to analyze the test at line

$$
\begin{array}{rcll}
loc\ (\in \mathtt{L}) & ::= & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) \\
& | & loc_1 \cdot \mathtt{f} & (loc_1 \in \mathtt{L}; \mathtt{f} \in \mathbb{F}) \\
& | & *exp & (exp \in \mathtt{E})
\end{array}
\qquad
\begin{array}{rcll}
exp\ (\in \mathtt{E}) & ::= & loc & (loc \in \mathtt{L}) \\
& | & \&loc & (loc \in \mathtt{L}) \\
& | & \mathrm{v} & (\mathrm{v} \in \mathbb{V}) \\
& | & exp_1 \oplus exp_2 & (exp_1, exp_2 \in \mathtt{E}) \\
\oplus & ::= & + \mid - \mid = \mid \neq \mid \ldots &
\end{array}
$$

$$
\begin{array}{rcll}
p\ (\in \mathtt{P}) & ::= & loc = exp & (loc \in \mathtt{L}; exp \in \mathtt{E}) \\
& | & loc = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}) & (loc \in \mathtt{L}; \mathtt{f}_i \in \mathbb{F}) \\
& | & \mathbf{free}(loc) & (loc \in \mathtt{L}) \\
& | & p_1\, ;\, p_2 & (p_1, p_2 \in \mathtt{P}) \\
& | & \mathbf{if}\,(exp)\, p_1\, \mathbf{else}\, p_2 & (exp \in \mathtt{E}; p_1, p_2 \in \mathtt{P}) \\
& | & \mathbf{while}\,(exp)\, p_1 & (exp \in \mathtt{E}; p_1 \in \mathtt{P})
\end{array}
$$

Fig. 5: Syntax of the C-like imperative programming language

5), and to fold inductive predicates in order to analyze loops. In addition to this, it also needs to reason over `Id`, $[\cdot \dashrightarrow \cdot]$ and $*_{\mathtt{R}}$ predicates, and perform operations similar to unfolding and folding on them.

3 Concrete Semantics

In this section, we define a concrete program semantics for a C-like imperative programming language.

3.1 Programming Language

We first consider a C-like imperative programming language, whose syntax is described in Figure 5. This language handles assignment, memory allocation and deallocation, sequences, conditionals and loops. It does not handle arrays. The sets `L` and `E` define respectively the sets of locations and expressions, and $\mathbb{X}$ defines the set of program variables. A *location* $loc \in \mathtt{L}$ designates the address of a memory cell. It can be either a program variable `x`, a location offset by a field ($loc \cdot \mathtt{f}$), or the value of a pointer expression ($*exp$). We assume that all field names $\mathtt{f} \in \mathbb{F}$ are implicitly converted into numerical offsets, and that `0` designates the null offset (scalars are viewed as records with only one field, which is `0`). We also write exp `->` `f` the syntactic sugar for the location $(*exp) \cdot \mathtt{f}$. An *expression* $exp \in \mathtt{E}$ denotes a value. It can be the content at a memory location (loc), the address of a memory location ($\&loc$), any value v, or a binary expression $exp_1 \oplus exp_2$. The operator $\oplus$ designates any standard binary operator, like addition, subtraction, equality test, etc...

3.2 Concrete Memory States

Let $\mathbb{A}$ be the set of addresses and $\mathbb{V}$ the set of values, we assume that any address $\mathrm{a} \in \mathbb{A}$ is also a value $\mathrm{v} \in \mathbb{V}$, i.e. $\mathbb{A} \subseteq \mathbb{V}$. A *concrete heap* $\mathrm{h} \in \mathbb{H} = \mathbb{A} \rightharpoonup \mathbb{V}$ is a partial function from addresses to values. We write $[\mathrm{a}_1 \mapsto \mathrm{v}_1; \ldots; \mathrm{a}_n \mapsto \mathrm{v}_n]$ for the concrete heap where each cell at address a_i contains the value v_i, with $1 \leq i \leq n$. We admit that all cells have the same size. We also note $\mathrm{h}[\mathrm{a} \leftarrow \mathrm{v}]$ the heap where we update the content of the cell at address a with value v in the heap h.

$$\mathscr{L}[\![\mathtt{x}]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathrm{e}(\mathtt{x})$$
$$\mathscr{L}[\![loc \cdot \mathtt{f}]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathscr{L}[\![loc]\!](\mathrm{e},\mathrm{h}) + \mathtt{f}$$
$$\mathscr{L}[\![*exp]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathscr{E}[\![exp]\!](\mathrm{e},\mathrm{h})$$
$$\mathscr{E}[\![loc]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathrm{h}(\mathscr{L}[\![loc]\!](\mathrm{e},\mathrm{h}))$$
$$\mathscr{E}[\![\&loc]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathscr{L}[\![loc]\!](\mathrm{e},\mathrm{h})$$
$$\mathscr{E}[\![\mathrm{v}]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathrm{v}$$
$$\mathscr{E}[\![exp_1 \oplus exp_2]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathscr{E}[\![exp_1]\!](\mathrm{e},\mathrm{h}) \oplus \mathscr{E}[\![exp_2]\!](\mathrm{e},\mathrm{h})$$

Fig. 6: Concrete semantics for memory locations and expressions

$$\frac{\mathscr{E}[\![exp]\!](\mathrm{e},\mathrm{h}) = \mathrm{v} \quad \mathscr{L}[\![loc]\!](\mathrm{e},\mathrm{h}) = \mathrm{a}}{[\![loc = exp]\!](\mathrm{e},\mathrm{h}) \Downarrow (\mathrm{e}, h[\mathrm{a} \leftarrow \mathrm{v}])} \qquad \frac{[\![p_1]\!](\mathrm{e},\mathrm{h}) \Downarrow (\mathrm{e}',\mathrm{h}')}{[\![p_1\,;\,p_2]\!](\mathrm{e},\mathrm{h}) \Downarrow [\![p_2]\!](\mathrm{e}',\mathrm{h}')}$$

$$\frac{\mathscr{L}[\![loc]\!](\mathrm{e},\mathrm{h}) = \mathrm{a} \quad \mathrm{h}' = [\mathrm{a}' + \mathtt{f}_1 \mapsto \mathrm{v}_1; \ldots; \mathrm{a}' + \mathtt{f}_n \mapsto \mathrm{v}_n] \quad \mathrm{a}' \in \mathbb{A}, \quad \mathrm{v}_1, \ldots, \mathrm{v}_n \in \mathbb{V}}{[\![loc = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\})]\!](\mathrm{e},\mathrm{h}) \Downarrow (\mathrm{e}, \mathrm{h}[\mathrm{a} \leftarrow \mathrm{a}'] \circledast \mathrm{h}')}$$

$$\frac{\mathscr{E}[\![loc]\!](\mathrm{e},\mathrm{h}) = \mathrm{a} \quad \mathrm{h} = \mathrm{h}_0 \circledast \mathrm{h}_1 \quad \mathbf{dom}(\mathrm{h}_1) = \{\mathrm{a} + \mathtt{f} \mid \mathtt{f} \in \mathbb{F}\}}{[\![\mathbf{free}(loc)]\!](\mathrm{e},\mathrm{h}) \Downarrow (\mathrm{e}, \mathrm{h}_0)}$$

$$\frac{\mathscr{E}[\![exp]\!](\mathrm{e},\mathrm{h}) \neq 0}{[\![\mathbf{if}\,(exp)\,p_1\,\mathbf{else}\,p_2]\!](\mathrm{e},\mathrm{h}) \Downarrow [\![p_1]\!](\mathrm{e},\mathrm{h})} \qquad \frac{\mathscr{E}[\![exp]\!](\mathrm{e},\mathrm{h}) = 0}{[\![\mathbf{if}\,(exp)\,p_1\,\mathbf{else}\,p_2]\!](\mathrm{e},\mathrm{h}) \Downarrow [\![p_2]\!](\mathrm{e},\mathrm{h})}$$

$$\frac{\mathscr{E}[\![exp]\!](\mathrm{e},\mathrm{h}) \neq 0 \quad [\![p]\!](\mathrm{e},\mathrm{h}) \Downarrow (\mathrm{e}',\mathrm{h}')}{[\![\mathbf{while}\,(exp)\,p]\!](\mathrm{e},\mathrm{h}) \Downarrow [\![\mathbf{while}\,(exp)\,p]\!](\mathrm{e}',\mathrm{h}')} \qquad \frac{\mathscr{E}[\![exp]\!](\mathrm{e},\mathrm{h}) = 0}{[\![\mathbf{while}\,(exp)\,p]\!](\mathrm{e},\mathrm{h}) \Downarrow (\mathrm{e},\mathrm{h})}$$

Fig. 7: Concrete big-step semantics for programs

We let the *domain* of h, noted $\mathbf{dom}(\mathrm{h})$, be the set of addresses at which it is defined. For example, the domain for the concrete heap $[\mathrm{a}_1 \mapsto \mathrm{v}_1; \mathrm{a}_2 \mapsto \mathrm{v}_2; \mathrm{a}_3 \mapsto \mathrm{v}_3]$ is $\{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3\}$. Additionally, if h_0 and h_1 are two concrete heaps such that $\mathbf{dom}(\mathrm{h}_0) \cap \mathbf{dom}(\mathrm{h}_1) = \emptyset$, we let $\mathrm{h}_0 \circledast \mathrm{h}_1$ denote the concrete heap obtained by merging h_0 and h_1 (its domain is $\mathbf{dom}(\mathrm{h}_0) \cup \mathbf{dom}(\mathrm{h}_1)$). We also denote the *null pointer* by $\mathbf{0x0}$ such that $\mathbf{0x0} \in \mathbb{V} \wedge \mathbf{0x0} \notin \mathbb{A}$. That is, the null pointer can be considered like a value, but not like an address.

Let $\mathbb{X}$ be the set of program variables. A *concrete environment* $\mathrm{e} \in \mathbb{E} = [\mathbb{X} \to \mathbb{A}]$ binds each program variable $\mathtt{x} \in \mathbb{X}$ to its numerical address $\mathrm{a} \in \mathbb{A}$. Thus, an environment indicates the address of a variable in the heap.

A *concrete memory state* $\mathrm{m} \in \mathbb{M} = \mathbb{E} \times \mathbb{H}$ is simply a pair made of a concrete environment and a concrete heap. Note that we do not use a stack. Consequently, the addresses of variables are also allocated in the heap.

3.3 Concrete Program Semantics

Big-step operational semantics. We assume that the semantics of a program $p \in \mathtt{P}$ is defined by a function $[\![p]\!]$ that maps its input memory state into its set of reachable outputs memory states (thus $[\![p]\!] : \mathbb{M} \to \mathscr{P}(\mathbb{M})$).

The semantics of locations and expressions are also defined by two functions, $\mathscr{L}[\![loc]\!] : \mathbb{M} \to \mathbb{A}$ and $\mathscr{E}[\![exp]\!] : \mathbb{M} \to \mathbb{V}$, respectively from memory states into addresses and from memory

Program Syntax

Name	Set	Element	Evaluation
Field	$\mathbb{F}$	$\mathtt{f}, 0$	
Location	$\mathtt{L}$	loc	$\mathscr{L}[\![loc]\!] : \mathbb{M} \to \mathbb{A}$
Expression	$\mathtt{E}$	exp	$\mathscr{E}[\![exp]\!] : \mathbb{M} \to \mathbb{V}$
Program	$\mathtt{P}$	p	$[\![p]\!] : \mathbb{M} \to \mathscr{P}(\mathbb{M})$

Memory State

Name	Set	Element	Property
Value	$\mathbb{V}$	v	
Address	$\mathbb{A}$	a	$\mathbb{A} \subseteq \mathbb{V}$
Variable	$\mathbb{X}$	$\mathtt{x}$	
Heap	$\mathbb{H}$	h	$\mathbb{H} = [\mathbb{A} \rightharpoonup \mathbb{V}]$
Environment	$\mathbb{E}$	e	$\mathbb{E} = [\mathbb{X} \to \mathbb{A}]$
Memory	$\mathbb{M}$	m	$\mathbb{M} = \mathbb{E} \times \mathbb{H}$

Table 1: Notations for the concrete semantics

states into values. They are mutually defined by induction on the structures of locations and expressions, as shown in Figure 6. The environment provides the address of the variable $\mathtt{x}$ for $\mathscr{L}[\![\mathtt{x}]\!](\mathrm{e},\mathrm{h})$ whereas $\mathscr{E}[\![loc]\!](\mathrm{e},\mathrm{h})$ first evaluates the address of the location loc, then returns the value contained in h at this address.

Figure 7 describes the evaluations of programs for assignment, conditionals, loops, allocation and deallocation. An important feature with our program semantics is that we should consider that the set of addresses $\mathbb{A}$ is *infinite*, and that every allocation generates fresh addresses. In a real imperative programming language like C, the same address can be deallocated, then allocated randomly (if it is available) during the execution of a program. This feature does not allow to express that some parts of memory have been freshly allocated or deallocated between two program points whereas these are exactly the kind of properties that our work attempts to prove. Therefore, we admit for sake of simplicity that when a program needs to allocate a memory region, fresh addresses and values are generated. Note: the actual behavior of C programs could be modeled by considering addresses as pairs of (numerical address, number of allocations in the program). As this makes the formalization heavier, we have chosen to simplify it by fresh allocations.

Table 1 summarizes the notations for the concrete semantics of this language.

Concrete Program Relational Semantics. Given a program $p \in \mathtt{P}$, we define its *relational semantics* $[\![p]\!]_{\mathscr{R}} \in \mathscr{P}(\mathbb{M} \times \mathbb{M})$ by:

$$[\![p]\!]_{\mathscr{R}} = \{(\mathrm{m_{in}}, \mathrm{m_{out}}) \mid \mathrm{m_{in}} \in \mathbb{M} \wedge \mathrm{m_{out}} \in [\![p]\!](\mathrm{m_{in}})\}$$

We observe that $[\![p]\!]_{\mathscr{R}}$ describes the set of pairs made of an input memory state $\mathrm{m_{in}}$ and an output memory state $\mathrm{m_{out}}$, where $\mathrm{m_{out}}$ is obtained by executing the program p from $\mathrm{m_{in}}$.

In the following, we define an analysis to compute an over-approximation of $[\![p]\!]_{\mathscr{R}}$.

4 Abstraction

In this section, we first define *abstract heaps*, that describe sets of memory heaps (as in [10]). We then set up *abstract heap relations* that describe binary relations over memory heaps. We also extend abstract heap relations with *abstract heap transformation predicates* to express more

Name	Set	Element	Abstracts
Symbolic Values	$\mathbb{V}^\sharp$	$\alpha, \beta, \delta, \ldots$	$\mathbb{V}$
Pure Formulas	$\mathbb{P}^\sharp$	$p^\sharp$	$\mathscr{P}(\mathbb{V} \times [\mathbb{V}^\sharp \to \mathbb{V}])$
Inductive Predicates	$\mathbb{I}\mathrm{nd}^\sharp$	**ind**	$\mathscr{P}(\mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$
Abstract Heaps	$\mathbb{H}^\sharp$	$h^\sharp$	$\mathscr{P}(\mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$
Abstract Heap Relations	$\mathbb{R}^\sharp$	$r^\sharp$	$\mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$
Abs. Heap Transformation Predicates	$\mathbb{T}^\sharp$	$t^\sharp$	$\mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$
Numerical Abstract Domains	$\mathbb{N}^\sharp$	$n^\sharp$	$\mathscr{P}([\mathbb{V}^\sharp \to \mathbb{V}])$
Abstract Environments	$\mathbb{E}^\sharp$	$e^\sharp$	$\mathscr{P}(\mathbb{E})$
Abstract Memory Relations	$\mathbb{M}^\sharp_{\mathscr{R}}$	$m^\sharp_{\mathscr{R}}$	$\mathscr{P}(\mathbb{M} \times \mathbb{M})$
Abstract Disjunction	$\mathbb{D}^\sharp$	$d^\sharp$	$\mathscr{P}(\mathbb{M} \times \mathbb{M})$

Table 2: Notations for the abstract domains and meta-variables used to denote an element of the corresponding domain.

$$
\begin{array}{rcll}
h^\sharp\ (\in \mathbb{H}^\sharp) & ::= & \mathbf{emp} & \\
& \mid & \alpha \cdot \mathtt{f} \mapsto \beta & (\alpha, \beta \in \mathbb{V}^\sharp; \mathtt{f} \in \mathbb{F}) \\
& \mid & h^\sharp_1 *_{\mathtt{S}} h^\sharp_2 & (h^\sharp_1, h^\sharp_2 \in \mathbb{H}^\sharp) \\
& \mid & \mathbf{ind} & (\mathbf{ind} \in \mathbb{I}\mathrm{nd}^\sharp)
\end{array}
$$

(a) Abstract heaps

$$
\begin{array}{rcll}
r^\sharp\ (\in \mathbb{R}^\sharp) & ::= & \mathtt{Id}(h^\sharp) & (h^\sharp \in \mathbb{H}^\sharp) \\
& \mid & [h^\sharp_i \dashrightarrow h^\sharp_o]_{t^\sharp} & (h^\sharp_i, h^\sharp_o \in \mathbb{H}^\sharp, t^\sharp \in \mathbb{T}^\sharp) \\
& \mid & r^\sharp_1 *_{\mathtt{R}} r^\sharp_2 & (r^\sharp_1, r^\sharp_2 \in \mathbb{R}^\sharp)
\end{array}
$$

(b) Abstract heap relations

Fig. 8: Syntax of abstract heaps and abstract heap relations

$$
\begin{array}{rcll}
p^\sharp\ (\in \mathbb{P}^\sharp) & ::= & \alpha & (\alpha \in \mathbb{V}^\sharp) \\
& \mid & v & (v \in \mathbb{V}) \\
& \mid & p^\sharp_1 \oplus p^\sharp_2 & (p^\sharp_1, p^\sharp_2 \in \mathbb{P}^\sharp) \\
& \mid & \mathbf{not}(p^\sharp) & (p^\sharp \in \mathbb{P}^\sharp) \\
& \mid & \mathbf{true} & \\
& \mid & \mathbf{false} & \\
\oplus & ::= & + \mid - \mid = \mid \neq, \ldots &
\end{array}
$$

Fig. 9: Pure Formulas Syntax.The $\oplus$ operators correspond to those in Figure 5.

precise relations between heaps. Finally we define *abstract memory relations* that describe relations between two memory states.

4.1 Abstract Heaps

4.1.1 An exact heap abstraction based on separation logic.

We first consider an *exact heap abstraction* without unbounded dynamic data structures, that is, an abstraction of a finite number of memory cells. We assume a countable set $\mathbb{V}^\sharp = \{\alpha, \beta, \delta, \ldots\}$

of *symbolic values* that abstract concrete addresses and values. An *abstract heap* $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$ is a separating conjunction of region predicates that abstract *separate* memory regions [39] (as mentioned above, separating conjunction is denoted by $*_\mathrm{S}$). Thus we write $\mathrm{h}_1^\sharp *_\mathrm{S} \mathrm{h}_2^\sharp$ the abstract heap that can be split into the two independent sub-abstract heaps $\mathrm{h}_1^\sharp$ and $\mathrm{h}_2^\sharp$. An individual cell is abstracted by an exact *points-to* predicate $\alpha \cdot \mathtt{f} \mapsto \beta$ where the memory cell at the address abstracted by α with the offset `f` contains the value abstracted by β. We note $\alpha \mapsto \beta$ as syntactic sugar of $\alpha \cdot \mathtt{0} \mapsto \beta$ (`0` is the null offset). We also use **emp** to describe an empty memory region.

We now define the meaning of exact abstract heaps using a *concretization function* [17], that associates abstract elements to the set of concrete elements that they describe. To concretize an abstract heap, we first need a *valuation* $\nu : \mathbb{V}^\sharp \to \mathbb{V}$, a function that defines how symbolic values are bound to concrete values and addresses. We remind that $\mathrm{h}_1 \circledast \mathrm{h}_2$ denotes the concrete heap obtained by merging h_1 and h_2 when $\mathbf{dom}(\mathrm{h}_1) \cap \mathbf{dom}(\mathrm{h}_2) = \emptyset$.

Definition 1 (Concretization of exact abstract heaps) The concretization function $\gamma_{\mathbb{H}^\sharp} : \mathbb{H}^\sharp \to \mathscr{P}(\mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ maps an abstract heap into a set of pairs made of a concrete heap and a valuation. It is defined by induction on the structure of abstract heaps as follows:

$$\begin{aligned}
&\gamma_{\mathbb{H}^\sharp}(\mathbf{emp}) = \{([], \nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\} \\
&\gamma_{\mathbb{H}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta) = \{([\nu(\alpha) + \mathtt{f} \mapsto \nu(\beta)], \nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\} \\
&\gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp *_\mathrm{S} \mathrm{h}_2^\sharp) = \{(\mathrm{h}_1 \circledast \mathrm{h}_2, \nu) \mid (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp) \land (\mathrm{h}_2, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_2^\sharp)\}
\end{aligned}$$

Example 1 (Exact abstract heap) The following exact abstract heap

$$\alpha_0 \mapsto \alpha_1 *_\mathrm{S} \alpha_1 \cdot \mathtt{data} \mapsto \alpha_2 *_\mathrm{S} \alpha_1 \cdot \mathtt{next} \mapsto \alpha_3 *_\mathrm{S} \beta_0 \mapsto \beta_1$$

describes a possible input heap for the insert function of Figure 1, where the address of `l` is α_0 and `l` points to a list of one element, and the address of `v` is β_0.

4.1.2 Heap abstraction with summarization.

This exact abstraction of memory cells does not allow to describe all the states of unbounded dynamic data structures, such as singly linked list or trees. Thus, abstract heaps should be extended with *inductive predicates* to *summarize* memory regions of unbounded size. An inductive predicate $\mathbf{ind} \in \mathbb{Ind}^\sharp$ is defined by a finite set of *rules*. Each rule is a pair made of an abstract heap $\mathrm{h}^\sharp$ and a *pure formula* $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$ that describes numerical constraints over symbolic values (although, in this paper we do not consider relational numerical constraints—in general the issue of extending a shape abstraction with numerical reasoning is non trivial and orthogonal to the scope of this paper). Figure 8a describes the syntax that defines abstract heaps while Figure 9 describes the syntax of pure formulas. The $\oplus$ operators of pure formulas are exactly the same as in Figure 5.

Example 2 (List inductive predicate) In the rest of the paper, we assume that `data` and `next` denote two fields corresponding to distinct offsets. The **list** predicate describes the structure of a singly linked list and is defined by induction as follows:

$$\begin{aligned}
\mathbf{list}(\alpha) ::= \{&(\mathbf{emp}, \alpha = \mathbf{0x0}), \\
&(\alpha \cdot \mathtt{data} \mapsto \delta *_\mathrm{S} \alpha \cdot \mathtt{next} \mapsto \beta *_\mathrm{S} \mathbf{list}(\beta), \alpha \neq \mathbf{0x0})\}
\end{aligned}$$

The first rule of that definition corresponds to the empty list, so that α is the null pointer. The second rule describes the case where the list contains one element and points to an other list (α cannot be the null pointer).

Example 3 (Binary tree inductive predicate) We can define similarly the **tree** predicate to describe the structure of a binary tree. We assume that `e`, `l`, and `r` denote three fields corresponding to distinct offsets. We also can enrich the numerical constraint of the second rule to specify that all the elements of the tree are strictly positive:

$$\begin{aligned}\mathbf{tree}(\alpha) ::= \{&(\mathbf{emp}, \alpha = \mathbf{0x0}),\\ &(\alpha \cdot \mathtt{e} \mapsto \delta *_{\mathrm{S}} \alpha \cdot \mathtt{l} \mapsto \beta_{\mathtt{l}} *_{\mathrm{S}} \alpha \cdot \mathtt{r} \mapsto \beta_{\mathtt{r}}\\ &\ *_{\mathrm{S}} \mathbf{tree}(\beta_{\mathtt{l}}) *_{\mathrm{S}} \mathbf{tree}(\beta_{\mathtt{r}}), \alpha \neq \mathbf{0x0} \wedge \delta > 0)\}\end{aligned}$$

Generally, inductive predicates such as **list** or **tree** summarize a memory region from a specific address. However, we often need to describe properties on different parts of a whole summarized memory region and inductive predicates are not precise enough to do that. For instance, in Figure 1, we need to express that that the insertion of the new element occurs somewhere at least after the first element of the list. Such property require a *segment predicate* to be expressed. Segment predicates summarize a memory region between two addresses and permit to split a whole summarized region into many contiguous summarized sub-regions. In this paper, for the sake of simplicity, we consider segment predicates as inductive predicates with an additional parameter that specifies the ending address of the summarized region. For example, the list segment predicate **listseg** is defined by induction as follows:

$$\begin{aligned}\mathbf{listseg}(\alpha,\beta) ::= \{&(\mathbf{emp}, \alpha = \beta),\\ &(\alpha \cdot \mathtt{data} \mapsto \delta *_{\mathrm{S}} \alpha \cdot \mathtt{next} \mapsto \gamma *_{\mathrm{S}} \mathbf{listseg}(\gamma,\beta), \alpha \neq \mathbf{0x0} \wedge \alpha \neq \beta)\}\end{aligned}$$

Thus, the **listseg** predicate denotes a list of any length starting at α and ending at β (when the list is empty, we have $\alpha = \beta$).

The concretization of inductive predicates requires a function $\Delta : \mathbb{Ind}^\sharp \to \mathscr{P}_{\mathrm{fin}}(\mathbb{H}^\sharp \times \mathbb{P}^\sharp)$ that maps an inductive predicate into the finite set of rules of its definition.

Definition 2 (Concretization of inductive predicates) The concretization function $\gamma_{\mathbb{P}^\sharp} : \mathbb{P}^\sharp \to \mathscr{P}(\mathbb{V} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ maps a pure formula into a set of pairs made of its concrete value and a valuation whereas $\gamma_\Sigma : \mathbb{H}^\sharp \times \mathbb{P}^\sharp \to \mathscr{P}(\mathbb{H}^\sharp \times [\mathbb{V}^\sharp \to \mathbb{V}])$ concretizes pairs of abstract heap and pure formula. The concretization of inductive predicates extends the concretization of abstract heaps $\gamma_{\mathbb{H}^\sharp}$:

$$\begin{aligned}\gamma_{\mathbb{P}^\sharp}(\alpha) &= \{(\nu(\alpha),\nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\}\\ \gamma_{\mathbb{P}^\sharp}(\mathrm{v}) &= \{(\mathrm{v},\nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\}\\ \gamma_{\mathbb{P}^\sharp}(\mathrm{p}_1^\sharp \oplus \mathrm{p}_2^\sharp) &= \{(\mathrm{v}_1 \oplus \mathrm{v}_2,\nu) \mid (\mathrm{v}_1,\nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}_1^\sharp) \wedge (\mathrm{v}_2,\nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}_2^\sharp)\}\\ \gamma_{\mathbb{P}^\sharp}(\mathbf{not}(\mathrm{p}^\sharp)) &= \{(\mathrm{v},\nu) \mid (0,\nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\}\\ &\quad \cup \{(0,\nu) \mid \exists \mathrm{v}, (\mathrm{v},\nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\}\\ \gamma_{\mathbb{P}^\sharp}(\mathbf{true}) &= \{(\mathrm{v},\nu) \mid \mathrm{v} \neq 0 \wedge \nu : \mathbb{V}^\sharp \to \mathbb{V}\}\\ \gamma_{\mathbb{P}^\sharp}(\mathbf{false}) &= \{(0,\nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\}\\ \\ \gamma_\Sigma(\mathrm{h}^\sharp,\mathrm{p}^\sharp) &= \{(\mathrm{h},\nu) \mid (\mathrm{h},\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \wedge \exists \mathrm{v}, (\mathrm{v},\nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\}\\ \\ \gamma_{\mathbb{H}^\sharp}(\mathbf{ind}) &= \bigcup_{(\mathrm{h}^\sharp,\mathrm{p}^\sharp) \in \Delta(\mathbf{ind})} \gamma_\Sigma(\mathrm{h}^\sharp,\mathrm{p}^\sharp)\end{aligned}$$

We remark that abstract states may contain incompatible constraints such as $\alpha = \mathbf{0x0}$ and $\alpha \mapsto \beta$ (the address of any memory cell is non null); the above concretization function naturally maps such abstract states into the empty set.

Note that the rules of inductive predicates should not contain unsatisfiable pure formulas. Indeed such a rule would describe the empty set of states and adding it would only clutter the inductive predicate. Still it would not pose any problem to the analysis, as an unsatisfiable rule would never be unfolded or folded. In the rest of the paper we pay attention to never write such inconsistent inductive predicates.

Remark 1 (Value in the concretization of pure formulas) Let $p^\sharp \in \mathbb{P}^\sharp$ a pure formula and $(v, \nu) \in \gamma_{\mathbb{P}^\sharp}(p^\sharp)$. If $v = 0$, that means that $p^\sharp$ is not a satisfiable pure formula (0 means false in the concrete semantics). As an example, consider that $p^\sharp = (\alpha = 1) \wedge (\alpha = 2)$. It is obvious that $p^\sharp$ cannot be satisfied, its concretization is thus $\{(0, \nu) \mid \nu : \mathbb{V}^\sharp \to \mathbb{V}\}$. Therefore, we can express that a pure formula $p^\sharp$ is satisfiable using the constraint $v \neq 0$ if $(v, \nu) \in \gamma_{\mathbb{P}^\sharp}(p^\sharp)$, like we did in the definition of γ_Σ in Definition 2.

Example 4 (Abstract heap with a summarized region) The following abstract heap

$$\alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \mathbf{list}(\alpha_1)$$

describes all the possible input heaps for the sort function of Figure 3, where the address of `l` is α_0 and `l` points to a list of unknown length.

4.2 Abstract Heap Relations

As explained in Section 2, two abstract heaps at different program points cannot describe relations (they describe two independent sets of concrete memory heaps). Thus, we propose *abstract heap relations* that describe a set of pairs made of an *input* heap h_i and an *output* heap h_o. Abstract heap relations are defined by new logical connectives over abstract heaps (syntax is given in Figure 8b) as follows:

- the *identity relation* $\mathtt{Id}(h^\sharp)$ describes pairs of heaps that are equal and both abstracted by $h^\sharp$.
- the *transform-into relation* $[h_i^\sharp \dashrightarrow h_o^\sharp]$ describes pairs corresponding to the transformation of a heap abstracted by $h_i^\sharp$ into a heap abstracted by $h_o^\sharp$. Note that we will only consider the $t^\sharp$ symbol that appears in Figure 8b from Section 4.3, for the sake of simplicity.
- the *relational separating conjunction* $r_1^\sharp *_{\mathtt{R}} r_2^\sharp$ of two abstract heap relations $r_1^\sharp$ and $r_2^\sharp$ denotes a relation that can be described by combining independently the relations described by $r_1^\sharp$ and $r_2^\sharp$ on disjoint memory regions.

The concretization of abstract heap relations also requires using valuations as it also needs to define the concrete values that symbolic values denote.

Definition 3 (Concretization of abstract heap relations) The concretization function $\gamma_{\mathbb{R}^\sharp} \in \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ maps an abstract heap relation into a set of triples made of an input heap, an output heap and a valuation. It is defined by induction on the structure of $r^\sharp$:

$$\begin{aligned}
\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h^\sharp)) &= \{(h,\ h,\ \nu) \mid (h,\ \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp)\} \\
\gamma_{\mathbb{R}^\sharp}([h_i^\sharp \dashrightarrow h_o^\sharp]) &= \{(h_i,\ h_o,\ \nu) \mid (h_i,\ \nu) \in \gamma_{\mathbb{H}^\sharp}(h_i^\sharp) \wedge (h_o,\ \nu) \in \gamma_{\mathbb{H}^\sharp}(h_o^\sharp)\} \\
\gamma_{\mathbb{R}^\sharp}(r_1^\sharp *_{\mathtt{R}} r_2^\sharp) &= \{(h_{i,1} \circledast h_{i,2},\ h_{o,1} \circledast h_{o,2},\ \nu) \mid \\
&\qquad (h_{i,1},\ h_{o,1},\ \nu) \in \gamma_{\mathbb{R}^\sharp}(r_1^\sharp) \wedge \mathbf{dom}(h_{i,1}) \cap \mathbf{dom}(h_{o,2}) = \emptyset \\
&\qquad \wedge (h_{i,2},\ h_{o,2},\ \nu) \in \gamma_{\mathbb{R}^\sharp}(r_2^\sharp) \wedge \mathbf{dom}(h_{i,2}) \cap \mathbf{dom}(h_{o,1}) = \emptyset\}
\end{aligned}$$

We remark that $*_{\mathtt{R}}$ is commutative and associative. We can also define the neutral element $\mathbf{emp}_{\mathtt{R}}$ for $*_{\mathtt{R}}$ that is syntactic sugar for $\mathtt{Id}(\mathbf{emp})$ and $[\mathbf{emp} \dashrightarrow \mathbf{emp}]$: we have $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathbf{emp})) = \gamma_{\mathbb{R}^\sharp}([\mathbf{emp} \dashrightarrow \mathbf{emp}])$.

Example 5 (Expressiveness 1) Let $\mathrm{r}_1^\sharp = \mathtt{Id}(\mathbf{list}(\alpha))$ and $\mathrm{r}_2^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]$. We observe that $\mathrm{r}_1^\sharp$ describes the identity relation applied to a well-formed linked list starting from the address α, whereas $\mathrm{r}_2^\sharp$ describes any transformation that inputs such a list and 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 α). This means that $\gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \subset \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_2^\sharp)$.

Example 6 (Expressiveness 2) Let $\mathrm{r}_1^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{emp}] *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \mathbf{list}(\beta)]$ and $\mathrm{r}_2^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$. The abstract heap relation $\mathrm{r}_1^\sharp$ describes two distinct transformations: the first one expresses the deallocation of a list starting from the address α and the second one expresses the allocation of a list starting from the address β. The abstract heap relation $\mathrm{r}_2^\sharp$ describes simply the transformation of a list starting from α into a list starting from β. In $\mathrm{r}_1^\sharp$, we know that the two lists are physically different but in $\mathrm{r}_2^\sharp$, we have no information about the output list is obtained from the input list. This means that the two lists can be either physically different, can share some memory cells, or can even be totally equal. Actually, we have $\gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \subset \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_2^\sharp)$.

More generally, we have the following properties:

Theorem 1 (Properties on abstract heap relations)
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. $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$
2. $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$ *(the opposite inclusion may not hold, as observed in Example 5);*
3. $\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp] *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]) \subseteq \gamma_{\mathbb{R}^\sharp}([(\mathrm{h}_{\mathrm{i},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{i},1}^\sharp) \dashrightarrow (\mathrm{h}_{\mathrm{o},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{o},1}^\sharp)])$ *(the opposite inclusion may not hold, as observed in Example 6).*

Property 1 allows to split and merge identity relations as we like. Property 2 allows to loose the identity relation on a heap, weakening it into a transform-into relation whereas Property 3 allows to forget that two relations are independent by merging respectively their inputs and output states. These properties will be useful in our static analysis described in Section 5 to compute new abstract heap relations.

Proof (Proof of Theorem 1) We are now going to prove Theorem 1. For each property, we just reduce both concretizations using their definition to show that the equality (for Property 1) or the inclusion (for Property 2 and Property 3) holds:

1. Proof of $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$:

$$\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) \\
&\quad = \{(\mathrm{h}, \mathrm{h}, \nu) \mid (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)\} \\
&\quad = \{(\mathrm{h}_0 \circledast \mathrm{h}_1, \mathrm{h}_0 \circledast \mathrm{h}_1, \nu) \mid (\mathrm{h}_0, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \wedge (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp)\}
\end{aligned}$$

$$\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp)) \\
&\quad = \{(\mathrm{h}_0 \circledast \mathrm{h}_1, \mathrm{h}_0' \circledast \mathrm{h}_1', \nu) \mid \\
&\qquad (\mathrm{h}_0, \mathrm{h}_0', \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp)) \wedge (\mathrm{h}_1, \mathrm{h}_1', \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_1^\sharp)) \wedge \\
&\qquad \mathbf{dom}(\mathrm{h}_0) \cap \mathbf{dom}(\mathrm{h}_1') = \emptyset \wedge \mathbf{dom}(\mathrm{h}_1) \cap \mathbf{dom}(\mathrm{h}_0') = \emptyset\} \\
&\quad = \{(\mathrm{h}_0 \circledast \mathrm{h}_1, \mathrm{h}_0 \circledast \mathrm{h}_1, \nu) \mid (\mathrm{h}_0, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \wedge (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp)\}\}
\end{aligned}$$

So $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h_0^\sharp *_\mathtt{S} h_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h_0^\sharp) *_\mathtt{R} \mathtt{Id}(h_1^\sharp))$

2. Proof of $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([h^\sharp \dashrightarrow h^\sharp])$:

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h^\sharp))\\&\quad = \{(h,h,\nu) \mid (h,\nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp)\}\end{aligned}$$

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}([h^\sharp \dashrightarrow h^\sharp])\\&\quad = \{(h_0,h_1,\nu) \mid (h_0,\nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp) \wedge (h_1,\nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp)\}\end{aligned}$$

We clearly have: $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([h^\sharp \dashrightarrow h^\sharp])$

3. Proof of $\gamma_{\mathbb{R}^\sharp}([h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp] *_\mathtt{R} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]) \subseteq \gamma_{\mathbb{R}^\sharp}([(h_{i,0}^\sharp *_\mathtt{S} h_{i,1}^\sharp) \dashrightarrow (h_{o,0}^\sharp *_\mathtt{S} h_{o,1}^\sharp)])$:

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}([h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp] *_\mathtt{R} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp])\\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid\\
&\qquad (h_{i,0},h_{o,0},\nu) \in \gamma_{\mathbb{R}^\sharp}([h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp]) \wedge\\
&\qquad (h_{i,1},h_{o,1},\nu) \in \gamma_{\mathbb{R}^\sharp}([h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]) \wedge\\
&\qquad \mathbf{dom}(h_{i,0}) \cap \mathbf{dom}(h_{o,1}) = \emptyset \wedge \mathbf{dom}(h_{i,1}) \cap \mathbf{dom}(h_{o,0}) = \emptyset\}\\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid\\
&\qquad (h_{i,0},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{i,0}^\sharp) \wedge (h_{o,0},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{o,0}^\sharp) \wedge\\
&\qquad (h_{i,1},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{i,1}^\sharp) \wedge (h_{o,1},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{o,1}^\sharp) \wedge\\
&\qquad \mathbf{dom}(h_{i,0}) \cap \mathbf{dom}(h_{o,1}) = \emptyset \wedge \mathbf{dom}(h_{i,1}) \cap \mathbf{dom}(h_{o,0}) = \emptyset\}\end{aligned}$$

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}([(h_{i,0}^\sharp *_\mathtt{S} h_{i,1}^\sharp) \dashrightarrow (h_{o,0}^\sharp *_\mathtt{S} h_{o,1}^\sharp)])\\
&\quad = \{(h_i,h_o,\nu) \mid\\
&\qquad (h_i,\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{i,0}^\sharp *_\mathtt{S} h_{i,1}^\sharp) \wedge (h_o,\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{o,0}^\sharp *_\mathtt{S} h_{o,1}^\sharp)\}\\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid\\
&\qquad (h_{i,0},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{i,0}^\sharp) \wedge (h_{o,0},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{o,0}^\sharp) \wedge\\
&\qquad (h_{i,1},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{i,1}^\sharp) \wedge (h_{o,1},\nu) \in \gamma_{\mathbb{H}^\sharp}(h_{o,1}^\sharp)\}\end{aligned}$$

So finally:

$$\gamma_{\mathbb{R}^\sharp}([h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp] *_\mathtt{R} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]) \subseteq \gamma_{\mathbb{R}^\sharp}([h_{i,0}^\sharp *_\mathtt{S} h_{i,1}^\sharp \dashrightarrow h_{o,0}^\sharp *_\mathtt{S} h_{o,1}^\sharp])$$

4.3 Abstract Heap Transformation Predicates

From now, we have two relational connectives that describe heap transformations: the identity relation and the transform-into relation. While the identity relation is very strong (it ensures that the heap is left unmodified), the transform-into relation is quite weak. Indeed, it just indicates that the input heap abstracted by $h_i^\sharp$ has been transformed into the output heap abstracted by $h_o^\sharp$, but without describing specifically how the transformation occurred. For instance in Figure 3, the function `sort` performs a list sort in place, modifying only the order of its elements. A reasonable abstraction of the effects of this function is that the output list is a *permutation in place* of the input list.

Remark 2 (Properties to ensure) We remark that a permutation in place of a list can be expressed ensuring these two properties:

(a) the permutation is in-place, so that the output list is obtained by manipulating directly the input list (the footprint of the two lists is the same).
(b) the data in the input and output lists are exactly the same (but may appear in a different order).

If we look at the abstract heap relation computed for the function `sort`:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$$

We observe that this abstraction is too weak to ensure the points (a) and (b), as it describes only a transformation of a well formed linked list into a well formed linked list. More generally, to capture stronger relations, abstract heap relations should be extended.

To fix this imprecision, we could enrich abstract heap relations with a new connective that would express specifically these two points. The problem with this approach is that it would be certainly useless to describe the behavior of other functions using other data structures. Moreover, creating a relational connective in abstract heap relations to express a specific property when needed is too costly. Indeed, this requires to update all the functions related to abstract heap relations for each new relational connective.

An elegant and efficient approach to describe any binary relational properties between memory heaps without adding new connectives is to annotate transform-into relations by *abstract heap transformation predicates*. We henceforth write $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}$ for the transform-into relation annotated by the abstract heap transformation predicate $\mathrm{t}^\sharp$. It describes the transformation abstracted by $\mathrm{t}^\sharp$ of the heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ into the heaps abstracted by $\mathrm{h}_\mathrm{o}^\sharp$.

We name an *abstract heap transformation predicates domain* a set $\mathbb{T}^\sharp$ of abstract heap transformation predicates. To be as generic as possible, we do not fix a specific abstract heap transformation predicates domain $\mathbb{T}^\sharp$. Instead, we parametrize the analysis with an interface of such $\mathbb{T}^\sharp$.

The concretization function $\gamma_{\mathbb{T}^\sharp}$ of an abstract heap transformation predicates domain $\mathbb{T}^\sharp$ should have the signature:

Condition 1 (Concretization) *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain. Its concretization should have the following signature:*

$$\gamma_{\mathbb{T}^\sharp} : \mathbb{T}^\sharp \to \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$$

It simply maps an abstract heap transformation predicate into a set of triples made of an input concrete heap, an output concrete heap and a valuation function.

We can now update the concretization function $\gamma_{\mathbb{R}^\sharp}$ defined in Definition 3 (page 13) of abstract heap relations to take into account transform-into relations annotated by the abstract heap transformation predicates.

Definition 4 (Concretization of annotated transform-into relations) Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}^\sharp \in \mathbb{T}^\sharp, \mathrm{h}_\mathrm{i}^\sharp, \mathrm{h}_\mathrm{o}^\sharp \in \mathbb{H}^\sharp$. Then, the concretization of transform-into relations annotated by abstract heap transformation predicates is defined as follows:

$$\begin{aligned}\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}) = \{(\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \mid\ & (\mathrm{h}_\mathrm{i}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_\mathrm{i}^\sharp) \wedge (\mathrm{h}_\mathrm{o}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_\mathrm{o}^\sharp) \\ & \wedge (\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)\}\end{aligned}$$

Like abstract heap relations, abstract heap transformation predicates can express identity (resp. independent transformations).

To that end, each abstract heap transformation predicates domain $\mathbb{T}^\sharp$ should define the identity $\mathbf{id}_{\mathbb{T}^\sharp} : \mathbb{H}^\sharp \to \mathbb{T}^\sharp$ (resp. the separating conjunction operator $*_{\mathtt{T}} : \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$), for abstract heap transformation predicates. These operators should satisfy the following conditions:

Condition 2 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$) *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain, $\mathrm{t}^\sharp \in \mathbb{T}^\sharp$ and $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$. Then $\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$ is sound if:*

$$\{(\mathrm{h},\mathrm{h},\nu) \mid (\mathrm{h},\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\} \subseteq \gamma_{\mathbb{T}^\sharp}(\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp))$$

Condition 3 (Soundness of $*_{\mathtt{T}}$) *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$. Then $\mathrm{t}_0^\sharp *_{\mathtt{T}} \mathrm{t}_1^\sharp$ is sound if:*

$$\gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp \ *_{\mathtt{T}} \ \mathrm{t}_1^\sharp) \supseteq \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid (\mathrm{h}_{\mathrm{i},0}, \mathrm{h}_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \wedge (\mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp)\}$$

Using these two operators, we can define new properties, similar to Theorem 1 (page 14), on abstract heap relations taking into account abstract heap transformation predicates.

Theorem 2 (Properties on abstract heap relations with abstract heap transformation predicates)
Let $\mathrm{h}^\sharp$, $\mathrm{h}_{\mathrm{i},0}^\sharp$, $\mathrm{h}_{\mathrm{i},1}^\sharp$, $\mathrm{h}_{\mathrm{o},0}^\sharp$, $\mathrm{h}_{\mathrm{o},1}^\sharp$ be abstract heaps, $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}^\sharp, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$. Then, we have the following properties:

1. $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})$ *with* $\mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$
2. $\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp}) \subseteq \gamma_{\mathbb{R}^\sharp}([(\mathrm{h}_{\mathrm{i},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{i},1}^\sharp) \dashrightarrow (\mathrm{h}_{\mathrm{o},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{o},1}^\sharp)]_{\mathrm{t}^\sharp})$ *with* $\mathrm{t}^\sharp = \mathrm{t}_0^\sharp \ *_{\mathtt{T}} \ \mathrm{t}_1^\sharp$

We observe with the property *1.* that the identity of abstract heap transformation predicates may be less precise than the identity relation of abstract heap relations.

Proof (Proof of Theorem 2) To prove Theorem 2, we use the proof of the properties 2 and 3 of Theorem 1.

1. Proof of $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})$ with $\mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$:
   We can prove this property exactly like we proved the property 2 of Theorem 1.

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp))\\ &\quad = \{(\mathrm{h},\mathrm{h},\nu) \mid (\mathrm{h},\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\}\end{aligned}$$

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})\\ &\quad = \{(\mathrm{h}_0,\mathrm{h}_1,\nu) \mid (\mathrm{h}_0,\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \wedge (\mathrm{h}_1,\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\\ &\qquad \wedge (\mathrm{h}_0,\mathrm{h}_1,\nu) \in \gamma_{\mathbb{T}^\sharp}(\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp))\}\end{aligned}$$

   By soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$, it is obvious that:

$$\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp}),\ \text{with } \mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$$

2. Proof of
$\gamma_{\mathbb{R}^\sharp}([h^\sharp_{i,0} \dashrightarrow h^\sharp_{o,0}]_{t^\sharp_0} *_R [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1}) \subseteq \gamma_{\mathbb{R}^\sharp}([(h^\sharp_{i,0} *_S h^\sharp_{i,1}) \dashrightarrow (h^\sharp_{o,0} *_S h^\sharp_{o,1})]_{t^\sharp})$ with $t^\sharp = t^\sharp_0 *_T t^\sharp_1$:
A major part of this property has been proven in the proof of Theorem 1 (property 3), the rest only relates to $t^\sharp_0$, $t^\sharp_1$ and $t^\sharp$. So for the sake of clarity, we write "..." for the parts of the proof that already are in the proof of Theorem 1, property 3.

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([h^\sharp_{i,0} \dashrightarrow h^\sharp_{o,0}]_{t^\sharp_0} *_R [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1}) \\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid \\
&\qquad (h^\sharp_{i,0}, h^\sharp_{o,0}, \nu) \in \gamma_{\mathbb{R}^\sharp}([h^\sharp_{i,0} \dashrightarrow h^\sharp_{o,0}]_{t^\sharp_0}) \wedge \\
&\qquad (h^\sharp_{i,1}, h^\sharp_{o,1}, \nu) \in \gamma_{\mathbb{R}^\sharp}([h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1}) \wedge \\
&\qquad \ldots\} \\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid \\
&\qquad (h^\sharp_{i,0}, h^\sharp_{o,0}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_0) \wedge \\
&\qquad (h^\sharp_{i,1}, h^\sharp_{o,1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_1) \wedge \\
&\qquad \ldots\}
\end{aligned}
$$

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([(h^\sharp_{i,0} *_S h^\sharp_{i,1}) \dashrightarrow (h^\sharp_{o,0} *_S h^\sharp_{o,1})]_{t^\sharp}) \\
&\quad = \{(h_i, h_o, \nu) \mid \\
&\qquad (h_i, \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp_{i,0} *_S h^\sharp_{i,1}) \wedge \\
&\qquad (h_o, \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp_{o,0} *_S h^\sharp_{o,1}) \wedge \\
&\qquad (h_i, h_o, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp)\} \\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid \\
&\qquad (h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp) \\
&\qquad \wedge \ldots\} \\
&\quad = \{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid \\
&\qquad (h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_0 *_T t^\sharp_1) \\
&\qquad \wedge \ldots\}
\end{aligned}
$$

By the soundness of $*_T$, we observe that:

$$
\begin{aligned}
&\{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid \\
&\qquad (h^\sharp_{i,0}, h^\sharp_{o,0}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_0) \wedge (h^\sharp_{i,1}, h^\sharp_{o,1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_1)\} \\
&\subseteq \\
&\{(h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \mid \\
&\qquad (h_{i,0} \circledast h_{i,1}, h_{o,0} \circledast h_{o,1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_0 *_T t^\sharp_1)\}
\end{aligned}
$$

So finally:

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([h^\sharp_{i,0} \dashrightarrow h^\sharp_{o,0}]_{t^\sharp_0} *_R [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1}) \\
&\subseteq \\
&\gamma_{\mathbb{R}^\sharp}([(h^\sharp_{i,0} *_S h^\sharp_{i,1}) \dashrightarrow (h^\sharp_{o,0} *_S h^\sharp_{o,1})]_{t^\sharp})
\end{aligned}
$$

In the following, we give two examples of abstract heap transformation predicates that, when combined together, are expressive enough to ensure the properties (a) and (b) of Remark 2. The first one describes relations between the sets of addresses that define the input and output heaps. The second one describes the set of fields for which the value of all cells may have been modified. Finally we define the combination of two abstract heap transformation predicates domains.

4.3.1 The Footprint Predicates Domain

Recall the property (a) of Remark 2. To ensure that the sorting algorithm operates in-place, we need to express that the set of addresses of the input and the output lists are strictly equal. To do that, we can design an abstract heap transformation predicates domain that compares the set of addresses of the input and output heaps. We name *the footprint predicates domain* the abstract heap transformation predicates domain where $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$. Each element of $\mathbb{T}^\sharp$ compares the set of addresses that defines the input heap with the set of addresses that defines the output heap. They do not provide relations between the content of the memory cells.

Let $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}$ be a transform-into relation annotated by $\mathrm{t}^\sharp$. Then:

- If $\mathrm{t}^\sharp$ is $=^\sharp$, then the input heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ and the output heaps abstracted $\mathrm{h}_\mathrm{o}^\sharp$ both span over the exact same set of addresses. Obviously, this property holds when no allocation and no deallocation takes place. More generally, it also holds for all executions where no existing cell is freed and where any cell that is allocated during the execution is freed before the end of the program is reached.
- If $\mathrm{t}^\sharp$ is $\subseteq^\sharp$ (respectively $\supseteq^\sharp$), the set of addresses of the heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ is included in (respectively includes) or is equal to the set of addresses of the heaps abstracted by $\mathrm{h}_\mathrm{o}^\sharp$. This indicates that only allocations (respectively deallocations) may have taken place, as the output heap is bigger than (respectively smaller than) or is equal to the input heap.
- Finally, if $\mathrm{t}^\sharp = \top$, then it indicates no specific relations between the input and the output heap.

Definition 5 (Concretization of the footprint predicates domain) We give a formal meaning to the set footprint predicates domain by defining its concretization function $\gamma_{\mathbb{T}^\sharp} : \mathbb{T}^\sharp \to \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$:

$$\begin{aligned}
\gamma_{\mathbb{T}^\sharp}(=^\sharp) &= \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}] \mid \mathbf{dom}(\mathrm{h_i}) = \mathbf{dom}(\mathrm{h_o})\} \\
\gamma_{\mathbb{T}^\sharp}(\subseteq^\sharp) &= \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}] \mid \mathbf{dom}(\mathrm{h_i}) \subseteq \mathbf{dom}(\mathrm{h_o})\} \\
\gamma_{\mathbb{T}^\sharp}(\supseteq^\sharp) &= \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}] \mid \mathbf{dom}(\mathrm{h_i}) \supseteq \mathbf{dom}(\mathrm{h_o})\} \\
\gamma_{\mathbb{T}^\sharp}(\top) &= \mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}]
\end{aligned}$$

Example 7 (Expressiveness) In this example, we discuss the expressiveness of the footprint predicates domain, for each value of $\mathrm{t}^\sharp$ in the following transform-into relation:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]_{\mathrm{t}^\sharp}$$

If $\mathrm{t}^\sharp$ is $=^\sharp$, then the input heap and the output heap have exactly the same footprint. When no allocation/deallocation takes place, this predicate holds (even if the order of the elements inside the list may have changed). More generally, it holds whenever no existing cell is freed and when any cell allocated during the execution is also freed during the execution. In any case, when $\mathrm{t}^\sharp$ is $=^\sharp$, the length and physical size of the list have not been modified. However, we have no information about the order or the value of each memory cell. If $\mathrm{t}^\sharp = \subseteq^\sharp$ (respectively $\mathrm{t}^\sharp = \supseteq^\sharp$), then the output list may contain more (respectively fewer) elements than the input list. Here too, any information about the order or the value of the elements. Finally, if $\mathrm{t}^\sharp = \top$, we have any interesting information about the transformation of the input list into the output list.

More generally, we have the following properties:

Lemma 1 (Properties of the footprint predicates domain) *We observe the following properties about the footprint predicates domain:*

1. $\gamma_{\mathbb{T}^\sharp}(=^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\subseteq^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\top)$
2. $\gamma_{\mathbb{T}^\sharp}(=^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\supseteq^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\top)$
3. $\gamma_{\mathbb{T}^\sharp}(\subseteq^\sharp) \cup \gamma_{\mathbb{T}^\sharp}(\supseteq^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\top)$

Proof (Proof of Lemma 1) The proof of these three properties is trivial using the definition of the concretization function. Thus we do not detail it.

We can deduce from these properties that $\top$ is the less precise element of $\mathbb{T}^\sharp$ (it provides any relation). On the contrary, $=^\sharp$ is the most precise: the set of addresses of the input and output heaps are the same but not necessarily the content of each memory cell. As well, it is relevant to express that the input heap has been manipulated in place (point (a)).

Definition 6 (Function $\mathbf{id}_{\mathbb{T}^\sharp}$ of the footprint predicates domain)

$$\text{Let } \mathrm{h}^\sharp \in \mathbb{H}^\sharp, \text{ then: } \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp) = \quad =^\sharp$$

Definition 7 (Operator $*_{\mathrm{T}}$ of the footprint predicates domain) For simplicity, we define this operator with a table as follows:

$*_{\mathrm{T}}$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$=^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$\subseteq^\sharp$	$\subseteq^\sharp$	$\subseteq^\sharp$	$\top$	$\top$
$\supseteq^\sharp$	$\supseteq^\sharp$	$\top$	$\supseteq^\sharp$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Theorem 3 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathrm{T}}$) *The functions* $\mathbf{id}_{\mathbb{T}^\sharp}$ *and* $*_{\mathrm{T}}$ *of the footprint predicates domain satisfy respectively Condition 2 and Condition 3.*

Proof (Proof of Theorem 3) The proof of this theorem is trivial. For $\mathbf{id}_{\mathbb{T}^\sharp}$, we could choose any other predicate of the domain, but the chosen one is the most precise, as expressed by Lemma 1.

4.3.2 The Fields Predicates Domain

When manipulating data structures, it is common that a function modifies *partially* the memory. For instance in Figure 3, the sort function may modify the order of the input list manipulating the `next` fields whereas the values of the `data` fields do not change. However, as data structures like linked lists are abstracted by inductive predicates that summarize memory heaps, we cannot capture the partial modification property. In our sort function, proving that only the values of the `next` fields may have been modified would prove that the values of `data` fields have not been modified. More specifically, it would ensure that all the data that are both in the input and output lists are the same.

We can express the set of structure fields whose value *may have changed* designing an abstract heap transformation predicates domain such as $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$. We name this domain *the fields predicates domain*. Let $[\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]_F$ be the transform-into relation annotated by $F \in \mathscr{P}(\mathbb{F})$. If `f` is a structure field that *is not* in F, then each field `f` of elements of the structure that is both in the input and output heaps abstracted respectively by $\mathrm{h}^\sharp_{\mathrm{i}}$ and $\mathrm{h}^\sharp_{\mathrm{o}}$ has the same value.

Definition 8 (Concretization of the fields predicates domain) Let $F \in \mathscr{P}(\mathbb{F})$. We define the concretization of the fields predicates domain $\gamma_{\mathbb{T}^\sharp} : \mathscr{P}(\mathbb{F}) \to \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ as follows:

$$\begin{aligned}\gamma_{\mathbb{T}^\sharp}(F) = \{&(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}] \mid \\ &\forall \mathtt{f} \notin F, \forall \mathrm{a} \in \mathbb{A}, (\mathrm{a} + \mathtt{f}) \in \mathbf{dom}(\mathrm{h_i}) \wedge (\mathrm{a} + \mathtt{f}) \in \mathbf{dom}(\mathrm{h_o}) \\ &\Rightarrow \mathrm{h_i}(\mathrm{a} + \mathtt{f}) = \mathrm{h_o}(\mathrm{a} + \mathtt{f})\}\end{aligned}$$

Example 8 (Expressiveness) We now discuss the expressiveness of the fields predicates domain for the following transformation-into relation:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]_{\mathrm{t}^\sharp}$$

If $\mathrm{t}^\sharp = \{\mathtt{next}\}$, then only the `next` fields of the elements of the two lists may have been modified. Consequently, all the `data` fields of the lists elements are unchanged. We observe that the lists may have a different number of elements. For instance, the output list can be the input where we deallocated randomly some elements, without modifying the value of the `data` fields.

Lemma 2 (Property of the fields predicates domain) *We observe the following property about the fields predicates domain:*

$$\forall F_1, F_2 \in \mathscr{P}(\mathbb{F}), F_1 \subseteq F_2 \Rightarrow \gamma_{\mathbb{T}^\sharp}(F_1) \subseteq \gamma_{\mathbb{T}^\sharp}(F_2)$$

Proof (Proof of Lemma 2) This proof is trivial using the definition of the concretization function.

We made the choice to use the set of fields that may have changed instead of the set of fields that have not changed for the sake of simplicity. Indeed, if we took the latest, we would have the following property: $\forall F_1, F_2 \in \mathscr{P}(\mathbb{F}), F_1 \supseteq F_2 \Rightarrow \gamma_{\mathbb{T}^\sharp}(F_1) \subseteq \gamma_{\mathbb{T}^\sharp}(F_2)$ that is not relevant and easy to manipulate.

Definition 9 (Function $\mathbf{id}_{\mathbb{T}^\sharp}$ of the fields predicates domain)

$$\text{Let } \mathrm{h}^\sharp \in \mathbb{H}^\sharp, \text{ then: } \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp) = \{\}$$

Definition 10 (Operator $*_{\mathrm{T}}$ of the fields predicates domain)

$$\text{Let } F_1, F_2 \in \mathscr{P}(\mathbb{F}), \text{ then: } F_1 *_{\mathrm{T}} F_2 = F_1 \cup F_2$$

Theorem 4 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathrm{T}}$) *The functions* $\mathbf{id}_{\mathbb{T}^\sharp}$ *and* $*_{\mathrm{T}}$ *of the fields predicates domain satisfy respectively Condition 2 and Condition 3.*

Proof (Proof of Theorem 4) The proof of this theorem is trivial. For $\mathbf{id}_{\mathbb{T}^\sharp}$, we can see from Lemma 2 that the empty set is the most precise element of $\mathbb{T}^\sharp$. This is why we use it to define this function.

4.3.3 The Combined Predicates Domain

The footprint predicates domain expresses relations between the set of addresses of the input and the output heaps but does not express information about the content of memory cells. On the other side, the fields predicates domain provides information about the content of memory cells but not about addresses. If we can combine the informations given by both of these abstract heap predicates domains, we can ensure the properties (a) and (b) of Remark 2.

A reduced product [18] between the footprint and the fields predicates domains is sufficient but not modular enough. Indeed, to analyze other programs, we have to combine other abstract heap transformation predicates domains. To be the most generic as possible, we define *the combined predicates domain*, the abstract heap transformation predicates domain $\mathbb{T}^\sharp = \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp$ as a partially reduced product of any abstract heap transformation predicates domains $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$, where reduction is computed as part of the abstract operations over transformation predicates (we call this produce partially reduced, since we do not require the optimum reduction to be computed by all operators).

Let $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}$ be a transform-into relation annotated by $\mathrm{t}^\sharp$. If $\mathrm{t}^\sharp$ is the product between two abstract heap transformation predicates $\mathrm{t}_1^\sharp \in \mathbb{T}_1^\sharp$ and $\mathrm{t}_2^\sharp \in \mathbb{T}_2^\sharp$, then it describes the transformation of the heap abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ into the heap abstracted by $\mathrm{h}_\mathrm{o}^\sharp$ by combining the transformations described both by $\mathrm{t}_1^\sharp$ and $\mathrm{t}_2^\sharp$.

Definition 11 (Concretization of the combined predicates domain) Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathrm{t}_1^\sharp \in \mathbb{T}_1^\sharp$ and $\mathrm{t}_2^\sharp \in \mathbb{T}_2^\sharp$. Let $\gamma_{\mathbb{T}_1^\sharp}$ and $\gamma_{\mathbb{T}_2^\sharp}$ be the concretization function of respectively $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$, we define the concretization function of the product between $\mathrm{t}_1^\sharp$ and $\mathrm{t}_2^\sharp$, $\gamma_{\mathbb{T}^\sharp} : \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp \to \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ as follows:

$$\gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp, \mathrm{t}_2^\sharp) = \gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_1^\sharp) \cap \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_2^\sharp)$$

Theorem 5 (Property of the combined predicates domain) *Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and let $\mathbb{T}^\sharp$ be their product. Let $\gamma_{\mathbb{T}_1^\sharp}$, $\gamma_{\mathbb{T}_2^\sharp}$ and $\gamma_{\mathbb{T}^\sharp}$ be respectively their concretization functions. Then:*

$$\forall \mathrm{t}_{1,\mathrm{a}}^\sharp, \mathrm{t}_{1,\mathrm{b}}^\sharp \in \mathbb{T}_1^\sharp, \quad \forall \mathrm{t}_{2,\mathrm{a}}^\sharp, \mathrm{t}_{2,\mathrm{b}}^\sharp \in \mathbb{T}_2^\sharp,$$

$$\gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_{1,\mathrm{a}}^\sharp) \subseteq \gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_{1,\mathrm{b}}^\sharp) \wedge \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_{2,\mathrm{a}}^\sharp) \subseteq \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_{2,\mathrm{b}}^\sharp) \Rightarrow \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_{1,\mathrm{a}}^\sharp, \mathrm{t}_{2,\mathrm{a}}^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_{1,\mathrm{b}}^\sharp, \mathrm{t}_{2,\mathrm{b}}^\sharp)$$

Proof (Proof of Theorem 5) To prove this property we simply substitute $\gamma_{\mathbb{T}^\sharp}$ by its definition and we obtain:

$$\forall \mathrm{t}_{1,\mathrm{a}}^\sharp, \mathrm{t}_{1,\mathrm{b}}^\sharp \in \mathbb{T}_1^\sharp, \quad \forall \mathrm{t}_{2,\mathrm{a}}^\sharp, \mathrm{t}_{2,\mathrm{b}}^\sharp \in \mathbb{T}_2^\sharp,$$

$$\begin{gathered}\gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_{1,\mathrm{a}}^\sharp) \subseteq \gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_{1,\mathrm{b}}^\sharp) \wedge \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_{2,\mathrm{a}}^\sharp) \subseteq \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_{2,\mathrm{b}}^\sharp) \\ \Rightarrow \\ \gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_{1,\mathrm{a}}^\sharp) \cap \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_{2,\mathrm{a}}^\sharp) \subseteq \gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_{1,\mathrm{b}}^\sharp) \cap \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_{2,\mathrm{b}}^\sharp)\end{gathered}$$

Definition 12 (Function $\mathbf{id}_{\mathbb{T}^\sharp}$ of the combined predicates domain) Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathbb{T}^\sharp$ their product. Let $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$, then :

$$\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp) = (\mathbf{id}_{\mathbb{T}_1^\sharp}(\mathrm{h}^\sharp), \mathbf{id}_{\mathbb{T}_2^\sharp}(\mathrm{h}^\sharp))$$

Definition 13 (Operator $*_{\mathtt{T}}$ of the combined predicates domain) Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathbb{T}^\sharp$ their product. If $t^\sharp_{1,a}, t^\sharp_{1,b} \in \mathbb{T}_1^\sharp$ and $t^\sharp_{2,a}, t^\sharp_{2,b} \in \mathbb{T}_2^\sharp$ then:

$$(t^\sharp_{1,a}, t^\sharp_{2,a}) *_{\mathtt{T}} (t^\sharp_{1,b}, t^\sharp_{2,b}) = (t^\sharp_{1,a} *_{\mathtt{T}_1} t^\sharp_{1,b} \ , \ t^\sharp_{2,a} *_{\mathtt{T}_2} t^\sharp_{2,b})$$

Theorem 6 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$) *We assume that the two abstract heap transformation predicates domains $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ are equiped with sound $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$ operators. The functions $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$ of the combined predicates domain satisfy respectively Condition 2 and Condition 3.*

Proof (Proof of Theorem 6) The function $\mathbf{id}_{\mathbb{T}^\sharp}$ is simply the product of the functions $\mathbf{id}_{\mathbb{T}_1^\sharp}$ and $\mathbf{id}_{\mathbb{T}_2^\sharp}$ of the sub-domains. It is easy to prove its soundness if we suppose that $\mathbf{id}_{\mathbb{T}_1^\sharp}$ and $\mathbf{id}_{\mathbb{T}_2^\sharp}$ are both sound. Similarly, we can prove that $*_{\mathtt{T}}$ is sound supposing that $*_{\mathtt{T}_1}$ and $*_{\mathtt{T}_2}$ are sound.

Example 9 (Computed transform-into relation for the list sort) Using the product of the fields and the footprints predicates domains, we obtain the following transform-into relation for the list sort program in Figure 3:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]_{t^\sharp}, \text{ with } t^\sharp = (\{\mathtt{next}\}, =^\sharp)$$

This abstract heap relation describes exactly the points (a) and (b) of Remark 2: the function works in place and the input and output lists have the same length, as they are defined by the same set of addresses (a). Furthermore, only the `next` fields of the list may have been modified, so this implies that the lists have exactly the same data (b).

The footprint and the fields predicates domains offer the advantages to be complementary and totally independent from data structures (they are not specific to linked lists). They can also express other properties like for example, if we traverse a binary tree and increment its data elements, the product of these domains is able to prove that only data may have been modified. For more specific relational properties, designing new abstract heap transformation predicates domains is possible.

4.4 Abstract Memory Relations and Disjunction Abstract Domain

Until now, we have defined abstractions for relations between two heaps. In this section, we give an abstraction for relations between two memory states. We also define an abstraction for a disjunction of memory relations.

4.4.1 Abstract Memory Relation

Numerical Abstract Domains. In our heap abstraction, we simply name addresses and values with symbolic values. We have no information about their concrete values. However, it is crucial in our analysis to remember whether that a pointer is null or not. Moreover, it is also necessary to be able to capture numerical invariants to increase the precision of the analysis. Such information can be captured with *numerical abstract domains* such as intervals [17] or convex polyhedra [15]. Thus, we enrich our abstraction with a numerical abstract domain $\mathbb{N}^\sharp$. We do not fix a particular numerical abstract domain, as it does not change our analysis. It just

requires to implement all the functions for numerical abstract domains and to respect their conditions of soundness provided all along in Section 5.

A *numerical abstract value* $\mathrm{n}^\sharp \in \mathbb{N}^\sharp$ abstracts the numerical value of the symbolic values that appear in an abstract heap relation.

The concretization of abstract numerical domains $\gamma_{\mathbb{N}^\sharp}$ gives a concrete value to each symbolic value. Thus, it just binds a numerical abstract value into a set a valuations:

$$\gamma_{\mathbb{N}^\sharp} : \mathbb{N}^\sharp \to \mathscr{P}([\mathbb{V}^\sharp \to \mathbb{V}])$$

Abstract Memory Relations. To have a complete abstraction of memory relations, we only have to abstract environments. An *abstract environment* $\mathrm{e}^\sharp \in \mathbb{E}^\sharp$ is simply a function that maps program variables to symbolic values that correspond to their concrete addresses (thus $\mathbb{E}^\sharp = [\mathbb{X} \to \mathbb{V}^\sharp]$).

Finally, an *abstract memory relation* $\mathrm{m}^\sharp_{\mathscr{R}} \in \mathbb{M}^\sharp_{\mathscr{R}} = \mathbb{E}^\sharp \times \mathbb{R}^\sharp \times \mathbb{N}^\sharp$ is a triplet made of an abstract environment $\mathrm{e}^\sharp$, that binds program variables into their symbolic values in the abstract heap relation $\mathrm{r}^\sharp$, and a numerical abstract value $\mathrm{n}^\sharp$ that abstracts the value of symbolic values of $\mathrm{r}^\sharp$.

Definition 14 (Concretization of abstract memory relations) The concretization of abstract memory relations $\gamma_{\mathbb{M}^\sharp_{\mathscr{R}}} : \mathbb{M}^\sharp_{\mathscr{R}} \to \mathscr{P}(\mathbb{M} \times \mathbb{M})$ maps an abstract memory relation $\mathrm{m}^\sharp_{\mathscr{R}} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ into the pairs of its concrete input and output memories.

$$\begin{aligned}\gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp) = \{((\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_i}),\ (\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_o})) \mid \\ (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \wedge \nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp)\}\end{aligned}$$

We observe that the concrete environment is always the same in the input and in the output memory as the address of a variable never changes between two program points.

4.4.2 Disjunction Abstract Domain

The analysis algorithm may require to *unfold* inductive predicates (see Section 5.2). Unfolding such predicates may generate a *finite set* of abstract memory relations. Consequently, the analysis needs an abstraction layer that reasons over it. We thus let $\mathbb{D}^\sharp = \mathscr{P}_{\mathrm{fin}}(\mathbb{M}^\sharp_{\mathscr{R}})$ be the disjunction abstract domain. An abstract disjunction $\mathrm{d}^\sharp \in \mathbb{D}^\sharp$ simply represents a finite set of abstract memory relations.

Definition 15 (Concretization of the disjunction abstract domain)
The concretization function $\gamma_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \to \mathscr{P}(\mathbb{M} \times \mathbb{M})$ maps an abstract disjunction $\mathrm{d}^\sharp$ into the pairs of its concrete input and output memories.

$$\gamma_{\mathbb{D}^\sharp}(\mathrm{d}^\sharp) = \bigcup_{\mathrm{m}^\sharp_{\mathscr{R}} \in \mathrm{d}^\sharp} \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}})$$

5 Static Analysis Operations

We now propose a static analysis to compute the abstract memory relations that we defined in 4.4. It proceeds by forward abstract interpretation [17], starting from the abstract heap relation $\mathtt{Id}(\mathrm{h}^\sharp)$ where $\mathrm{h}^\sharp$, supplied by the user, describes the heap at the function entry (e.g. states that

$$\frac{\mathrm{e}^\sharp(\mathtt{x}) = \alpha}{\mathbf{eval}_{\mathtt{L}}(\mathtt{x}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{0})}$$

$$\frac{\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f})}{\mathbf{eval}_{\mathtt{L}}(loc \cdot \mathtt{g}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f} + \mathtt{g})}$$

$$\frac{\mathbf{eval}_{\mathtt{E}}(exp, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \alpha)}{\mathbf{eval}_{\mathtt{L}}(*exp, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{0})}$$

$$\frac{\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f}) \quad \mathrm{r}^\sharp = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)}{\mathbf{eval}_{\mathtt{E}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta, \beta)}$$

$$\frac{\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f}) \quad \mathrm{r}^\sharp = \mathrm{r}_0^\sharp *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow (\mathrm{h}_{\mathrm{o}}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)]_{\mathrm{t}^\sharp}}{\mathbf{eval}_{\mathtt{E}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta, \beta)}$$

$$\frac{\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{0})}{\mathbf{eval}_{\mathtt{E}}(\&loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \alpha)}$$

$$\frac{\alpha \text{ fresh}}{\mathbf{eval}_{\mathtt{E}}(\mathrm{v}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathrm{v})}$$

$$\frac{\mathbf{eval}_{\mathtt{E}}(exp_1, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_1, \mathrm{p}_1^\sharp) \quad \mathbf{eval}_{\mathtt{E}}(exp_2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_2, \mathrm{p}_2^\sharp) \quad \alpha_3 \text{ fresh}}{\mathbf{eval}_{\mathtt{E}}(exp_1 \oplus exp_2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_3, \mathrm{p}_1^\sharp \oplus \mathrm{p}_2^\sharp)}$$

Fig. 10: Abstract evaluation of locations $\mathbf{eval}_{\mathtt{L}} : \mathtt{L} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{F}$ and expressions $\mathbf{eval}_{\mathtt{E}} : \mathtt{E} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{P}^\sharp$. We remind that $\mathtt{0}$ designs the null offset.

the first argument is a linked list). Indeed, before executing a program, any transformation has been performed on its initial state, so the initial relation of the analysis is the identity relation.

More generally, the analysis of a program $p \in \mathtt{P}$ is a function $[\![p]\!]_{\mathscr{R}}^\sharp$ that over-approximates the concrete program relational semantics $[\![p]\!]_{\mathscr{R}}$ defined in Section 3.3. It inputs an abstract disjunction describing a previous transformation $\mathscr{T}$ done on the input *before* running p and returns an other abstract disjunction describing that transformation $\mathscr{T}$ followed by the execution of p. Thus, $[\![p]\!]_{\mathscr{R}}^\sharp$ should meet the following soundness condition:

$$\begin{aligned} &\forall \mathrm{d}^\sharp \in \mathbb{D}^\sharp, \forall (\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{D}^\sharp}(\mathrm{d}^\sharp), \forall \mathrm{m}_2 \in \mathbb{M}, \\ &\qquad (\mathrm{m}_1, \mathrm{m}_2) \in [\![p]\!]_{\mathscr{R}} \Longrightarrow (\mathrm{m}_0, \mathrm{m}_2) \in \gamma_{\mathbb{D}^\sharp}([\![p]\!]_{\mathscr{R}}^\sharp(\mathrm{d}^\sharp)) \end{aligned}$$

5.1 Abstract Evaluation of Locations and Expressions

In this section, we define how locations and expressions are evaluated in our static analysis from abstract memory relations. We first consider only the abstract heap relations that do not contain inductive predicates.

We start by defining the abstract evaluation of locations $\mathbf{eval}_{\mathtt{L}} : \mathtt{L} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{F}$ and expressions $\mathbf{eval}_{\mathtt{E}} : \mathtt{E} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{P}^\sharp$ from an abstract environment $\mathrm{e}^\sharp$ and an abstract heap relations $\mathrm{r}^\sharp$. They follow the same principles as $\mathscr{L}[\![loc]\!]$ and $\mathscr{E}[\![exp]\!]$ defined in 3.3 and their definition rules are given in Figure 10.

The function $\mathbf{eval}_{\mathtt{L}}$ evaluates a location into a pair of an abstract value and an offset that correspond to the address of the location in $\mathrm{r}^\sharp$.

The abstract evaluation $\mathbf{eval}_{\mathtt{E}}$ should return the symbolic value $\alpha \in \mathbb{V}^\sharp$ in $\mathrm{r}^\sharp$ corresponding to the result of the concrete evaluation of an expression exp. However, if exp is of the form

$exp_1 \oplus exp_2$ or v, then there is no symbolic value α in $\mathrm{r}^\sharp$ that can result from the abstract evaluation of exp. To deal with this, the function $\mathbf{eval}_{\mathtt{E}}$ returns a pair of a symbolic value $\alpha \in \mathbb{V}^\sharp$ and a pure formula $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$. The pure formula $\mathrm{p}^\sharp$ is simply a translation of the expression, and α is a (potentially fresh) name for $\mathrm{p}^\sharp$. The concretization of this pair is such that if $(\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp)$, then $\nu(\alpha) = \mathrm{v}$.

When the evaluation needs to read the content of a cell (when we apply $\mathbf{eval}_{\mathtt{E}}$ on a location loc), we need to look at the value stored in the cell. For instance, if $\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f})$, in the case where $\mathrm{r}^\sharp = \mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \beta)$, it is obvious that the cell at address $\alpha \cdot \mathtt{f}$ contains β. In the case where $\mathrm{r}^\sharp = [(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \beta)]_{\mathrm{t}^\sharp}$, we remark that δ was the value in the cell at address $\alpha \cdot \mathtt{f}$ in the heap at the beginning of the analysis whereas β corresponds to its last value. Thus, $\mathbf{eval}_{\mathtt{E}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp)$ should return (β, β).

Theorem 7 (Soundness of $\mathbf{eval}_{\mathrm{L}}$ and $\mathbf{eval}_{\mathrm{E}}$) *The functions* $\mathbf{eval}_L$ *and* $\mathbf{eval}_E$ *are sound: Let* $\mathrm{e}^\sharp \in \mathbb{E}^\sharp$, $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$, $(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp)$, *$exp \in E$ and $loc \in L$ then:*

$$\begin{array}{l}\mathbf{eval}_L(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f}) \\ \quad \Rightarrow \mathscr{L}[\![loc]\!](\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_o}) = \nu(\alpha) + \mathtt{f}\end{array}$$

$$\begin{array}{l}\mathbf{eval}_E(exp, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathrm{p}^\sharp) \\ \quad \Rightarrow \mathscr{E}[\![exp]\!](\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_o}) = \nu(\alpha) \wedge (\nu(\alpha),\ \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp)\end{array}$$

Example 10 (Abstract evaluation of the expression $((*\mathtt{x}) \cdot \mathtt{f}) + 2$*)* We assume that $\mathrm{e}^\sharp(\mathtt{x}) = \alpha_0$ and $\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1) *_{\mathtt{R}} [(\alpha_1 \cdot \mathtt{f} \mapsto \alpha_2) \dashrightarrow (\alpha_1 \cdot \mathtt{f} \mapsto \alpha_3)]_{\mathrm{t}^\sharp}$.

We remark that $\mathbf{eval}_{\mathtt{L}}(*\mathtt{x}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_1, \mathtt{0})$, that $\mathbf{eval}_{\mathtt{E}}((*\mathtt{x}) \cdot \mathtt{f}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta_1, \alpha_3)$ and that $\mathbf{eval}_{\mathtt{E}}(2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta_2, 2)$ where β_1 and β_2 are fresh symbolic values.

Finally, we obtain that $\mathbf{eval}_{\mathtt{E}}(((*\mathtt{x}) \cdot \mathtt{f}) + 2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta_3, \alpha_3 + 2)$, where β_3 is a fresh symbolic value.

5.2 Inductive Predicates Unfolding

In the previous section, we considered only abstract memory relations without inductive predicates. We now remove this restriction.

5.2.1 Unfolding Abstract Memory Relations

The abstract evaluation of a location may require to *unfold* inductive predicates. Indeed, when a location is summarized by an inductive predicate, there is no points-to predicate corresponding to this location and so, its abstract evaluation cannot be done. The analysis first proceeds to the *unfolding* [10] of the inductive predicate in order to materialize it into points-to predicates. This is performed by the function $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathscr{R}}}$. This step generates a *finite disjunction* of abstract memory relations, where the inductive predicate has been syntactically substituted by the rules of its definition, as defined is Section 4.1 (one disjunct per rule of the inductive predicate). Also, the pure formula of each rule is evaluated and added in each disjunct of abstract memory relations. The irrelevant disjuncts with the location are discarded (for example the case where the summarized cell is null). The abstract evaluation of the location is then performed for each valid disjunct. This process is known in shape analysis as *materialization* of cells [40,25,10]. The abstract memory relations unfolding operator $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ builds upon the abstract heap

relations unfolding operator $\mathbf{unfold}_{\mathbb{R}^\sharp}$, itself building upon the (defined below) abstract heap unfolding operator $\mathbf{unfold}_{\mathbb{H}^\sharp}$.

The definition of $\mathbf{unfold}_{\mathbb{R}^\sharp}$ requires an *unfolding operation over abstract heaps* $\mathbf{unfold}_{\mathbb{H}^\sharp} : \mathbb{V}^\sharp \times \mathbb{H}^\sharp \to \mathscr{P}_{\mathrm{fin}}(\mathbb{H}^\sharp \times \mathbb{P}^\sharp)$. It takes a symbolic value α which is the origin of an inductive predicate and an abstract heap $h^\sharp$ that contains this inductive predicate. It returns the set of pairs $\{(h^\sharp_u, p^\sharp)\}$ where each $h^\sharp_u$ refines $h^\sharp$ following each definition rule of the inductive predicate and $p^\sharp$ is the pure formula of the corresponding rule. For instance, $\mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathbf{list}(\alpha))$ is $\{(\mathbf{emp}, \alpha = \mathbf{0x0}), (\alpha \cdot \mathtt{next} \mapsto \alpha_n *_S \alpha \cdot \mathtt{data} \mapsto \alpha_d *_S \mathbf{list}(\alpha_n), \alpha \neq \mathbf{0x0})\}$. If there is no inductive predicate attached to α in $h^\sharp$, we let $\mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp) = \{(h^\sharp, \mathbf{true})\}$. This operator is sound in the sense that, $\gamma_{\mathbb{H}^\sharp}(h^\sharp)$ is included in $\cup\{\gamma_\Sigma(h^\sharp_u, p^\sharp) \mid (h^\sharp_u, p^\sharp) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp)\}$.

The unfolding operation over abstract heap relations $\mathbf{unfold}_{\mathbb{R}^\sharp} : \mathbb{V}^\sharp \times \mathbb{R}^\sharp \to \mathscr{P}_{\mathrm{fin}}(\mathbb{R}^\sharp \times \mathbb{P}^\sharp)$ uses $\mathbf{unfold}_{\mathbb{H}^\sharp}$ to unfold the inductive predicate at abstract heap relations level.

Definition 16 (Abstract Heap Relation Unfolding) Let $r^\sharp \in \mathbb{R}^\sharp$, we define the unfolding operator for abstract heap relations
$\mathbf{unfold}_{\mathbb{R}^\sharp} : \mathbb{V}^\sharp \times \mathbb{R}^\sharp \to \mathscr{P}_{\mathrm{fin}}(\mathbb{R}^\sharp \times \mathbb{P}^\sharp)$:

- $\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathtt{Id}(h^\sharp)) = \{(\mathtt{Id}(h^\sharp_u), p^\sharp) \mid (h^\sharp_u, p^\sharp) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp)\}$
- $\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, [h^\sharp_i \dashrightarrow h^\sharp_o]_{t^\sharp}) = \{([h^\sharp_{i,u} \dashrightarrow h^\sharp_{o,u}]_{t^\sharp}, p^\sharp_{i,u} \wedge p^\sharp_{o,u}) \mid$
  $(h^\sharp_{i,u}, p^\sharp_{i,u}) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp_i) \wedge (h^\sharp_{o,u}, p^\sharp_{o,u}) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp_o)\}$
- $\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, r^\sharp_0 *_R r^\sharp_1) = \{(r^\sharp_{0,u} *_R r^\sharp_1, p^\sharp) \mid (r^\sharp_{0,u}, p^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, r^\sharp_0)\}$, when α carries an inductive predicate in $r^\sharp_0$.

Unfolding an inductive predicate under an identity relation only consists on unfolding the abstract heap and conserving the identity relation over the unfolded abstract heap. Unfolding under a transform-into relation requires to unfold *independently* both the input and the output abstract heaps of the relation. Then, the resulting pure formula is the conjunction of the pure formulas of each unfolded abstract heaps. We remark that the unfolding does not modify the transformation predicates: indeed, as unfolding refines an existing description of a relation, the transformation that is described by this relation still holds. However, we later comment on possible precision gains where the transformation predicates also get unfolded. Finally, unfolding an inductive predicate under a relational separating conjunction consists on unfolding locally the abstract heap relation where the inductive predicate appears.

Definition 17 (Abstract Memory Relation Unfolding)
Let $m^\sharp_{\mathscr{R}} = (e^\sharp, r^\sharp, n^\sharp) \in \mathbb{M}^\sharp_{\mathscr{R}}$, we define the unfolding operator for abstract memory relations $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathscr{R}}} : \mathbb{V}^\sharp \times \mathbb{M}^\sharp_{\mathscr{R}} \to \mathscr{P}_{\mathrm{fin}}(\mathbb{M}^\sharp_{\mathscr{R}})$:

$$\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathscr{R}}}(\alpha, m^\sharp_{\mathscr{R}}) = \{(e^\sharp, r^\sharp_u, \mathbf{guard}_{\mathbb{N}^\sharp}(p^\sharp, n^\sharp)) \mid (r^\sharp_u, p^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, r^\sharp)\}$$

The function $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ applies the pure conditions returned by $\mathbf{unfold}_{\mathbb{R}^\sharp}$ using $\mathbf{guard}_{\mathbb{N}^\sharp}$. This relies on the numerical guard $\mathbf{guard}_{\mathbb{N}^\sharp} : \mathbb{P}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$ that updates an abstract numerical value $n^\sharp$ taking into account the effects of a pure formula $p^\sharp$. This numerical guard should satisfy the following condition:

Condition 4 (Soundness of $\mathbf{guard}_{\mathbb{N}^\sharp}$) *Let* $p^\sharp \in \mathbb{P}^\sharp, n^\sharp \in \mathbb{N}^\sharp$. *The function* $\mathbf{guard}_{\mathbb{N}^\sharp}$ *is sound if:*

$$\nu \in \gamma_{\mathbb{N}^\sharp}(n^\sharp) \wedge (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(p^\sharp) \wedge \mathrm{v} \neq 0 \quad \Longrightarrow \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{guard}_{\mathbb{N}^\sharp}(p^\sharp, n^\sharp))$$

Intuitively, this condition asserts that the condition test operator $\mathbf{guard}_{\mathbb{N}^\sharp}$ should always return an over-approximation of the valuations that satisfy the condition. It is standard in all value abstract domains that we know of.

Theorem 8 (Soundness of unfolding operators) *Let* $\gamma_\Pi : \mathbb{R}^\sharp \times \mathbb{P}^\sharp \to \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ *be the concretization function of pairs of abstract heap relation and pure formula defined as follow:*

$$\gamma_\Pi(\mathrm{r}^\sharp, \mathrm{p}^\sharp) = \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \mid (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \wedge (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\}$$

Let $\mathrm{r}^\sharp \in \mathbb{R}^\sharp, \mathrm{m}^\sharp_{\mathscr{R}} \in \mathbb{M}^\sharp_{\mathscr{R}}$ *and* $\alpha \in \mathbb{V}^\sharp$*. Then, the unfolding operators are sound, in the sense that:*

$$\gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \subseteq \bigcup\{\gamma_\Pi(\mathrm{r}^\sharp_\mathrm{u}, \mathrm{p}^\sharp) \mid (\mathrm{r}^\sharp_\mathrm{u}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathrm{r}^\sharp)\}$$

$$\gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}}) \subseteq \bigcup\{\gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},\mathrm{u}}) \mid (\mathrm{m}^\sharp_{\mathscr{R},\mathrm{u}}) \in \mathbf{unfold}_{\mathbb{M}^\sharp_{\mathscr{R}}}(\alpha, \mathrm{m}^\sharp_{\mathscr{R}})\}$$

Example 11 (Unfolding a transform-into relation) Consider the following abstract heap relation $\mathrm{r}^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]_{\mathrm{t}^\sharp}$. Unfolding $\mathrm{r}^\sharp$ at symbolic value α should generate four pairs of abstract heap relation and pure formula, included two cases where the pure formula is $\alpha = \mathbf{0x0} \wedge \alpha \neq \mathbf{0x0}$. These cases should be discarded by $\mathbf{guard}_{\mathbb{N}^\sharp}$ at the abstract memory relation level. The other remaining cases are thus: $([\mathbf{emp} \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp}, \alpha = \mathbf{0x0} \wedge \alpha = \mathbf{0x0})$ and $([(\alpha \cdot \mathtt{data} \mapsto \delta_\mathrm{i} *_\mathtt{S} \alpha \cdot \mathtt{next} \mapsto \beta_\mathrm{i} *_\mathtt{S} \mathbf{list}(\beta_\mathrm{i})) \dashrightarrow (\alpha \cdot \mathtt{data} \mapsto \delta_\mathrm{o} *_\mathtt{S} \alpha \cdot \mathtt{next} \mapsto \beta_\mathrm{o} *_\mathtt{S} \mathbf{list}(\beta_\mathrm{o}))]_{\mathrm{t}^\sharp}, \alpha \neq \mathbf{0x0} \wedge \alpha \neq \mathbf{0x0})$. Remark that in the latter case we obtained different symbolic values for the `data` and `next` fields of α because we have unfolded independently each abstract heap.

Example 12 (Abstract memory relation unfolding) 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). The analysis should start from $\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_\mathtt{S} \mathbf{list}(\alpha))$ (in this example, we omit v for the sake of concision). Note that we did not specify that α is not null. Before the loop entry, the analysis computes the abstract heap relation $\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_\mathtt{S} \mathbf{list}(\alpha)) *_\mathtt{R} [\mathbf{emp} \dashrightarrow \&\mathtt{c} \mapsto \alpha]_{\mathrm{t}^\sharp}$. To deal with the test `c->next != NULL`, the analysis should materialize α. This unfolding is performed under the `Id` connective, and produces:

$$\begin{array}{l}(\mathtt{Id}(\&\mathtt{l} \mapsto \alpha *_\mathtt{S} \alpha \cdot \mathtt{next} \mapsto \alpha_0 *_\mathtt{S} \alpha \cdot \mathtt{data} \mapsto \beta_0 *_\mathtt{S} \mathbf{list}(\alpha_0)) \\ \quad *_\mathtt{R} [\mathbf{emp} \dashrightarrow (\&\mathtt{c} \mapsto \alpha)]_{\mathrm{t}^\sharp}, \alpha \neq \mathbf{0x0})\end{array}$$

Then, the condition $\alpha \neq \mathbf{0x0}$ is kept in the numerical abstract value. We observe that we have automatically inferred that the input of the function should be non-empty, as this condition also applies to the abstract input memory. In turn, the effect of the condition test and of the assignment in the loop body can be precisely analyzed from this abstract memory relation.

5.2.2 Refining unfolding with abstract heap transformation predicates

Abstract heap transformation predicates can also help to gain more precision. As an example, we consider the following abstract heap relation: $[\mathbf{list}(\alpha_0) \dashrightarrow \mathbf{list}(\alpha_0)]_{\mathrm{t}^\sharp}$. Unfolding α_0 will generate the disjuncts $([\mathbf{emp} \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp}, \alpha_0 = \mathbf{0x0})$ and $([(\alpha_0 \cdot \mathtt{data} \mapsto \beta_1 *_\mathtt{S} \alpha_0 \cdot \mathtt{next} \mapsto \beta_2 *_\mathtt{S} \mathbf{list}(\beta_2)) \dashrightarrow (\alpha_0 \cdot \mathtt{data} \mapsto \delta_1 *_\mathtt{S} \alpha_0 \cdot \mathtt{next} \mapsto \delta_2 *_\mathtt{S} \mathbf{list}(\delta_2))]_{\mathrm{t}^\sharp}, \alpha_0 \neq \mathbf{0x0})$.

In the second disjunct, we observe that we have no information that says whether the values of the fields `next` and `data` of α_0 are respectively the same in both sides of the

$[. \dashrightarrow .]$ relation. However, if we consider that we use the fields predicates domain defined in Section 4.3 and that $t^\sharp = \{\texttt{data}\}$, we know that all `next` fields in this abstract relation are left unmodified. Thus, unfolding α_0 taking in account this information will generate as second disjunct $([(\alpha_0 \cdot \texttt{data} \mapsto \beta_1 *_S \alpha_0 \cdot \texttt{next} \mapsto \alpha_2 *_S \mathbf{list}(\alpha_2)) \dashrightarrow (\alpha_0 \cdot \texttt{data} \mapsto \delta_1 *_S \alpha_0 \cdot \texttt{next} \mapsto \alpha_2 *_S \mathbf{list}(\alpha_2))]_{t^\sharp}, \alpha_0 \neq \mathbf{0x0})$ (as only `data` fields may be different, we can use the same variable α_2 in both sides of the relation).

The refinement of materialization is performed by the function $\mathbf{unfold}_{\mathbb{T}^\sharp} : \mathbb{V}^\sharp \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{T}^\sharp \to \mathscr{P}_{\mathrm{fin}}(\mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{P}^\sharp)$ that refines the unfolded input and output abstract heaps at the given address, taking in account the information provided by the transformation predicate. This function should satisfy the following condition:

Condition 5 (Soundness of $\mathbf{unfold}_{\mathbb{T}^\sharp}$) *Before giving the soundness condition of* $\mathbf{unfold}_{\mathbb{T}^\sharp}$*, we define the concretization function* $\gamma_\Pi : \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{P}^\sharp \to \mathscr{P}(\mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}])$ *of a triplet* $(h^\sharp_{i,u}, h^\sharp_{o,u}, p^\sharp) \in \mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, h^\sharp_i, h^\sharp_o, t^\sharp)$*:*

$$\gamma_\Pi(h^\sharp_{i,u}, h^\sharp_{o,u}, p^\sharp) = \{(h_i, h_o, \nu) \mid (h_i, \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp_{i,u}) \land (h_o, \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp_{o,u}) \\ \land \exists v, (v, \nu) \in \gamma_{\mathbb{P}^\sharp}(p^\sharp) \land v \neq 0\}$$

Let $\mathbb{T}^\sharp$ *be an abstract heap transformation predicates domain,* $t^\sharp \in \mathbb{T}^\sharp, h^\sharp_i, h^\sharp_o \in \mathbb{H}^\sharp$ *and* $\alpha \in \mathbb{V}^\sharp$*. Then* $\mathbf{unfold}_{\mathbb{T}^\sharp}$ *is sound if:*

$$\{(h_i, h_o, \nu) \mid (h_i, h_o, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp) \land (h_i, \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp_i) \land (h_o, \nu) \in \gamma_{\mathbb{H}^\sharp}(h^\sharp_o)\} \\ \subseteq \\ \bigcup\{\gamma_\Pi(h^\sharp_{i,u}, h^\sharp_{o,u}, p^\sharp) \mid (h^\sharp_{i,u}, h^\sharp_{o,u}, p^\sharp) \in \mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, h^\sharp_i, h^\sharp_o, t^\sharp)\}\}$$

Intuitively, we can see that $\mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, h^\sharp_i, h^\sharp_o, t^\sharp)$ propagates the information provided by $t^\sharp$ into $h^\sharp_i$ and $h^\sharp_o$. The result of this propagation should not introduce more information than $t^\sharp$.

Definition 18 (Extended definition of $\mathbf{unfold}_{\mathbb{R}^\sharp}$) We now extend the definition of $\mathbf{unfold}_{\mathbb{R}^\sharp}$ taking into account abstract heap transformations predicates:

$$\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, [h^\sharp_i \dashrightarrow h^\sharp_o]_{t^\sharp}) = \{([h^\sharp_{i,t} \dashrightarrow h^\sharp_{o,t}]_{t^\sharp}, p^\sharp_t \land p^\sharp_{i,u} \land p^\sharp_{o,u}) \mid \\ (h^\sharp_{i,u}, p^\sharp_{i,u}) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp_i) \land (h^\sharp_{o,u}, p^\sharp_{o,u}) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, h^\sharp_o) \\ \land (h^\sharp_{i,t}, h^\sharp_{o,t}, p^\sharp_t) \in \mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, h^\sharp_{i,u}, h^\sharp_{o,u}, t^\sharp)\}$$

Remark 3 In the remainder of this paper, we assume that unfolding is already performed before reading a location or an expression. So that we do not explicit the steps when unfolding occurs.

5.3 Assignment

In this section, we define the transfer function for assignments $\mathbf{assign}_{\mathbb{M}^\sharp_{\mathscr{R}}} : \mathtt{L} \times \mathtt{E} \times \mathbb{M}^\sharp_{\mathscr{R}} \to \mathbb{M}^\sharp_{\mathscr{R}}$. Remind that the concrete assignment over a pair of memory states (m_i, m_o) results in an other pair of memory states (m_i, m'_o) where only the output state m_o has been modified. Likewise the abstract assignment will keep the abstract input state unmodified. It will also preserve as many relations between the abstract input and output states as possible.

The main part of the algorithm consists in the function $\mathbf{assign}_{\mathbb{R}^\sharp}$ that inputs a symbolic value α and a field `f` (the address), a symbolic value β (the value) and an abstract heap relation $r^\sharp$. It performs the following steps:

1. Decompose inductively $r^\sharp$ when it is of the form $r_0^\sharp *_R r_1^\sharp$ until we find the points-to predicate whose address is $\alpha \cdot \mathtt{f}$. The remainder of the abstract relation is integrally preserved.
2. If the found abstract heap relation is of the form $[h_i^\sharp \dashrightarrow h_o^\sharp]_{t^\sharp}$, replace it by $[h_i^\sharp \dashrightarrow h_{o'}^\sharp]_{t'^\sharp}$ where $h_{o'}^\sharp$ and $t'^\sharp$ reflect respectively the assignment in the abstract heap domain and the abstract heap transformation predicates domain.
3. If the found abstract heap relation is of the form $\mathtt{Id}(h^\sharp)$, decompose it into $\mathtt{Id}(h_1^\sharp) *_R \mathtt{Id}(h_2^\sharp)$ where $h_2^\sharp$ only contains the points-to predicate whose address is $\alpha \cdot \mathtt{f}$. Then decay $\mathtt{Id}(h_2^\sharp)$ into $[h_2^\sharp \dashrightarrow h_2^\sharp]_{t^\sharp}$ where $t^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(h_2^\sharp)$ tries to recover a part of the lost identity relation. Finally, proceed to the assignment in $[h_2^\sharp \dashrightarrow h_2^\sharp]_{t^\sharp}$ and let $\mathtt{Id}(h_1^\sharp)$ unchanged.
   For example (forgetting about abstract heap transformation predicates):

$$\begin{aligned}&\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta,\ \mathtt{Id}(h^\sharp *_S \alpha \cdot \mathtt{f} \mapsto \delta))\\ &\quad = \mathtt{Id}(h^\sharp) *_R [(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \beta)]\end{aligned}$$

On top of this algorithm, $\mathbf{assign}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ evaluates the assignment $loc = exp$ over the abstract memory relation $m^\sharp_{\mathscr{R}} = (e^\sharp, r^\sharp, n^\sharp)$ using the following steps:

1. Evaluate *loc* and *exp* with respectively $\mathbf{eval}_L$ and $\mathbf{eval}_E$. Note that this may require prior unfoldings.
2. Update the abstract numerical value $n^\sharp$ with the result of the evaluation of *exp*, using the function $\mathbf{assign}_{\mathbb{N}^\sharp}$.
3. Update $r^\sharp$ with $\mathbf{assign}_{\mathbb{R}^\sharp}$.

Definition 19 (Assignments for abstract heap relations) We define the assignment function for abstract heap relations $\mathbf{assign}_{\mathbb{R}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$:

$$\frac{\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, r_1^\sharp) = r_2^\sharp}{\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, r_0^\sharp *_R r_1^\sharp) = r_0^\sharp *_R r_2^\sharp}$$

$$\frac{t_2^\sharp = \mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, t_1^\sharp)}{\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, [h_i^\sharp \dashrightarrow h_o^\sharp *_S (\alpha \cdot \mathtt{f} \mapsto \gamma)]_{t_1^\sharp}) = [h_i^\sharp \dashrightarrow h_o^\sharp *_S (\alpha \cdot \mathtt{f} \mapsto \beta)]_{t_2^\sharp}}$$

$$\frac{t^\sharp = \mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta,\ \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \delta))}{\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathtt{Id}(h_0^\sharp *_S \alpha \cdot \mathtt{f} \mapsto \delta)) = \mathtt{Id}(h_0^\sharp) *_R [\alpha \cdot \mathtt{f} \mapsto \delta \dashrightarrow \alpha \cdot \mathtt{f} \mapsto \beta]_{t^\sharp}}$$

Case of a Relational Separating Conjunction. We now assume that $r^\sharp = r_0^\sharp *_R r_1^\sharp$. The points-to predicate at address $\alpha \cdot \mathtt{f}$ can only appear in one of $r_0^\sharp$ or $r_1^\sharp$. If it appears in $r_0^\sharp$, the assignment should have no effect on $r_1^\sharp$ (and vice versa).

The function $\mathbf{assign}_{\mathbb{R}^\sharp}$ can thus be applied recursively on the sub-abstract heap relation where the points-to predicate appears. This relies on the same principle as the Frame rule [39] for separation logic, but for abstract heap relations. More generally, if $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, r_1^\sharp) = r_2^\sharp$, then $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, r_0^\sharp *_R r_1^\sharp) = r_0^\sharp *_R r_2^\sharp$.

Case of a Transform-into Relation. In the case when $r^\sharp = [h_0^\sharp \dashrightarrow h_1^\sharp]_{t_1^\sharp}$, $\mathbf{assign}_{\mathbb{R}^\sharp}$ applies on $h_1^\sharp$ the Frame rule [39] of separation logic (separating the points-to predicate at address $\alpha \cdot \mathtt{f}$ from the rest). Then it updates this points-to predicate to point to β, producing a new abstract heap $h_2^\sharp$. The algorithm also needs to take into account the effects of the assignment in the abstract heap transformation predicate $t_1^\sharp$. This is done by $\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, t_1^\sharp)$ that produces the abstract heap transformations predicate $t_2^\sharp$. So a valid definition for this assignment is $[h_0^\sharp \dashrightarrow h_2^\sharp]_{t_2^\sharp}$.

Case of an Identity Relation. We now assume that $r^\sharp = \mathtt{Id}(h^\sharp)$.
If $h^\sharp = \alpha \cdot \mathtt{f} \mapsto \delta *_{\mathtt{S}} h_0^\sharp$, two preliminary steps are necessary before updating $\alpha \cdot \mathtt{f}$. Indeed, two important points should be considered: first the points-to predicate $\alpha \cdot \mathtt{f} \mapsto \delta$ cannot be substituted by $\alpha \cdot \mathtt{f} \mapsto \beta$ under the Id connective, because the assignment breaks the identity relation. Second, the assignment should only modify the points-to predicate $\alpha \cdot \mathtt{f} \mapsto \delta$, and should preserve the identity relation over $h_0^\sharp$, to improve precision.

The first step consists in splitting the identity relation into two identity relations. As observed in Theorem 1, $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h_0^\sharp *_{\mathtt{S}} h_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(h_1^\sharp))$. In our case, $r^\sharp$ is split into $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta) *_{\mathtt{R}} \mathtt{Id}(h_0^\sharp)$. The first identity relation contains only the points-to predicate that is going to be modified and the second identity relation contains the other parts of $h^\sharp$ that are only read or useless during the assignment.

The second step consists on weakening $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$ into a transform-into relation. In Theorem 2, we have $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(h^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([h^\sharp \dashrightarrow h^\sharp]_{t^\sharp})$ with $t^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(h^\sharp)$. This property allows to weaken $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$ into $[(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \delta)]_{t_0^\sharp}$ where $t_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \delta)$. Recall that $\mathbf{id}_{\mathbb{T}^\sharp}$ has been introduced in Section 4.3 and depends on the abstract heap transformation predicates domain $\mathbb{T}^\sharp$ used by the analysis.

After these steps, the analysis can perform the assignment in the obtained transform-into relation and preserve the split identity relation. This relies on the combination of the assignment in the cases of relational separating conjunction relations and transform-into relations. Thus, a valid definition for this assignment is $\mathtt{Id}(h_0^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \gamma \dashrightarrow \alpha \cdot \mathtt{f} \mapsto \beta]_{t^\sharp}$, with $t^\sharp = \mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, t_0^\sharp)$.

Recall that $h[a \leftarrow v]$ is the concrete heap where we update the content of the cell at address a with the value v in the concrete heap h. We now give the soundness conditions for the function $\mathbf{assign}_{\mathbb{T}^\sharp}$ and the theorem soundness for the functions $\mathbf{assign}_{\mathbb{R}^\sharp}$.

Condition 6 (Soundness of $\mathbf{assign}_{\mathbb{T}^\sharp}$) *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $t^\sharp \in \mathbb{T}^\sharp$. Let $\alpha, \beta \in \mathbb{V}^\sharp$ and $\mathtt{f} \in \mathbb{F}$, then $\mathbf{assign}_{\mathbb{T}^\sharp}$ is sound if:*

$$\begin{aligned}&\{(h_i,\ h_o[\nu(\alpha) + \mathtt{f} \leftarrow \nu(\beta)],\ \nu) \mid (h_i,\ h_o\ \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp)\}\\ &\quad \subseteq \gamma_{\mathbb{T}^\sharp}(\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, t^\sharp))\end{aligned}$$

Theorem 9 (Soundness of $\mathbf{assign}_{\mathbb{R}^\sharp}$) *We assume that function $\mathbf{assign}_{\mathbb{T}^\sharp}$ is sound. Let $\alpha, \beta \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}$ and $r_0^\sharp \in \mathbb{R}^\sharp$. Assuming that Condition 6 is satisfied, the function $\mathbf{assign}_{\mathbb{R}^\sharp}$ is sound:*

$$\begin{aligned}&(h_0, h_1, \nu) \in \gamma_{\mathbb{R}^\sharp}(r_0^\sharp)\\ &\quad \Longrightarrow (h_0, h_1[\nu(\alpha) + \mathtt{f} \leftarrow \nu(\beta)], \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, r_0^\sharp))\end{aligned}$$

The analysis algorithm for assignments also needs to update the numerical constraints. To do this, we assume that the numerical abstract domain provides a function $\mathbf{assign}_{\mathbb{N}^\sharp} : \mathbb{V}^\sharp \times \mathbb{P}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$, which satisfies the following soundness condition:

Condition 7 (Soundness of $\mathbf{assign}_{\mathbb{N}^\sharp}$) *Let $\beta \in \mathbb{V}^\sharp, p^\sharp \in \mathbb{P}^\sharp$ and $n^\sharp \in \mathbb{N}^\sharp$, then $\mathbf{assign}_{\mathbb{N}^\sharp}$ is sound if:*

$$\nu \in \gamma_{\mathbb{N}^\sharp}(n^\sharp) \wedge (v, \nu) \in \gamma_{\mathbb{P}^\sharp}(\beta = p^\sharp) \wedge v \neq 0 \quad \Longrightarrow \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{assign}_{\mathbb{N}^\sharp}(\beta, p^\sharp, n^\sharp))$$

Definition 20 (Assignment in abstract memory relations)
Let $\mathrm{m}^\sharp_{\mathscr{R}} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$ be an abstract memory relation. If $\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp_0) = (\alpha, \mathtt{f})$ and $\mathbf{eval}_{\mathtt{E}}(exp, \mathrm{e}^\sharp, \mathrm{r}^\sharp_0) = (\beta, \mathrm{p}^\sharp)$, and if $\mathbf{assign}_{\mathbb{N}^\sharp}(\beta, \mathrm{p}^\sharp, \mathrm{n}^\sharp_0) = \mathrm{n}^\sharp_1$ and
$\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}^\sharp_0) = \mathrm{r}^\sharp_1$, then:

$$\mathbf{assign}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc, exp, (\mathrm{e}^\sharp, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1)$$

The function $\mathbf{assign}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ applies respectively $\mathbf{eval}_{\mathtt{L}}$ and $\mathbf{eval}_{\mathtt{E}}$ on loc and exp. Remark that this step may unfold inductive predicates, as explained in the previous section. It then applies $\mathbf{assign}_{\mathbb{R}^\sharp}$ on the address returned by $\mathbf{eval}_{\mathtt{L}}$ and the value returned by $\mathbf{eval}_{\mathtt{E}}$. Remind that $\mathbf{eval}_{\mathtt{E}}$ returns a pair $(\beta, \mathrm{p}^\sharp)$, such as $\beta \in \mathbb{V}^\sharp$ and $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$. To keep track of the numerical constraint $\beta = \mathrm{p}^\sharp$, the abstract assignment requires a function that over-approximates this constraint and put it in the abstract numerical value $\mathrm{n}^\sharp$. The $\mathbf{assign}_{\mathbb{N}^\sharp}$ function allows to perform this last step.

Theorem 10 (Soundness of $\mathbf{assign}_{\mathbb{M}^\sharp_{\mathscr{R}}}$) *We assume that function $\mathbf{assign}_{\mathbb{N}^\sharp}$ is sound. Let $loc \in \mathtt{L}, exp \in \mathtt{E}$. Let $\mathrm{m}^\sharp_{\mathscr{R}} \in \mathbb{M}^\sharp_{\mathscr{R}}$. For all $(\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}})$ such that $\mathrm{m}_1 = (\mathrm{e}_1, \mathrm{h}_1)$, then:*

$$(\mathrm{m}_0,\ (\mathrm{e}_1, \mathrm{h}_1[\mathscr{L}[\![loc]\!](\mathrm{m}_1) \leftarrow \mathscr{E}[\![exp]\!](\mathrm{m}_1)])) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathbf{assign}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc, exp, \mathrm{m}^\sharp_{\mathscr{R}}))$$

Definition 21 (Assignment for abstract heap transformation predicates domains) We define the assignment function $\mathbf{assign}_{\mathbb{T}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ for each abstract heap transformation predicates domain defined in Section 4.3.

1. The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$:

$$\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp) = \mathrm{t}^\sharp$$

2. The fields predicates domain, $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$:

$$\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp) = \mathrm{t}^\sharp \cup \{\mathtt{f}\}$$

3. The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}^\sharp_1 \times \mathbb{T}^\sharp_2$:

$$\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, (\mathrm{t}^\sharp_1, \mathrm{t}^\sharp_2)) = (\mathbf{assign}_{\mathbb{T}^\sharp_1}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp_1),\ \mathbf{assign}_{\mathbb{T}^\sharp_2}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp_2))$$

Theorem 11 (Soundness of Definition 21) *The operators from Definition 21 are sound in the sense of Condition 6.*

Example 13 (Assignment in transform-into relation) In this example, we consider the effect of $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha_1, \mathtt{f}, \beta_2, [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1]_{\mathrm{t}^\sharp_1})$, with $\mathrm{h}^\sharp_1 = \alpha_1 \cdot \mathtt{f} \mapsto \beta_1 *_{\mathtt{S}} \alpha_2 \cdot \mathtt{g} \mapsto \beta_2$, and $\mathrm{t}^\sharp_1 = (=^\sharp, \{\mathtt{g}\})$.

Applying the assignment in the combined predicates domain leads to apply the assignment in the footprint and the fields predicates abstract domains. We obtain that $\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha_1, \mathtt{f}, \beta_2, \mathrm{t}^\sharp_1) = (=^\sharp, \{\mathtt{g}, \mathtt{f}\})$. Then performing the assignment in $\mathrm{h}^\sharp_1$ produces that abstract heap $\mathrm{h}^\sharp_2 = \alpha_1 \cdot \mathtt{f} \mapsto \beta_2 *_{\mathtt{S}} \alpha_2 \cdot \mathtt{g} \mapsto \beta_2$. Finally, $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha_1, \mathtt{f}, \beta_2, [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1]_{\mathrm{t}^\sharp_1}) = [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_2]_{\mathrm{t}^\sharp_2}$ with $\mathrm{t}^\sharp_2 = (=^\sharp, \{\mathtt{g}, \mathtt{f}\})$.

Example 14 (Assignment in identity relation) In this example, we consider the effect of $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha_0, \mathtt{f}, \beta_1, \mathrm{r}^\sharp)$ with $\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0 *_{\mathtt{S}} \mathrm{h}_1^\sharp)$. We also admit that the analysis is using the combined predicates domain of the footprint and the fields predicates domains. After splitting $\mathrm{r}^\sharp$ into $\mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$ such as $\mathrm{r}_0^\sharp = \mathtt{Id}(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0)$ and $\mathrm{r}_1^\sharp = \mathtt{Id}(\mathrm{h}_1^\sharp)$, the analysis needs to call $\mathbf{id}_{\mathbb{T}^\sharp}$ on $\alpha_0 \cdot \mathtt{f} \mapsto \beta_0$ to weaken $\mathrm{r}_0^\sharp$. Applying the Definitions 6 , 9 and 12, $\mathbf{id}_{\mathbb{T}^\sharp}(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0)$ should return $\mathrm{t}_0^\sharp = (=^\sharp, \{\})$. Then, the weakening of $\mathrm{r}_0^\sharp$ is $[(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0) \dashrightarrow (\alpha_0 \cdot \mathtt{f} \mapsto \beta_0)]_{\mathrm{t}_0^\sharp}$. In turn, the analysis performs the assignment for abstract heap relations on the form $\mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$ and results $[(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0) \dashrightarrow (\alpha_0 \cdot \mathtt{f} \mapsto \beta_1)]_{\mathrm{t}_2^\sharp} *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp)$ with $\mathrm{t}_2^\sharp = (=^\sharp, \{\mathtt{f}\})$.

5.4 Allocations and Deallocations

In this section, we define the transfer functions for allocations $\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ and deallocations $\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ from abstract memory relations. We first comment on allocations.

5.4.1 Allocations.

The function $\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc, \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}, \mathrm{m}^\sharp_{\mathscr{R}})$ returns an abstract memory relation that over-approximates the allocation $loc = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\})$ in $\mathrm{m}^\sharp_{\mathscr{R}}$. It represents the creation of a new memory block of n cells (one cell per field of $\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$), and assigns the address of this new block to the location loc. The resulting abstract memory relation should express that the new block has been freshly allocated, and left the rest of the memory untouched.

The major part of the algorithm consists in the creation of a *single memory cell*. This is performed by the function $\mathbf{alloc}_{\mathbb{R}^\sharp}$ that inputs an abstract value β, a field $\mathtt{f}$ and an abstract heap relation $\mathrm{r}^\sharp$. It creates the cell $\beta \cdot \mathtt{f} \mapsto \delta$ (where δ is a fresh abstract value) and returns an abstract heap relation that expresses the allocation of this cell. The function $\mathbf{alloc}_{\mathbb{R}^\sharp}$ also builds upon the allocation for abstract heap transformation predicates $\mathbf{alloc}_{\mathbb{T}^\sharp}$.

Definition 22 (Allocation for abstract heap relations) We define the allocation function for abstract heap relations $\mathbf{alloc}_{\mathbb{R}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$:

$$\mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}, \mathrm{r}^\sharp) = \mathrm{r}^\sharp *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{f} \mapsto \delta)]_{\mathrm{t}^\sharp},$$

$$\text{where } \delta \text{ is fresh and } \mathrm{t}^\sharp = \mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta)$$

The definition of $\mathbf{alloc}_{\mathbb{R}^\sharp}$ ensures that the new cell has been freshly allocated thanks to the transform-into relation $[\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{f} \mapsto \delta)]_{\mathrm{t}^\sharp}$. It also ensures that the input abstract heap $\mathrm{r}^\sharp$ is not affected by the allocation, by separating it from the latter transform-into relation with the relational separating conjunction $*_{\mathtt{R}}$. The function $\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta)$ generates a new abstract heap transformation predicate $\mathrm{t}^\sharp$ that over-approximates the allocation of the cell $\beta \cdot \mathtt{f} \mapsto \delta$.

Condition 8 (Soundness of $\mathbf{alloc}_{\mathbb{T}^\sharp}$) *Let $\mathbb{T}^\sharp$ be an abstract heap predicates domain. Let $\beta, \delta \in \mathbb{V}^\sharp$ and $\mathtt{f} \in \mathbb{F}$. The function $\mathbf{alloc}_{\mathbb{T}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \to \mathbb{T}^\sharp$ is sound if:*

$$\{([],\ [\nu(\beta) + \mathtt{f} \mapsto \nu(\delta)],\ \nu) \in \mathbb{H} \times \mathbb{H} \times [\mathbb{V}^\sharp \to \mathbb{V}]\} \subseteq \gamma_{\mathbb{T}^\sharp}(\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta))$$

Theorem 12 (Soundness of $\mathbf{alloc}_{\mathbb{R}^\sharp}$) *Let $\beta \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}$ and $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$. The function $\mathbf{alloc}_{\mathbb{R}^\sharp}$ is sound if:*

$$(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \Longrightarrow \exists \mathrm{v} \in \mathbb{V}, (\mathrm{h_i}, \mathrm{h_o} \circledast [\nu(\beta) + \mathtt{f} \mapsto \mathrm{v}], \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}, \mathrm{r}^\sharp))$$

Definition 23 (Allocation in abstract memory relations)
Let $(\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ be an abstract memory relation and $n \geqslant 1$ the number of cells to allocate. We assume that $\mathbf{eval}_{\mathrm{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}_0^\sharp) = (\alpha, \mathrm{g})$ and that β is a fresh symbolic value. The abstract heap relations $\mathrm{r}_1^\sharp, \ldots, \mathrm{r}_n^\sharp$ are defined as follow:
$\forall i$ *s.t.* $1 \leqslant i \leqslant n, \mathrm{r}_i^\sharp = \mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$.
Finally, if $\mathrm{n}_1^\sharp = \mathbf{guard}_{\mathbb{N}^\sharp}((\beta \neq \mathbf{0x0}), \mathrm{n}_0^\sharp)$ and $\mathrm{r}_{n+1}^\sharp = \mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathrm{g}, \beta, \mathrm{r}_n^\sharp)$, then:

$$\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc,\ \{\mathtt{f}_1, \ldots, \mathtt{f}_n\},\ (\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)) = (\mathrm{e}^\sharp, \mathrm{r}_{n+1}^\sharp, \mathrm{n}_1^\sharp)$$

Like abstract assignments, the function $\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ evaluates the location loc with $\mathbf{eval}_{\mathrm{L}}$ into a pair (α, g) (this step may also unfold inductive predicates). It then generates a fresh symbolic value β, that is the base address of the block being created. For each fields $\mathtt{f}_i \in \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$, it applies $\mathbf{assign}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$, where $\mathrm{r}_{i-1}^\sharp$ has accumulated the allocations of the previous cells. Once the full block is allocated, it saves the fact that the address of the new block is not null with $\mathbf{guard}_{\mathbb{N}^\sharp}((\beta \neq \mathbf{0x0}), \mathrm{n}_0^\sharp)$ and assigns this address to the given location with $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathrm{g}, \beta, \mathrm{r}_n^\sharp)$.

Theorem 13 (Soundness of $\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathscr{R}}}$) *Let $loc \in L, \mathtt{f}_1, \ldots, \mathtt{f}_n \in \mathbb{F}$ and $\mathrm{m}^\sharp_{\mathscr{R}} \in \mathbb{M}^\sharp_{\mathscr{R}}$. If $(\mathrm{m}_0, (\mathrm{e}, \mathrm{h})) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}})$ then:*

$$\exists \mathrm{a}' \in \mathbb{A}, \mathrm{v}_1, \ldots, \mathrm{v}_n \in \mathbb{V},$$

$$(\mathrm{m}_0,\ (\mathrm{e},\ \mathrm{h}[\mathscr{L}[\![loc]\!](\mathrm{e}, \mathrm{h}) \leftarrow \mathrm{a}'] \circledast [\mathrm{a}' + \mathtt{f}_1 \mapsto \mathrm{v}_1, \ldots, \mathrm{a}' + \mathtt{f}_n \mapsto \mathrm{v}_n])) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc, \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}, \mathrm{m}^\sharp_{\mathscr{R}})$$

Definition 24 (Allocation for abstract heap transformation predicates domains) We define the allocation function $\mathbf{alloc}_{\mathbb{T}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \to \mathbb{T}^\sharp$ for each abstract heap transformation predicates domain defined in Section 4.3.

1. The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$:

$$\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta) = \subseteq^\sharp$$

2. The fields predicates domain, $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$:

$$\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta) = \{\}$$

3. The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp$:

$$\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta) = (\mathbf{alloc}_{\mathbb{T}_1^\sharp}(\beta, \mathtt{f}, \delta),\ \mathbf{alloc}_{\mathbb{T}_2^\sharp}(\beta, \mathtt{f}, \delta))$$

Theorem 14 (Soundness of Definition 24) *The operators from Definition 24 are sound in the sense of Condition 8.*

Example 15 (Allocation of a list element) We consider the analysis of the following statement, from the abstract memory relation $(e^\sharp, r^\sharp, n^\sharp)$, with $e^\sharp(\mathtt{p}) = \alpha_0$ and $r^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1)$:

$$\mathtt{p} = \mathbf{malloc}(\{\mathtt{next};\mathtt{data}\})$$

The abstract heap relation computed in this case is:

$$[(\alpha_0 \mapsto \alpha_1) \dashrightarrow (\alpha_0 \mapsto \beta)]_{t_0^\sharp} *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{next} \mapsto \beta_1)]_{t_1^\sharp} \\ *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{data} \mapsto \beta_2)]_{t_2^\sharp}$$

with $t_0^\sharp = (=^\sharp, \{0\})$ and $t_1^\sharp = t_2^\sharp = (\subseteq^\sharp, \{\})$.

5.4.2 Deallocations.

The function $\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc, (e^\sharp, r^\sharp, n^\sharp))$ returns an abstract memory relation that over-approximates the effect of the statement $\mathbf{free}(loc)$. It represents the deletion of the memory block pointed by loc. The resulting abstract memory relation should express the absence of cells that were present in its input state. Similarly to abstract allocation, the abstract deallocation of the memory block can be done cell per cell.

The function $\mathbf{free}_{\mathbb{R}^\sharp}$ inputs a symbolic value α, a field $\mathtt{f}$ and an abstract heap relation $r^\sharp$. It returns an abstract heap relation where the points-to predicate whose address is $\alpha \cdot \mathtt{f}$ has been deleted in the output abstract heap of $r^\sharp$. It also builds upon the function for deallocation of abstract heap transformation predicates $\mathbf{free}_{\mathbb{T}^\sharp}$.

Definition 25 (Deallocation for abstract heap relations) We define the deallocation function for abstract heap relations $\mathbf{free}_{\mathbb{R}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$:

$$\frac{t_1^\sharp = \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, t_0^\sharp)}{\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, [h_i^\sharp \dashrightarrow h_o^\sharp *_{\mathtt{S}} (\alpha \cdot \mathtt{f} \mapsto \beta)]_{t_0^\sharp}) = [h_i^\sharp \dashrightarrow h_o^\sharp]_{t_1^\sharp}}$$

$$\frac{\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r_1^\sharp) = r_2^\sharp}{\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r_0^\sharp *_{\mathtt{R}} r_1^\sharp) = r_0^\sharp *_{\mathtt{R}} r_2^\sharp}$$

$$\frac{t^\sharp = \mathbf{free}_{\mathbb{T}^\sharp}(\alpha,\ \mathtt{f},\ \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta))}{\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathtt{Id}(h^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)) = \mathtt{Id}(h^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \beta \dashrightarrow \mathbf{emp}]_{t^\sharp}}$$

To perform the deallocation in an abstract heap relation $r^\sharp$, the function $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r^\sharp)$ should express the absence of the memory cell at address $\alpha \cdot \mathtt{f}$ in the concrete output heaps of $r^\sharp$. If $(h_i,\ h_o \circledast [\nu(\alpha) + \mathtt{f} \mapsto v],\ \nu) \in \gamma_{\mathbb{R}^\sharp}(r^\sharp)$, then $(h_i,\ h_o,\ \nu)$ should be in $\gamma_{\mathbb{R}^\sharp}(\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r^\sharp))$. The definition of $\mathbf{free}_{\mathbb{R}^\sharp}$ follows the same principles than $\mathbf{assign}_{\mathbb{R}^\sharp}$ in Section 5.3:

- When $r^\sharp = [h_i^\sharp \dashrightarrow (h_o^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)]_{t_0^\sharp}$, the function $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r^\sharp)$ should return the abstract heap relation $[h_i^\sharp \dashrightarrow h_o^\sharp]_{t_1^\sharp}$. The function $\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, t_0^\sharp)$ returns the abstract heap transformation predicate $t_1^\sharp$ that over-approximates the deallocation of the cell from $t_0^\sharp$.
- When $r^\sharp = r_0^\sharp *_{\mathtt{R}} r_1^\sharp$, if the cell to delete is in $r_0^\sharp$, $\mathbf{free}_{\mathbb{R}^\sharp}$ is called recursively on $r_0^\sharp$ (we apply the Frame rule [39] but for abstract heap relations). Similarly to abstract assignment, if $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r_1^\sharp) = r_2^\sharp$, then $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, r_0^\sharp *_{\mathtt{R}} r_1^\sharp) = r_0^\sharp *_{\mathtt{R}} r_2^\sharp$.

- When $\mathrm{r}^\sharp = \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)$, the abstract deallocation proceeds exactly like the abstract assignment. It first splits $\mathrm{r}^\sharp$ into $\mathtt{Id}(\mathrm{h}^\sharp) *_{\mathtt{R}} \mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \beta)$, then weakens the right hand identity relation into $[(\alpha \cdot \mathtt{f} \mapsto \beta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \beta)]_{\mathrm{t}_0^\sharp}$ with $\mathrm{t}_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta)$, and perform the deallocation in the obtained transform into relation. It finally produces $\mathtt{Id}(\mathrm{h}^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \beta \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp}$ with $\mathrm{t}^\sharp = \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}_0^\sharp)$.

Condition 9 (Soundness of $\mathbf{free}_{\mathbb{T}^\sharp}$) *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}^\sharp \in \mathbb{T}^\sharp$. Let $\alpha \in \mathbb{V}^\sharp$ and $\mathtt{f} \in \mathbb{F}$. The function $\mathbf{free}_{\mathbb{T}^\sharp}$ is sound if:*

$$\{(\mathrm{h_i},\ \mathrm{h_o},\ \nu) \mid \exists \mathrm{v} \in \mathbb{V}, (\mathrm{h_i},\ \mathrm{h_o} \circledast [\nu(\alpha) + \mathtt{f} \mapsto \mathrm{v}],\ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)\} \\ \subseteq \\ \gamma_{\mathbb{T}^\sharp}(\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}^\sharp))$$

Theorem 15 (Soundness of $\mathbf{free}_{\mathbb{R}^\sharp}$) *Let $\alpha \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}$ and $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$. Then the function $\mathbf{free}_{\mathbb{R}^\sharp}$ is sound:*

$$\exists \mathrm{v} \in \mathbb{V}, (\mathrm{h_i}, \mathrm{h_o} \circledast [\nu(\alpha) + \mathtt{f} \mapsto \mathrm{v}], \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \Longrightarrow \\ (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp))$$

The function $\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ mainly builds upon $\mathbf{free}_{\mathbb{R}^\sharp}$. It also requires to evaluate the base address of the block to delete and to obtain the set of fields of the block. For simplicity, we admit that those fields are provided by the function **get_fields**. It inputs an abstract value α (that corresponds to the base address of the block), an abstract heap relation $\mathrm{r}^\sharp$ and returns the set of fields attached to α in $\mathrm{r}^\sharp$.

Definition 26 (Deallocation in abstract memory relations)
Let $(\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}^\sharp)$ be an abstract memory relation and $n \geqslant 1$ the number of cells to deallocate. We assume that $\mathbf{eval}_{\mathtt{E}}(loc, \mathrm{e}^\sharp, \mathrm{r}_0^\sharp) = (\alpha, \alpha)$ and $\{f_1, \ldots, \mathtt{f}_n\} = \mathbf{get_fields}(\alpha, \mathrm{r}_0^\sharp)$. The abstract heap relations $\mathrm{r}_1^\sharp, \ldots, \mathrm{r}_n^\sharp$ are defined as follow:
$\forall i, 1 \leqslant i \leqslant n, \mathrm{r}_i^\sharp = \mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$.
Finally we have:

$$\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc,\ (\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}^\sharp)) = (\mathrm{e}^\sharp, \mathrm{r}_n^\sharp, \mathrm{n}^\sharp)$$

The function $\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ evaluated the base address α of the block pointed by loc with $\mathbf{eval}_{\mathtt{E}}$. It obtains the fields of the cells to delete $\{f_1, \ldots, \mathtt{f}_n\}$ with $\mathbf{get_fields}(\alpha, \mathrm{r}_0^\sharp)$. For each field $\mathtt{f}_i \in \{f_1, \ldots, \mathtt{f}_n\}$, it applies $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$ where $\mathrm{r}_{i-1}^\sharp$ has accumulated the deletion of the previous cells. Finally, $\mathrm{r}_n^\sharp$ describes the deallocation of the entire block.

Theorem 16 (Soundness of $\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}$) *Let $loc \in L$ and $\mathrm{m}^\sharp_{\mathscr{R}} \in \mathbb{M}^\sharp_{\mathscr{R}}$.*
If $(\mathrm{m}_0, (\mathrm{e}, \mathrm{h}_1 \circledast \mathrm{h}'_1)) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}})$ *such that:*

$$\mathbf{dom}(\mathrm{h}'_1) = \{\mathscr{E}[\![loc]\!](\mathrm{e}, \mathrm{h}_1) + \mathtt{f} \mid \mathtt{f} \in \mathbb{F}\}$$

then:

$$(\mathrm{m}_0, (\mathrm{e}, \mathrm{h}_1)) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathbf{free}_{\mathbb{M}^\sharp_{\mathscr{R}}}(loc, \mathrm{m}^\sharp_{\mathscr{R}}))$$

Definition 27 (Deallocation for abstract heap transformation predicates domains) We define the deallocation function $\mathbf{free}_{\mathbb{T}^\sharp} : \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ for each abstract heap transformation predicates domain defined in Section 4.3.

1. The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$:

$$\begin{aligned}
\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, =^\sharp) &= \supseteq^\sharp \\
\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \supseteq^\sharp) &= \supseteq^\sharp \\
\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \subseteq^\sharp) &= \top \\
\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \top) &= \top
\end{aligned}$$

2. The fields predicates domain, $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$:

$$\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}^\sharp) = \mathrm{t}^\sharp$$

3. The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}^\sharp_1 \times \mathbb{T}^\sharp_2$:

$$\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, (\mathrm{t}^\sharp_1, \mathrm{t}^\sharp_2)) = (\mathbf{free}_{\mathbb{T}^\sharp_1}(\alpha, \mathtt{f}, \mathrm{t}^\sharp_1),\ \mathbf{free}_{\mathbb{T}^\sharp_2}(\alpha, \mathtt{f}, \mathrm{t}^\sharp_2))$$

Theorem 17 (Soundness of Definition 27) *The operators from Definition 27 are sound in the sense of Condition 9.*

Example 16 (Deallocation of a list element) We show the effect of the analysis of the deallocation of a list element, from the abstract memory relation $(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$, with $\mathrm{e}^\sharp(\mathtt{p}) = \alpha_0$, $\mathrm{t}^\sharp_0 = (=^\sharp, \{\mathtt{data}\})$ and $\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{next} \mapsto \alpha_2) *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow (\mathrm{h}^\sharp_{\mathrm{o}} *_{\mathtt{S}} \alpha_1 \cdot \mathtt{data} \mapsto \alpha_3)]_{\mathrm{t}^\sharp_0}$.

Thus, the abstract heap relation computed for the instruction $\mathbf{free}(\mathtt{p})$ is: $\mathtt{Id}(\alpha_0 \mapsto \alpha_1) *_{\mathtt{R}} [(\alpha_1 \cdot \mathtt{next} \mapsto \alpha_2) \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp_1} *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]_{\mathrm{t}^\sharp_2}$ with $\mathrm{t}^\sharp_1 = (\supseteq^\sharp, \{\})$ and $\mathrm{t}^\sharp_2 = (\supseteq^\sharp, \{\mathtt{data}\})$.

5.4.3 Local Variables Initialization and Deletion.

The declaration or the initialization of a local variable to a program block is also considered as an allocation. This is justified by the fact that we do not distinguish the heap and the stack, and that a new program variable does not belong to the initial input memory state. Similarly, when we exit a program block, we deallocate all the addresses of local variables to this block.

5.5 Condition Tests

In this section, we define the transfer function $\mathbf{guard}_{\mathbb{M}^\sharp_{\mathscr{R}}} : \mathtt{E} \times \mathbb{M}^\sharp_{\mathscr{R}} \to \mathbb{M}^\sharp_{\mathscr{R}}$ for condition tests. It evaluates the boolean expression of a condition test and returns an abstract memory relation that has taken into account the effects of the expression. It first translates the expression into a pure formula with $\mathbf{eval}_{\mathtt{E}}$ (note that this step may perform materialization). Finally, it updates the abstract numerical value interpreting the pure formula with the function $\mathbf{guard}_{\mathbb{N}^\sharp}$, introduced in Section 5.2.

Definition 28 (Condition test in abstract memory relations)
Let $(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp_0)$ be an abstract memory relation and *exp* an expression.
If $\mathbf{eval}_{\mathtt{E}}(exp, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathrm{p}^\sharp)$ and $\mathrm{n}^\sharp_1 = \mathbf{guard}_{\mathbb{N}^\sharp}(\mathrm{p}^\sharp, \mathrm{n}^\sharp_0)$, then:

$$\mathbf{guard}_{\mathbb{M}^\sharp_{\mathscr{R}}}(exp, (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp_0)) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp_1)$$

Theorem 18 (Soundness of $\textbf{guard}_{\mathbb{M}^\sharp_{\mathscr{R}}}$)
Let $exp \in E$ *and* $\mathrm{m}^\sharp_{\mathscr{R},0}, \mathrm{m}^\sharp_{\mathscr{R},1} \in \mathbb{M}^\sharp_{\mathscr{R}}$.

$$\text{If } \mathrm{m}^\sharp_{\mathscr{R},1} = \textbf{guard}_{\mathbb{M}^\sharp_{\mathscr{R}}}(exp, \mathrm{m}^\sharp_{\mathscr{R},0}), \text{ then:}$$

$$\forall (\mathrm{m_i}, \mathrm{m_o}) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},0}), \qquad \mathscr{E}[\![exp]\!](\mathrm{m_o}) \neq 0 \implies (\mathrm{m_i}, \mathrm{m_o}) \in \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},1})$$

Example 17 (Condition test) Consider the following condition test:

$$\textbf{if}(\texttt{l->next} == \textbf{0x0})$$

applied to the abstract memory relation $\mathrm{m}^\sharp_{\mathscr{R}} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ where

$$\mathrm{r}^\sharp = \texttt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathrm{S}} \textbf{list}(\alpha_1)) \text{ and } \mathrm{e}^\sharp(\texttt{l}) = \alpha_0$$

The analysis first unfolds the inductive predicate $\textbf{list}(\alpha_1)$ into a points-to predicate and obtains the following abstract heap relation:

$$\texttt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathrm{S}} \alpha_1 \cdot \texttt{data} \mapsto \delta *_{\mathrm{S}} \alpha_1 \cdot \texttt{next} \mapsto \alpha_2 *_{\mathrm{S}} \textbf{list}(\alpha_2))$$

It then saves the constraint $\alpha_2 = \textbf{0x0}$ in the numerical abstract value $\mathrm{n}^\sharp$.

5.6 Inclusion Checking

Like classical shape analyses [25, 10], our analysis needs to *fold* inductive predicates so as to (conservatively) decide inclusion and join abstract states. We first present the inclusion checking algorithm in this section.

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 $\textbf{isle}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ that inputs two abstract memory relations $\mathrm{m}^\sharp_{\mathscr{R},0} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$ and $\mathrm{m}^\sharp_{\mathscr{R},1} = (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1)$, that either returns **true** (meaning that the inclusion of concretizations holds) or **false** (meaning that the analysis cannot conclude that inclusion holds).

An important feature of inclusion checking is that the underlying abstract heaps may have *distinct* sets of symbolic values. Yet, to compare abstract heaps, the algorithm requires to compare symbolic values. That is, the inclusion checking algorithm requires a *renaming function* $\Psi : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ that maps symbolic values of $\mathrm{m}^\sharp_{\mathscr{R},1}$ into symbolic values of $\mathrm{m}^\sharp_{\mathscr{R},0}$ in order to maintain a notion of equivalence between symbolic values.

The definition of inclusion checking consists in three steps: the *generation of an initial renaming function* from the abstract environments of the two abstract memory relations, the *inclusion checking in abstract heap relations* that checks inclusion but also refines the renaming function, and the *inclusion checking in the numerical abstract domain* that makes use of the latter renaming function.

Generation of an initial renaming function. First, the abstract environment domain generates an initial renaming function Ψ_{init}. It is clear that each program variable should be mapped to the same address, thus the initial renaming function is defined as follows: $\forall \texttt{x} \in \mathbb{X}, \Psi_{\mathrm{init}}(\mathrm{e}^\sharp_1(\texttt{x})) = \mathrm{e}^\sharp_0(\texttt{x})$.

$$\frac{h^\sharp \text{ is not of the form } h_0^\sharp *_S h_1^\sharp}{h^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h^\sharp} \ (\sqsubseteq_=)$$

$$\frac{h^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{list}(\beta)}{\mathbf{listseg}(\alpha,\beta) *_S h^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{list}(\alpha)} \ (\sqsubseteq_{\text{seg}}) \qquad \frac{h_{0,0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_{1,0}^\sharp \quad h_{0,1}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_{1,1}^\sharp}{h_{0,0}^\sharp *_S h_{0,1}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_{1,0}^\sharp *_S h_{1,1}^\sharp} \ (\sqsubseteq_{*_S})$$

$$\frac{\alpha \text{ carries an inductive predicate in } r_1^\sharp \quad (r_u^\sharp, p^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, r_1^\sharp) \quad r_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_u^\sharp}{r_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_1^\sharp} \ (\sqsubseteq_{\text{unfold}})$$

$$\frac{h_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_1^\sharp}{\mathtt{Id}(h_0^\sharp) \sqsubseteq_{\mathbb{R}^\sharp} \mathtt{Id}(h_1^\sharp)} \ (\sqsubseteq_{\mathtt{Id}}) \qquad \frac{t_0^\sharp \sqsubseteq_{\mathbb{T}^\sharp} t_1^\sharp \quad h_{i,0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_{i,1}^\sharp \quad h_{o,0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_{o,1}^\sharp}{[h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp]_{t_0^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp}} \ (\sqsubseteq_{\dashrightarrow})$$

$$\frac{r_{0,0}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_{1,0}^\sharp \quad r_{0,1}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_{1,1}^\sharp}{r_{0,0}^\sharp *_R r_{0,1}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_{1,0}^\sharp *_R r_{1,1}^\sharp} \ (\sqsubseteq_{*_R})$$

$$\frac{\mathtt{Id}(h_0^\sharp) *_R \mathtt{Id}(h_1^\sharp) *_R r^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_1^\sharp}{\mathtt{Id}(h_0^\sharp *_S h_1^\sharp) *_R r^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_1^\sharp} \ (\sqsubseteq_{\mathtt{Id}-\text{split}_L}) \qquad \frac{r_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathtt{Id}(h_0^\sharp) *_R \mathtt{Id}(h_1^\sharp) *_R r^\sharp}{r_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathtt{Id}(h_0^\sharp *_S h_1^\sharp) *_R r^\sharp} \ (\sqsubseteq_{\mathtt{Id}-\text{split}_R})$$

$$\frac{t_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(h^\sharp) \quad r^\sharp *_R [h^\sharp \dashrightarrow h^\sharp]_{t_0^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [h_i^\sharp \dashrightarrow h_o^\sharp]_{t_1^\sharp}}{r^\sharp *_R \mathtt{Id}(h^\sharp) \sqsubseteq_{\mathbb{R}^\sharp} [h_i^\sharp \dashrightarrow h_o^\sharp]_{t_1^\sharp}} \ (\sqsubseteq_{\mathtt{Id}-\text{weak}})$$

$$\frac{t_2^\sharp = t_0^\sharp *_T t_1^\sharp \quad r^\sharp *_R [h_{i,0}^\sharp *_S h_{i,1}^\sharp \dashrightarrow h_{o,0}^\sharp *_S h_{o,1}^\sharp]_{t_2^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [h_i^\sharp \dashrightarrow h_o^\sharp]_{t^\sharp}}{r^\sharp *_R [h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp]_{t_0^\sharp} *_R [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [h_i^\sharp \dashrightarrow h_o^\sharp]_{t^\sharp}} \ (\sqsubseteq_{\dashrightarrow-\text{weak}})$$

Fig. 11: Inclusion checking rules

Inclusion checking in abstract heap relations. Then, the analysis proceeds to the inclusion checking of two abstract heap relations. It consists of a function $\mathbf{isle}_{\mathbb{R}^\sharp}(\Psi, r_0^\sharp, r_1^\sharp)$ over the abstract heap relations $r_0^\sharp$ and $r_1^\sharp$ where Ψ is an initial renaming function. The inclusion holds if it returns $(\Psi', \mathbf{true})$ where Ψ' is the final renaming function. It requires a function $\mathbf{isle}_{\mathbb{H}^\sharp}(\Psi, h_0^\sharp, h_1^\sharp)$ that returns $(\Psi', \mathbf{true})$ if the inclusion of abstract heaps $h_0^\sharp$ and $h_1^\sharp$ holds and extends Ψ into Ψ'. It also requires a conservative function $\mathbf{isle}_{\mathbb{T}^\sharp}$ over abstract heap transformation predicates.

The definition of $\mathbf{isle}_{\mathbb{R}^\sharp}$, $\mathbf{isle}_{\mathbb{H}^\sharp}$ and $\mathbf{isle}_{\mathbb{T}^\sharp}$ relies on a conservative algorithm, that implements a proof search, based on the rules shown in Figure 11 (for clarity, we omit the pure formulas inclusion checking). We do not detail the algorithm to compute $\mathbf{isle}_{\mathbb{R}^\sharp}$, $\mathbf{isle}_{\mathbb{H}^\sharp}$ and $\mathbf{isle}_{\mathbb{T}^\sharp}$, we rather focus on the correctness of the rules system on which they are based.

In this rules system, we assume that the renaming function has already been performed on abstract heaps. Thus, we do not show how the renaming function is extended (this is made fully explicit in [10, Figure 6]), as the issue is orthogonal to the reasoning over abstract heap relations which is the goal of this paper. This rules system is based on three operators, $\sqsubseteq_{\mathbb{H}^\sharp}$, $\sqsubseteq_{\mathbb{R}^\sharp}$ and $\sqsubseteq_{\mathbb{T}^\sharp}$, that respectively reason over abstract heaps, abstract heap relations and abstract heap transformation predicates. They satisfy the following properties:

$$h_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} h_1^\sharp \Longrightarrow \gamma_{\mathbb{H}^\sharp}(h_0^\sharp) \subseteq \gamma_{\mathbb{H}^\sharp}(h_1^\sharp)$$

$$r_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} r_1^\sharp \Longrightarrow \gamma_{\mathbb{R}^\sharp}(r_0^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(r_1^\sharp)$$

$$t_0^\sharp \sqsubseteq_{\mathbb{T}^\sharp} t_1^\sharp \Longrightarrow \gamma_{\mathbb{T}^\sharp}(t_0^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(t_1^\sharp)$$

The rules $(\sqsubseteq_=)$, $(\sqsubseteq_{\text{seg}})$ and $(\sqsubseteq_{*_S})$ are specific to reasoning about abstract heaps, and are directly inspired from [10, Figure 6] (they allow to reason over equal abstract regions, over segments, and over separating conjunction). The rule $(\sqsubseteq_{\text{unfold}})$ allows to unfold inductive predicates as part of the inclusion checking process, at the level of relations. The rules $(\sqsubseteq_{\mathtt{Id}})$ and $(\sqsubseteq_{\dashrightarrow})$ are the canonic rules for abstract heap relations. They apply $\sqsubseteq_{\mathbb{H}^\sharp}$ on the abstract heaps contained in the abstract heap relations. The rule $(\sqsubseteq_{*_R})$ makes inclusion checking as local. Finally, the rules $(\sqsubseteq_{\mathtt{Id}-\text{split}_L})$, $(\sqsubseteq_{\mathtt{Id}-\text{split}_R})$, $(\sqsubseteq_{\mathtt{Id}-\text{weak}})$, and $(\sqsubseteq_{\dashrightarrow-\text{weak}})$ allow to derive inclusion over abstract heap relations, and implement the properties observed in Theorem 1 (page 14) and Theorem 2 (page 17). Thus, their correctness derives from these properties. The proof search algorithm starts from the goal to prove, and attempts to apply these rules so as to get a complete derivation (i.e. a formal proof for which each step is one of the inclusion rule).

Inclusion checking in the numerical abstract domain. Finally, the analysis proceeds to the inclusion checking between the two abstract numerical values $n_0^\sharp$ and $n_1^\sharp$, modulo the renaming function Ψ' that results from $\mathbf{isle}_{\mathbb{R}^\sharp}$. Thus the numerical abstract domain should provide a function $\mathbf{isle}_{\mathbb{N}^\sharp} : [\mathbb{V}^\sharp \to \mathbb{V}^\sharp] \times \mathbb{N}^\sharp \times \mathbb{N}^\sharp \to \{\mathbf{true}, \mathbf{false}\}$.

Definition 29 (Inclusion checking in abstract memory relations) Let $m_{\mathscr{R},0}^\sharp = (e_0^\sharp, r_0^\sharp, n_0^\sharp) \in \mathbb{M}_{\mathscr{R}}^\sharp$ and $m_{\mathscr{R},1}^\sharp = (e_1^\sharp, r_1^\sharp, n_1^\sharp) \in \mathbb{M}_{\mathscr{R}}^\sharp$.

$$\forall \mathtt{x} \in \mathbb{X}, \text{Let } \Psi_{\text{init}}(e_1^\sharp(\mathtt{x})) = e_0^\sharp(\mathtt{x}).$$

$$\text{If } \mathbf{isle}_{\mathbb{R}^\sharp}(\Psi_{\text{init}}, r_0^\sharp, r_1^\sharp) = (\Psi', \mathbf{true}) \text{ and } \mathbf{isle}_{\mathbb{N}^\sharp}(\Psi', n_0^\sharp, n_1^\sharp) = \mathbf{true}, \text{ then:}$$

$$\mathbf{isle}_{\mathbb{M}_{\mathscr{R}}^\sharp}(m_{\mathscr{R},0}^\sharp, m_{\mathscr{R},1}^\sharp) = \mathbf{true}$$

That is, the function $\mathbf{isle}_{\mathbb{M}_{\mathscr{R}}^\sharp}$ calls $\mathbf{isle}_{\mathbb{R}^\sharp}$ with the initial renaming function Ψ_{init}. If the inclusion holds for the two abstract heap relations $r_0^\sharp$ and $r_1^\sharp$, it tests the inclusion of the abstract numerical values $n_0^\sharp$ and $n_1^\sharp$ with the resulting renaming function of $\mathbf{isle}_{\mathbb{R}^\sharp}$.

Theorem 19 (Soundness of inclusion checking) *If* $h_0^\sharp, h_1^\sharp \in \mathbb{H}^\sharp$, $r_0^\sharp, r_1^\sharp \in \mathbb{R}^\sharp$, $\Psi, \Psi' : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $m_{\mathscr{R},0}^\sharp, m_{\mathscr{R},1}^\sharp \in \mathbb{M}_{\mathscr{R}}^\sharp$ *then:*

$$\begin{aligned}&\mathbf{isle}_{\mathbb{H}^\sharp}(\Psi, h_0^\sharp, h_1^\sharp) = (\Psi', \mathbf{true})\\ &\quad\Longrightarrow \forall (h, \nu) \in \gamma_{\mathbb{H}^\sharp}(h_0^\sharp), (h, \Psi' \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(h_1^\sharp)\\ &\qquad \wedge \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi(\alpha) = \beta \Rightarrow \Psi'(\alpha) = \beta\end{aligned}$$

$$\begin{aligned}&\mathbf{isle}_{\mathbb{R}^\sharp}(\Psi, r_0^\sharp, r_1^\sharp) = (\Psi', \mathbf{true})\\ &\quad\Longrightarrow \forall (h_i, h_o, \nu) \in \gamma_{\mathbb{R}^\sharp}(r_0^\sharp), (h_i, h_o, \Psi' \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(r_1^\sharp)\\ &\qquad \wedge \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi(\alpha) = \beta \Rightarrow \Psi'(\alpha) = \beta\end{aligned}$$

$$\begin{aligned}&\mathbf{isle}_{\mathbb{M}_{\mathscr{R}}^\sharp}(m_{\mathscr{R},0}^\sharp, m_{\mathscr{R},1}^\sharp) = \mathbf{true}\\ &\quad\Longrightarrow \gamma_{\mathbb{M}_{\mathscr{R}}^\sharp}(m_{\mathscr{R},0}^\sharp) \subseteq \gamma_{\mathbb{M}_{\mathscr{R}}^\sharp}(m_{\mathscr{R},1}^\sharp)\end{aligned}$$

The soundness of this definition requires that the functions $\mathbf{isle}_{\mathbb{N}^\sharp}$ and $\mathbf{isle}_{\mathbb{T}^\sharp}$ satisfy respectively the following conditions:

Condition 10 (Soundness of $\mathbf{isle}_{\mathbb{N}^\sharp}$) *Let* $\Psi : (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)$ *and* $n_0^\sharp, n_1^\sharp \in \mathbb{N}^\sharp$. *Then:*

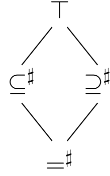
$$\mathbf{isle}_{\mathbb{N}^\sharp}(\Psi, n_0^\sharp, n_1^\sharp) = \mathbf{true} \quad \Rightarrow \quad \forall \nu \in \gamma_{\mathbb{N}^\sharp}(n_0^\sharp), \quad \Psi \circ \nu \in \gamma_{\mathbb{N}^\sharp}(n_1^\sharp)$$

Condition 11 (Soundness of $\mathbf{isle}_{\mathbb{T}^\sharp}$) *Let* $\Psi : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $t_0^\sharp, t_1^\sharp \in \mathbb{T}^\sharp$,
if $\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, t_0^\sharp, t_1^\sharp) = \mathbf{true}$ *then:*

$$(h_i, h_o, \nu) \in \gamma_{\mathbb{T}^\sharp}(t_0^\sharp) \Rightarrow (h_i, h_o, \Psi \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(t_1^\sharp)$$

Definition 30 (Inclusion in abstract heap transformation predicates domains) We define the inclusion function $\mathbf{isle}_{\mathbb{T}^\sharp} : [\mathbb{V}^\sharp \to \mathbb{V}^\sharp] \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \{\mathbf{true}, \mathbf{false}\}$ for each abstract heap transformation predicates domain defined in Section 4.3.

1. The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$: the function $\mathbf{isle}_{\mathbb{T}^\sharp}$ is defined using the Hasse diagram of $\mathbb{T}^\sharp$:



2. The fields predicates domain, $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$:

$$\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, t_0^\sharp, t_1^\sharp) = t_0^\sharp \subseteq t_1^\sharp$$

3. The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}_a^\sharp \times \mathbb{T}_b^\sharp$:

$$\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, (t_{a,0}^\sharp, t_{b,0}^\sharp), (t_{a,1}^\sharp, t_{b,1}^\sharp)) = \mathbf{isle}_{\mathbb{T}_a^\sharp}(\Psi, t_{a,0}^\sharp, t_{a,1}^\sharp) \wedge \mathbf{isle}_{\mathbb{T}_b^\sharp}(\Psi, t_{b,0}^\sharp, t_{b,1}^\sharp)$$

Theorem 20 (Soundness of Definition 30) *The operators from Definition 30 are sound in the sense of Condition 11.*

The soundness of this definition is proven using respectively Lemma 1 (page 19), Lemma 2 (page 21) and Theorem 5 (page 22).

Example 18 (Inclusion checking) Let us consider the following abstract heap relations $r_0^\sharp$ and $r_1^\sharp$, and discuss the computation of $\mathbf{isle}_{\mathbb{R}^\sharp}(\Psi, r_0^\sharp, r_1^\sharp)$. For simplicity, we suppose that $r_0^\sharp$ and $r_1^\sharp$ share the same set of symbolic values (i.e. the renaming function Ψ has been already computed and applied):

$$r_0^\sharp = \mathtt{Id}(\alpha \cdot \mathtt{next} \mapsto \alpha_0 *_\mathtt{S} \mathbf{list}(\alpha_0)) *_\mathtt{R} [\alpha \cdot \mathtt{data} \mapsto \alpha_1 \dashrightarrow \alpha \cdot \mathtt{data} \mapsto \alpha_2]_{t_0^\sharp},$$
$$\text{with } t_0^\sharp = (=^\sharp, \{\mathtt{data}\})$$
$$r_1^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]_{t_1^\sharp}, \text{ with } t_1^\sharp = (=^\sharp, \{\mathtt{data}, \mathtt{next}\})$$

Using first the rule ($\sqsubseteq_{\mathtt{Id}-\mathrm{weak}}$) then the rule ($\sqsubseteq_{\dashrightarrow-\mathrm{weak}}$), this goal gets reduced into checking the inclusion $[h_0^\sharp \dashrightarrow h_1^\sharp]_{t_2^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} r_1^\sharp$, where $h_0^\sharp = \alpha \cdot \mathtt{next} \mapsto \alpha_0 *_\mathtt{S} \mathbf{list}(\alpha_0) *_\mathtt{S} \alpha \cdot \mathtt{data} \mapsto \alpha_1$ and $h_1^\sharp = \alpha \cdot \mathtt{next} \mapsto \alpha_0 *_\mathtt{S} \mathbf{list}(\alpha_0) *_\mathtt{S} \alpha \cdot \mathtt{data} \mapsto \alpha_2$ and $t_2^\sharp = (=^\sharp, \{\mathtt{data}\})$. In turn, this inclusion follows from rule ($\sqsubseteq_{\mathrm{unfold}}$).

$$\frac{h^\sharp_0 \sqcup_{\mathbb{H}^\sharp} h^\sharp_1 \rightsquigarrow h^\sharp}{\mathtt{Id}(h^\sharp_0) \sqcup_{\mathbb{R}^\sharp} \mathtt{Id}(h^\sharp_1) \rightsquigarrow \mathtt{Id}(h^\sharp)} \; (\sqcup_{\mathtt{Id}})$$

$$\frac{t^\sharp_0 \sqcup_{\mathbb{T}^\sharp} t^\sharp_1 \rightsquigarrow t^\sharp \qquad h^\sharp_{i,0} \sqcup_{\mathbb{H}^\sharp} h^\sharp_{i,1} \rightsquigarrow h^\sharp_i \qquad h^\sharp_{o,0} \sqcup_{\mathbb{H}^\sharp} h^\sharp_{o,1} \rightsquigarrow h^\sharp_o}{[h^\sharp_{i,0} \dashrightarrow h^\sharp_{o,0}]_{t^\sharp_0} \sqcup_{\mathbb{R}^\sharp} [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1} \rightsquigarrow [h^\sharp_i \dashrightarrow h^\sharp_o]_{t^\sharp}} \; (\sqcup_{\dashrightarrow})$$

$$\frac{r^\sharp_{0,0} \sqcup_{\mathbb{R}^\sharp} r^\sharp_{1,0} \rightsquigarrow r^\sharp_0 \qquad r^\sharp_{0,1} \sqcup_{\mathbb{R}^\sharp} r^\sharp_{1,1} \rightsquigarrow r^\sharp_1}{r^\sharp_{0,0} *_{\mathtt{R}} r^\sharp_{0,1} \sqcup_{\mathbb{R}^\sharp} r^\sharp_{1,0} *_{\mathtt{R}} r^\sharp_{1,1} \rightsquigarrow r^\sharp_0 *_{\mathtt{R}} r^\sharp_1} \; (\sqcup_{*_{\mathtt{R}}})$$

$$\frac{t^\sharp_0 = \mathbf{id}_{\mathbb{T}^\sharp}(h^\sharp_0) \qquad [h^\sharp_0 \dashrightarrow h^\sharp_0]_{t^\sharp_0} \sqcup_{\mathbb{R}^\sharp} [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1} \rightsquigarrow r^\sharp}{\mathtt{Id}(h^\sharp_0) \sqcup_{\mathbb{R}^\sharp} [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1} \rightsquigarrow r^\sharp} \; (\sqcup_{\mathtt{Id}-\mathrm{weak}})$$

$$\frac{\mathtt{Id}(h^\sharp_0 *_{\mathtt{S}} h^\sharp_1) *_{\mathtt{R}} r^\sharp_0 \sqcup_{\mathbb{R}^\sharp} r^\sharp_1 \rightsquigarrow r^\sharp}{\mathtt{Id}(h^\sharp_0) *_{\mathtt{R}} \mathtt{Id}(h^\sharp_1) *_{\mathtt{R}} r^\sharp_0 \sqcup_{\mathbb{R}^\sharp} r^\sharp_1 \rightsquigarrow r^\sharp} \; (\sqcup_{\mathtt{Id}-\mathrm{merge}})$$

$$\frac{t^\sharp = t^\sharp_0 *_{\mathtt{T}} t^\sharp_1 \qquad [h^\sharp_{i,0} *_{\mathtt{S}} h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,0} *_{\mathtt{S}} h^\sharp_{o,1}]_{t^\sharp} *_{\mathtt{R}} r^\sharp_0 \sqcup_{\mathbb{R}^\sharp} r^\sharp_1 \rightsquigarrow r^\sharp}{[h^\sharp_{i,0} \dashrightarrow h^\sharp_{o,0}]_{t^\sharp_0} *_{\mathtt{R}} [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1} *_{\mathtt{R}} r^\sharp_0 \sqcup_{\mathbb{R}^\sharp} r^\sharp_1 \rightsquigarrow r^\sharp} \; (\sqcup_{\dashrightarrow-\mathrm{weak}})$$

$$\frac{t^\sharp_0 = \mathbf{id}_{\mathbb{T}^\sharp}(h^\sharp_0) \qquad [h^\sharp_0 \dashrightarrow h^\sharp_0]_{t^\sharp_0} *_{\mathtt{R}} [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1} *_{\mathtt{R}} r^\sharp_0 \sqcup_{\mathbb{R}^\sharp} r^\sharp_1 \rightsquigarrow r^\sharp}{\mathtt{Id}(h^\sharp_0) *_{\mathtt{R}} [h^\sharp_{i,1} \dashrightarrow h^\sharp_{o,1}]_{t^\sharp_1} *_{\mathtt{R}} r^\sharp_0 \sqcup_{\mathbb{R}^\sharp} r^\sharp_1 \rightsquigarrow r^\sharp} \; (\sqcup_{\dashrightarrow-\mathrm{intro}})$$

Fig. 12: Join rewriting rules

5.7 Join and Widening

5.7.1 Join operator.

In the following, we define the abstract operator $\mathbf{join}_{\mathbb{M}^\sharp_{\mathscr{R}}}$. This operator computes an over-approximation of the union of abstract memory relations. Like for the inclusion checking, $\mathbf{join}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ inputs two abstract memory relations $m^\sharp_{\mathscr{R},0} = (e^\sharp_0, r^\sharp_0, n^\sharp_0)$ and $m^\sharp_{\mathscr{R},1} = (e^\sharp_1, r^\sharp_1, n^\sharp_1)$, but instead of a boolean, outputs a new abstract memory relation $m^\sharp_{\mathscr{R}} = (e^\sharp, r^\sharp, n^\sharp)$. It is defined such that the union of the concretizations of $m^\sharp_{\mathscr{R},0}$ and $m^\sharp_{\mathscr{R},1}$ is included in the concretization of $m^\sharp_{\mathscr{R}}$.

The creation of a new abstract memory relation implies the creation of new symbolic values. Thus, the join operator needs to maintain a relation between the symbolic values of its two arguments and the resulting symbolic values. Slightly differently than the inclusion checking, the join requires a pair of renaming functions $\Phi = (\Psi_0, \Psi_1)$ that map each output symbolic value to the pair of the two corresponding input symbolic values. For instance, if α is a resulting symbolic value of the join operator, we note $\Phi(\alpha) = (\alpha_0, \alpha_1)$ if $\Psi_0(\alpha) = \alpha_0$ and $\Psi_1(\alpha) = \alpha_1$. The join algorithm proceeds in the same way as the inclusion checking, following the three same main steps: *initialization* that creates the initial pair of renaming functions, *join of abstract heap relations* that joins two abstract heap relations and *join in the numerical abstract domain* that joins the two abstract numerical values.

Initialization. The join operation starts with the initialization of the pair of renaming functions and the generation of the resulting abstract environment $e^\sharp$ as follows: $\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{init}}(\alpha) = (e_0^\sharp(\mathtt{x}), e_1^\sharp(\mathtt{x}))$ and $e^\sharp(\mathtt{x}) = \alpha$.

Join of abstract heap relations. The join operation then proceeds to the computation of the new abstract heap relation $r^\sharp$. This is done by the functions, $\mathbf{join}_{\mathbb{R}^\sharp}$, $\mathbf{join}_{\mathbb{H}^\sharp}$ and $\mathbf{join}_{\mathbb{T}^\sharp}$ that operate respectively on abstract heap relations, abstract heaps and abstract heap predicate transformations. The functions $\mathbf{join}_{\mathbb{R}^\sharp}$, $\mathbf{join}_{\mathbb{H}^\sharp}$ also input and extend the pair of renaming functions.

The algorithm to compute these functions follows the same principle than inclusion checking. It implements rewriting rules, given in Figure 12. This rules system is based on three operators: $\sqcup_{\mathbb{H}^\sharp}$ reasoning on abstract heaps , $\sqcup_{\mathbb{R}^\sharp}$ reasoning on abstract heap relations and $\sqcup_{\mathbb{T}^\sharp}$ reasoning on abstract heap transformation predicates. They satisfy the following properties:

$$h_0^\sharp \sqcup_{\mathbb{H}^\sharp} h_1^\sharp \rightsquigarrow h^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{H}^\sharp}(h_0^\sharp) \cup \gamma_{\mathbb{H}^\sharp}(h_1^\sharp) \subseteq \gamma_{\mathbb{H}^\sharp}(h^\sharp)$$

$$r_0^\sharp \sqcup_{\mathbb{R}^\sharp} r_1^\sharp \rightsquigarrow r^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{R}^\sharp}(r_0^\sharp) \cup \gamma_{\mathbb{R}^\sharp}(r_1^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(r^\sharp)$$

$$t_0^\sharp \sqcup_{\mathbb{T}^\sharp} t_1^\sharp \rightsquigarrow t^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{T}^\sharp}(t_0^\sharp) \cup \gamma_{\mathbb{T}^\sharp}(t_1^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(t^\sharp)$$

Note that the rules for $\sqcup_{\mathbb{H}^\sharp}$ are given in [10, Figure 7]. Here, we discuss the rules of Figure 12. The rules $(\sqcup_{\mathtt{Id}})$ and $(\sqcup_{\dashrightarrow})$ are applied on two abstract heap relations that consist of the same relational connective (respectively $\mathtt{Id}(.)$ and $[. \dashrightarrow .]_.$). They simply apply $\sqcup_{\mathbb{H}^\sharp}$ on the abstract heaps they contain and conserve the relational connective. The rule $(\sqcup_{*_\mathtt{R}})$ is based on the separation principle and allows to apply the other rules independently. The next rules can all be applied symmetrically and follow the principles of Theorem 1 (page 14) and Theorem 2 (page 17). When applied to an identity relation and a transform-into relation, the rule $(\sqcup_{\mathtt{Id}-\text{weak}})$ first weakens the identity relation into a transform-into relation and applies recursively $\sqcup_{\mathbb{R}^\sharp}$. When the left operand of $\sqcup_{\mathbb{R}^\sharp}$ contains two identity relations, the rule $(\sqcup_{\mathtt{Id}-\text{merge}})$ merges them (in this rule, $r_0^\sharp$ is needed to handle the cases where more than two identity relations need to be merged). When the left operand contains two transform-into relations, the rule $(\sqcup_{\dashrightarrow-\text{weak}})$ weakens them into one transform-into relation. Finally, when the left operand contains an identity relation and a transform-into relation, the rule $(\sqcup_{\dashrightarrow-\text{intro}})$ weakens the identity relation into a transform-into relation and applies recursively $\sqcup_{\mathbb{R}^\sharp}$.

Join in the numerical abstract domain. Finally, the join operation proceeds to the join in the numerical abstract domain. Like for the inclusion checking, the numerical abstract values have to take into account the renaming performed by the abstract heap relations. Thus, the join in the numerical abstract domain is performed by the function $\mathbf{join}_{\mathbb{N}^\sharp} : [\mathbb{V}^\sharp \to \mathbb{V}^\sharp]^2 \times \mathbb{N}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$.

Definition 31 (Join in abstract memory relations)
Let $m_{\mathscr{R},0}^\sharp = (e_0^\sharp, r_0^\sharp, n_0^\sharp)$ and $m_{\mathscr{R},1}^\sharp = (e_1^\sharp, r_1^\sharp, n_1^\sharp)$ be two abstract memory relations.

$$\forall \mathtt{x} \in \mathbb{X}, \text{Let } \Phi_{\text{init}}(\alpha) = (e_0^\sharp(\mathtt{x}), e_1^\sharp(\mathtt{x})) \text{ and } e^\sharp(\mathtt{x}) = \alpha.$$

$$\text{If } \mathbf{join}_{\mathbb{R}^\sharp}(\Phi_{\text{init}}, r_0^\sharp, r_1^\sharp) = (\Phi', r^\sharp) \text{ and } \mathbf{join}_{\mathbb{N}^\sharp}(\Phi', n_0^\sharp, n_1^\sharp) = n^\sharp \text{ then:}$$

$$\mathbf{join}_{\mathbb{M}_{\mathscr{R}}^\sharp}(m_{\mathscr{R},0}^\sharp, m_{\mathscr{R},1}^\sharp) = (e^\sharp, r^\sharp, n^\sharp)$$

Similarly as inclusion checking, $\mathbf{join}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ firsts initializes the pair of renaming functions Φ_{init} and creates the joined environment $\mathrm{e}^\sharp$. It then proceeds to the join of the abstract heap relations and to the join with Φ_{init}. Finally, it joins the abstract numerical values with the final pair of renaming functions returned by $\mathbf{join}_{\mathbb{R}^\sharp}$.

Condition 12 (Soundness of $\mathbf{join}_{\mathbb{N}^\sharp}$) *Let* $\Psi_0, \Psi_1 : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{n}^\sharp_0, \mathrm{n}^\sharp_1 \in \mathbb{N}^\sharp$, *then:*

$$(\Psi_0 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp_0) \vee (\Psi_1 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp_1) \quad \Longrightarrow \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{join}_{\mathbb{N}^\sharp}((\Psi_0, \Psi_1), \mathrm{n}^\sharp_0, \mathrm{n}^\sharp_1))$$

Condition 13 (Soundness of $\mathbf{join}_{\mathbb{H}^\sharp}$) *Let* $\Psi_0, \Psi_1 : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1 \in \mathbb{H}^\sharp$.
If $\mathbf{join}_{\mathbb{H}^\sharp}((\Psi_0, \Psi_1), \mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1) = ((\Psi'_0, \Psi'_1), \mathrm{h}^\sharp)$, *then:*

$$\begin{array}{lcl} \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_0(\alpha) = \beta & \Longrightarrow & \Psi'_0(\alpha) = \beta \\ \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_1(\alpha) = \beta & \Longrightarrow & \Psi'_1(\alpha) = \beta \\ (\mathrm{h}, \Psi'_0 \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_0) \vee (\mathrm{h}, \Psi'_1 \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_1) & \Longrightarrow & (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \end{array}$$

Theorem 21 (Soundness of $\mathbf{join}_{\mathbb{R}^\sharp}$) *Let* $\mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1 \in \mathbb{R}^\sharp$ *and* $\Psi_0, \Psi_1 : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$.
If $\mathbf{join}_{\mathbb{R}^\sharp}((\Psi_0, \Psi_1), \mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1) = ((\Psi'_0, \Psi'_1), \mathrm{r}^\sharp)$ *then:*

$$\begin{array}{lcl} \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_0(\alpha) = \beta & \Longrightarrow & \Psi'_0(\alpha) = \beta \\ \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_1(\alpha) = \beta & \Longrightarrow & \Psi'_1(\alpha) = \beta \\ (\mathrm{h_i}, \mathrm{h_o}, \Psi'_0 \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_0) \vee (\mathrm{h_i}, \mathrm{h_o}, \Psi'_1 \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_1) & \Longrightarrow & (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \end{array}$$

Theorem 22 (Soundness of $\mathbf{join}_{\mathbb{M}^\sharp_{\mathscr{R}}}$) *Let* $\mathrm{m}^\sharp_{\mathscr{R},0}, \mathrm{m}^\sharp_{\mathscr{R},1} \in \mathbb{M}^\sharp_{\mathscr{R}}$.
If $\mathbf{join}_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},0}, \mathrm{m}^\sharp_{\mathscr{R},1}) = \mathrm{m}^\sharp_{\mathscr{R}}$ *then:*

$$\gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},0}) \cup \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},1}) \subseteq \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}})$$

Condition 14 (Soundness of $\mathbf{join}_{\mathbb{T}^\sharp}$) *Let* $\Psi_0, \Psi_1 : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{t}^\sharp_0, \mathrm{t}^\sharp_1 \in \mathbb{T}^\sharp$.
If $\mathbf{join}_{\mathbb{T}^\sharp}((\Psi_0, \Psi_1), \mathrm{t}^\sharp_0, \mathrm{t}^\sharp_1) = \mathrm{t}^\sharp$, *then:*

$$(\mathrm{h_i}, \mathrm{h_o}, \Psi_0 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_0) \vee (\mathrm{h_i}, \mathrm{h_o}, \Psi_1 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_1) \quad \Longrightarrow \quad (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)$$

Definition 32 (Join in abstract heap transformation predicates domains) We define the join function $\mathbf{join}_{\mathbb{T}^\sharp} : [\mathbb{V}^\sharp \to \mathbb{V}^\sharp]^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ for each abstract heap transformation predicates domain defined in Section 4.3.

1. The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$: the definition of the operation $\mathbf{join}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}^\sharp_0, \mathrm{t}^\sharp_1) = \mathrm{t}^\sharp$ is given by the following tabular (each line for $\mathrm{t}^\sharp_0$ and each column for $\mathrm{t}^\sharp_1$).

$\mathrm{t}^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$=^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$\subseteq^\sharp$	$\subseteq^\sharp$	$\subseteq^\sharp$	$\top$	$\top$
$\supseteq^\sharp$	$\supseteq^\sharp$	$\top$	$\supseteq^\sharp$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

2. The fields predicates domain, $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$:

$$\mathbf{join}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}^\sharp_0, \mathrm{t}^\sharp_1) = \mathrm{t}^\sharp_0 \cup \mathrm{t}^\sharp_1$$

3. The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}^\sharp_a \times \mathbb{T}^\sharp_b$:

$$\mathbf{join}_{\mathbb{T}^\sharp}(\Phi,(t^\sharp_{a,0},t^\sharp_{b,0}),(t^\sharp_{a,1},t^\sharp_{b,1})) = (\mathbf{join}_{\mathbb{T}^\sharp_a}(\Phi,t^\sharp_{a,0},t^\sharp_{a,1}),\mathbf{join}_{\mathbb{T}^\sharp_b}(\Phi,t^\sharp_{b,0},t^\sharp_{b,1}))$$

Theorem 23 (Soundness of Definition 32) *The operators from Definition 32 are sound in the sense of Condition 14.*

5.7.2 Widening operator.

During the analysis of loops and recursive programs, the number of iterations of the static analysis has to be *finite*. To always terminate in a finite number of steps, the analysis requires a widening operator $\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ that joins abstract memory relations and provides a convergence acceleration for the iteration process.

The widening operator $\mathbf{wid}_{\mathbb{R}^\sharp}$ for abstract heap relations can be implemented using the same rules system of Figure 12. Indeed, each rule strictly decreases the number of abstract heap relations, which ensures termination in a finite number of steps. Moreover, $\sqcup_{\mathbb{H}^\sharp}$ is already a widening operator, as it converges in a finite number of steps, as explained in [10].

However, $\mathbf{join}_{\mathbb{N}^\sharp}$ cannot be used as a widening operator. Indeed, some numerical abstract domains such as intervals [16] or convex polyhedra [15] require a specific widening operation. Thus, the abstract numerical domain should implement its own widening operator $\mathbf{wid}_{\mathbb{N}^\sharp}$.

Condition 15 (Soundness of $\mathbf{wid}_{\mathbb{N}^\sharp}$) *Let* $\Psi_0,\Psi_1 : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $n^\sharp_0, n^\sharp_1 \in \mathbb{N}^\sharp$*, then:*

$$(\Psi_0 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(n^\sharp_0) \vee (\Psi_1 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(n^\sharp_1) \quad \Longrightarrow \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{wid}_{\mathbb{N}^\sharp}((\Psi_0,\Psi_1),n^\sharp_0,n^\sharp_1))$$

The function $\mathbf{wid}_{\mathbb{N}^\sharp}$ *also enforces termination.*

The operator $\sqcup_{\mathbb{T}^\sharp}$ cannot be used as a widening operator: it may not converge in a finite number of steps. Indeed, the used abstract heap transformation predicates domain $\mathbb{T}^\sharp$ may denote an infinite set. The solution is to define a converging widening operator $\triangledown_{\mathbb{T}^\sharp}$ for abstract heap transformation predicates. This operator is then implemented by a function $\mathbf{wid}_{\mathbb{T}^\sharp} : [\mathbb{V}^\sharp \to \mathbb{V}^\sharp]^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ that is assumed to ensure termination. Its soundness property is the same as the function $\mathbf{join}_{\mathbb{T}^\sharp}$, except that it also enforces termination.

Condition 16 (Soundness of $\mathbf{wid}_{\mathbb{T}^\sharp}$) *Let* $\Psi_0,\Psi_1 : \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $t^\sharp_0, t^\sharp_1 \in \mathbb{T}^\sharp$.
If $\mathbf{wid}_{\mathbb{T}^\sharp}((\Psi_0,\Psi_1),t^\sharp_0,t^\sharp_1) = t^\sharp$*, then:*

$$(h_i,h_o,\Psi_0 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_0) \vee (h_i,h_o,\Psi_1 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp_1) \quad \Longrightarrow \quad (h_i,h_o,\nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp)$$

The function $\mathbf{wid}_{\mathbb{T}^\sharp}$ *also enforces termination.*

Definition 33 (Widening for abstract heap transformation predicates domains) We define the widening function $\mathbf{wid}_{\mathbb{T}^\sharp} : [\mathbb{V}^\sharp \to \mathbb{V}^\sharp]^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ for each abstract heap transformation predicates domain defined respectively in Section 4.3.1, Section 4.3.2 and Section 4.3.3.

1. The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$:

$$\mathbf{wid}_{\mathbb{T}^\sharp}(\Phi,t^\sharp_0,t^\sharp_1) = \mathbf{join}_{\mathbb{T}^\sharp}(\Phi,t^\sharp_0,t^\sharp_1)$$

2. The fields predicates domain, $\mathbb{T}^\sharp = \mathscr{P}(\mathbb{F})$:

$$\mathbf{wid}_{\mathbb{T}^\sharp}(\Phi,t^\sharp_0,t^\sharp_1) = \mathbf{join}_{\mathbb{T}^\sharp}(\Phi,t^\sharp_0,t^\sharp_1)$$

3. The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}^\sharp_a \times \mathbb{T}^\sharp_b$:

$$\mathbf{wid}_{\mathbb{T}^\sharp}(\Phi, (\mathrm{t}^\sharp_{a,0}, \mathrm{t}^\sharp_{b,0}), (\mathrm{t}^\sharp_{a,1}, \mathrm{t}^\sharp_{b,1})) = (\mathbf{wid}_{\mathbb{T}^\sharp_a}(\Phi, \mathrm{t}^\sharp_{a,0}, \mathrm{t}^\sharp_{a,1}), \mathbf{wid}_{\mathbb{T}^\sharp_b}(\Phi, \mathrm{t}^\sharp_{b,0}, \mathrm{t}^\sharp_{b,1}))$$

Theorem 24 (Soundness of Definition 33) *The operators from Definition 33 are sound in the sense of Condition 16.*

Regarding to the footprint and the fields predicates domains, the function $\mathbf{wid}_{\mathbb{T}^\sharp}$ is defined similary as the function $\mathbf{join}_{\mathbb{T}^\sharp}$. These definitions are valid because the footprint and the fields predicate domains are both finite. On the other hand, the combined predicates domain applies recursively the widening of its two sub-predicates domains.

Finally, the widening of abstract memory relation $\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ can be defined like $\mathbf{join}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ by substituting in its definition $\mathbf{join}_{\mathbb{N}^\sharp}$ by $\mathbf{wid}_{\mathbb{R}^\sharp}$ and by substituting $\mathbf{join}_{\mathbb{R}^\sharp}$ by $\mathbf{wid}_{\mathbb{N}^\sharp}$.

Definition 34 (Widening for abstract memory relations)
Let $\mathrm{m}^\sharp_{\mathscr{R},0} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$ and $\mathrm{m}^\sharp_{\mathscr{R},1} = (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1)$ be two abstract memory relations.

$$\forall \mathtt{x} \in \mathbb{X}, \text{Let } \Phi_{\text{init}}(\alpha) = (\mathrm{e}^\sharp_0(\mathtt{x}), \mathrm{e}^\sharp_1(\mathtt{x})) \text{ and } \mathrm{e}^\sharp(\mathtt{x}) = \alpha.$$

$$\text{If } \mathbf{wid}_{\mathbb{R}^\sharp}(\Phi_{\text{init}}, \mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1) = (\Phi', \mathrm{r}^\sharp) \text{ and } \mathbf{wid}_{\mathbb{N}^\sharp}(\Phi', \mathrm{n}^\sharp_0, \mathrm{n}^\sharp_1) = \mathrm{n}^\sharp \text{ then:}$$
$$\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},0}, \mathrm{m}^\sharp_{\mathscr{R},1}) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$$

Theorem 25 (Soundness of $\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}$) *Let* $\mathrm{m}^\sharp_{\mathscr{R},0}, \mathrm{m}^\sharp_{\mathscr{R},1} \in \mathbb{M}^\sharp_{\mathscr{R}}$.
If $\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},0}, \mathrm{m}^\sharp_{\mathscr{R},1}) = \mathrm{m}^\sharp_{\mathscr{R}}$ *then:*

$$\gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},0}) \cup \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R},1}) \subseteq \gamma_{\mathbb{M}^\sharp_{\mathscr{R}}}(\mathrm{m}^\sharp_{\mathscr{R}})$$

The function $\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}$ *also enforces termination*

Example 19 (Widening) We consider the analysis of the program of Figure 1, and more specifically, the widening after the first abstract iteration over the loop. We assume that $\mathbf{wid}_{\mathbb{M}^\sharp_{\mathscr{R}}}((\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0), ((\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1))) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ where $\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{e}^\sharp_1$ and $\mathrm{r}^\sharp_1$ are detailed below:

$$\begin{aligned}
&\mathrm{e}^\sharp_0 = [\mathtt{l} \mapsto \alpha_0; \mathtt{v} \mapsto \beta_0; \mathtt{c} \mapsto \delta_0] \\
&\mathrm{r}^\sharp_0 = \mathtt{Id}(\alpha_0 \mapsto \alpha'_0 *_{\mathtt{S}} \mathbf{list}(\alpha'_0) *_{\mathtt{S}} \beta_0 \mapsto \beta'_0) *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\delta_0 \mapsto \alpha'_0)]_{\mathrm{t}^\sharp_0} \\
&\qquad \text{with } \mathrm{t}^\sharp_0 = (\subseteq^\sharp, \{\})
\end{aligned}$$

$$\begin{aligned}
&\mathrm{e}^\sharp_1 = [\mathtt{l} \mapsto \alpha_1; \mathtt{v} \mapsto \beta_1; \mathtt{c} \mapsto \delta_1] \\
&\mathrm{r}^\sharp_1 = \mathtt{Id}(\alpha_1 \mapsto \alpha'_1 *_{\mathtt{S}} \alpha'_1 \cdot \mathtt{data} \mapsto \delta'_1 *_{\mathtt{S}} \alpha'_1 \cdot \mathtt{next} \mapsto \alpha''_1 *_{\mathtt{S}} \mathbf{list}(\alpha''_1) *_{\mathtt{S}} \beta_1 \mapsto \beta'_1) \\
&\qquad *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\delta_1 \mapsto \alpha''_1)]_{\mathrm{t}^\sharp_1} \text{ with } \mathrm{t}^\sharp_1 = (\subseteq^\sharp, \{\mathtt{0}\}) \quad (\mathtt{0} \text{ represents the null offset})
\end{aligned}$$

First, the initialization produces the initial pair of renaming functions Φ_{init} such as $\Phi_{\text{init}}(\alpha) = (\alpha_0, \alpha_1)$, $\Phi_{\text{init}}(\beta) = (\beta_0, \beta_1)$ and $\Phi_{\text{init}}(\delta) = (\delta_0, \delta_1)$. Thus, the resulting abstract environment is $\mathrm{e}^\sharp = [\mathtt{l} \mapsto \alpha; \mathtt{v} \mapsto \beta; \mathtt{c} \mapsto \delta]$.

Then, $\mathbf{wid}_{\mathbb{R}^\sharp}(\Phi_{\text{init}}, \mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1)$ is applied and produces:

$$\begin{aligned}
\mathrm{r}^\sharp = {} & \mathtt{Id}(\alpha \mapsto \alpha' *_{\mathtt{S}} \mathbf{listseg}(\alpha', \alpha'') *_{\mathtt{S}} \mathbf{list}(\alpha'') *_{\mathtt{S}} \beta \mapsto \beta') \\
& *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\delta \mapsto \alpha'')]_{\mathrm{t}^\sharp} \text{ with } \mathrm{t}^\sharp = (\subseteq^\sharp, \{\mathtt{0}\})
\end{aligned}$$

$$\begin{array}{rll}
\mathbf{assign}_{\mathbb{D}^\sharp} : & \mathtt{L} \times \mathtt{E} \times \mathbb{D}^\sharp & \to \mathbb{D}^\sharp \\
\mathbf{alloc}_{\mathbb{D}^\sharp} : & \mathtt{L} \times \mathscr{P}_{\mathrm{fin}}(\mathbb{F}) \times \mathbb{D}^\sharp & \to \mathbb{D}^\sharp \\
\mathbf{free}_{\mathbb{D}^\sharp} : & \mathtt{L} \times \mathbb{D}^\sharp & \to \mathbb{D}^\sharp \\
\mathbf{guard}_{\mathbb{D}^\sharp} : & \mathtt{E} \times \mathbb{D}^\sharp & \to \mathbb{D}^\sharp \\
\mathbf{isle}_{\mathbb{D}^\sharp} : & \mathbb{D}^\sharp \times \mathbb{D}^\sharp & \to \{\mathbf{true}, \mathbf{false}\} \\
\mathbf{join}_{\mathbb{D}^\sharp} : & \mathbb{D}^\sharp \times \mathbb{D}^\sharp & \to \mathbb{D}^\sharp \\
\mathbf{wid}_{\mathbb{D}^\sharp} : & \mathbb{D}^\sharp \times \mathbb{D}^\sharp & \to \mathbb{D}^\sharp
\end{array}$$

Fig. 13: Disjunction Abstract Domain Interface ($\mathbb{D}^\sharp = \mathscr{P}_{\mathrm{fin}}(\mathbb{M}^\sharp_{\mathscr{R}})$).

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 extends Φ_{init} into Φ with $\Phi(\alpha') = (\alpha'_0, \alpha'_1), \Phi(\beta') = (\beta'_0, \beta'_1)$ and $\Phi(\alpha'') = (\alpha'_0, \alpha''_1)$.

In turn, $\mathbf{wid}_{\mathbb{N}^\sharp}$ can be applied with $\Phi, \mathrm{n}^\sharp_0$ and $\mathrm{n}^\sharp_1$.

5.8 Analysis

5.8.1 Manipulating Disjunctions in the Analysis

Because of unfolding operations, the analysis should deal with disjunctions. This is exactly the role of the disjunction abstract domain, defined in Section 4.4.2. It is build on top of abstract memory relations domain and permit to perform standard abstract operations (assignment, allocation, ...) over finite sets of abstract memory relations. Its interface is given in Figure 13. Each function of this interface applies the function of the same name, but at the abstract memory relation level. In doing so, it carries disjunctions and may create additional disjuncts when unfolding is performed. Although we do not formalize this operator here, an implementation of this abstract domain may rely on a function $\mathbf{collapse}_{\mathbb{D}^\sharp} : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ that transforms a disjunctive abstraction into another disjunctive abstraction with fewer disjuncts, which helps keeping the analysis cost down. The soundness of the functions of this interface and more details are discussed in [11].

5.8.2 Abstract Relational Semantics

The abstract semantics $[\![.]\!]^\sharp_{\mathscr{R}}$ relies on the abstract operations defined in Sections 5.3, 5.4 and 5.5, on the unfolding of Section 5.2 to analyze basic statements, and on the folding operations defined in Sections 5.6 and 5.7 to cope with control flow joins and loop invariants computation. It is defined by induction over the syntax of the programming language defined in Section 3.3 and operates over abstract disjunctions, as shown in Figure 14.

Soundness of $[\![.]\!]^\sharp_{\mathscr{R}}$ follows from the soundness of the basic operations.

Theorem 26 (Soundness) *The analysis is sound in the sense that, for all program p and for all abstract disjunction* $\mathrm{d}^\sharp$*:*

$$\begin{array}{l}
\forall(\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{D}^\sharp}(\mathrm{d}^\sharp),\ \forall \mathrm{m}_2 \in \mathbb{M}, \\
\qquad (\mathrm{m}_1, \mathrm{m}_2) \in [\![p]\!]_{\mathscr{R}} \quad \Longrightarrow \quad (\mathrm{m}_0, \mathrm{m}_2) \in \gamma_{\mathbb{D}^\sharp}([\![p]\!]^\sharp_{\mathscr{R}}(\mathrm{d}^\sharp))
\end{array}$$

$$[\![loc = exp]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}) = \mathbf{assign}_{\mathbb{D}^{\sharp}}(loc, exp, \mathrm{d}^{\sharp})$$

$$[\![loc = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\})]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}) = \mathbf{alloc}_{\mathbb{D}^{\sharp}}(loc, \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}, \mathrm{d}^{\sharp})$$

$$[\![\mathbf{free}(loc)]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}) = \mathbf{free}_{\mathbb{D}^{\sharp}}(loc, \mathrm{d}^{\sharp})$$

$$[\![p_1; p_2]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}) = [\![p_2]\!]^{\sharp}_{\mathscr{R}}([\![p_1]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}))$$

$$\begin{aligned}[\![\mathbf{if}\,(exp)\,p_1\,\mathbf{else}\,p_2]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}) = \mathbf{join}_{\mathbb{D}^{\sharp}}(&[\![p_1]\!]^{\sharp}_{\mathscr{R}}(\mathbf{guard}_{\mathbb{D}^{\sharp}}(exp, \mathrm{d}^{\sharp})),\\ &[\![p_2]\!]^{\sharp}_{\mathscr{R}}(\mathbf{guard}_{\mathbb{D}^{\sharp}}(\neg exp, \mathrm{d}^{\sharp})))\end{aligned}$$

$$\begin{aligned}[\![\mathbf{while}(exp)\,p]\!]^{\sharp}_{\mathscr{R}}(\mathrm{d}^{\sharp}_0) = \;&\text{Let } \mathrm{d}^{\sharp}_1 = [\![p]\!]^{\sharp}_{\mathscr{R}}(\mathbf{guard}_{\mathbb{D}^{\sharp}}(exp, \mathrm{d}^{\sharp}_0)) \text{ in}\\ &\text{If } \mathbf{isle}_{\mathbb{D}^{\sharp}}(\mathrm{d}^{\sharp}_1, \mathrm{d}^{\sharp}_0) = \mathbf{true}\\ &\text{Then } \mathbf{guard}_{\mathbb{D}^{\sharp}}(\neg exp,\, \mathbf{join}_{\mathbb{D}^{\sharp}}(\mathrm{d}^{\sharp}_0, \mathrm{d}^{\sharp}_1))\\ &\text{Else } [\![\mathbf{while}(exp)\,p]\!]^{\sharp}_{\mathscr{R}}(\mathbf{wid}_{\mathbb{D}^{\sharp}}(\mathbf{join}_{\mathbb{D}^{\sharp}}(\mathrm{d}^{\sharp}_0, \mathrm{d}^{\sharp}_1),\, \mathrm{d}^{\sharp}_0))\end{aligned}$$

Fig. 14: Abstract semantics for the programing language defined in Section 3.3. The expression $\neg exp$ is the negation of exp.

6 Experimental Evaluation

In this section, we report on the implementation and on the evaluation of our relational shape abstraction. The purpose of this evaluation is to assess whether it is able to represent and compute strong relational properties, how it compares with standard state shape analyses, and whether it is adapted to verify programs. More precisely, we intend to evaluate how the relational predicates (both the relational connectors and the transformation predicates) behave in practice. To reach these goals, our evaluation is split into two parts:

1. in the first part, we consider basic data-structure libraries and check that the relational analysis infers interesting input-output relations and we compare it with more conventional state analyses (such test cases are often used to assess state analyses thus results of such analyses are known quantity in this part of the evaluation);
2. in the second part, we apply the relational analysis to the verification of the list module of the Contiki operating system and compare our results with those obtained with an approach based on deductive verification [4].

Note that our evaluation focuses on the relational shape abstract domain, thus we do not include test cases relying on sophisticated data invariants or that would involve many static analysis design choices unrelated to the abstraction of shapes and relations. As an example, the analysis of a large interprocedural code using relations as procedure summaries would necessarily evaluate the way procedures are handled (bottom-up or top-down, etc) as much as, if not more than, the shape abstraction itself, thus it would not provide an ideal target for our study.

6.1 On Basics Library of Lists and Trees

In a first time, we report on the implementation of our analysis and try to evaluate:

1. whether it can infer precise and useful relational properties,
2. how abstract heap transformation predicates domains can improve its precision, and
3. how it compares with a state shape analysis that does not use relations.

Struct.	Function	Time (in s)			Logical Strength	
		State	Rel.	Rel.+	State vs Rel.	Rel. vs Rel.+
sll	`allocation`	0.56	0.77	0.78	<	=
sll	`deallocation`	0.46	0.80	0.79	<	=
sll	`traversal`	0.58	0.79	0.77	<	=
sll	`head_insertion`	0.43	0.43	0.44	<	=
sll	`insert` (Figure 1)	1.11	1.92	1.93	<	=
sll	`reverse`	0.60	1.01	1.06	=	<
sll	`map`	0.59	0.92	0.91	=	<
sll	`tail`	0.42	0.55	0.54	<	=
sll	`nth`	0.70	1.17	1.15	<	=
sll	`partition`	2.18	4.85	4.93	=	<
sll	`appends`	0.88	1.60	1.59	<	=
sll	`contains`	0.82	1.22	1.24	<	=
sll	`deep_copy`	1.15	2.08	2.16	<	=
sll	`sort` (Figure 3)	4.09	21.95	22.16	=	<
sll	`filter`	1.21	2.70	2.79	=	<
bst	`allocation`	0.71	1.11	1.45	<	=
bst	`search`	0.97	1.63	1.67	<	=
bst	`insert`	2.25	6.10	6.22	<	=

Table 3: Experiment results (sll: singly linked lists; bst: binary search trees; time in seconds averaged over 1000 runs on a laptop with Intel Core i5 running at 2.4 GHz, with 4 Gb RAM, for the state, basic and extended relational analyses; the last column compares the logical strength of the inferred result of each 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 state shape analysis in the style of [10], and supporting the same inductive predicates. It was implemented as a Frama-C [32] plug-in consisting of roughly 15000 lines of OCaml.

We have ran both the state shape analysis and the relational shape analysis (in a first time without abstract heap transformation predicates, and a second with) on series of small programs manipulating lists and trees listed in Table 3. This allows us to not only assess the results of the analysis computing abstract state relations, but also to compare them with an analysis that infers abstract states. Each program requires a short (1-line) precondition stating that the argument of the function are well-formed linked-list and trees.

The results obtained are listed in Table 3. The column 'State' corresponds to the execution time of the state analysis. The column 'Rel.', the *basic relational analysis*, corresponds to the relational analysis *without* abstract heap transformation predicates. The column 'Rel.+', the *extended relational analysis*, corresponds to the abstract heap relation analysis *with* the abstract heap transformation predicates. As a transformation predicate, we use the combined predicates domain of the fields and the footprint predicates domains, defined in Section 4.3. The analysis runtimes are averaged over 1000 runs of the analysis. The last two columns compare the logical strength of the results of the analyses. This is indicated in the cells by comparison symbols ($<$, $>$ or $=$). For instance, if we find '$<$' in a cell of the State vs Rel. column, that means that the basic relational analysis inferred a stronger property than the state analysis for the given function.

6.1.1 Logical Strength Comparison

We first discuss on the logical strength. We observe that both the basic and the extended relational analyses never infer weaker properties than the state analysis (for instance, there is no '$>$' in the State vs Rel. column). In most cases, the basic relational analysis is sufficient to infer stronger properties than the state analysis. In these cases, the extended relational analysis often does not infer more information. The main reason is that the inferred relation already describes a very precise relation. Otherwise, when the basic relational analysis did not infer a stronger property than the state analysis, the extended relational analysis infers in all the cases a stronger relational property.

The most important observation is that for all the cases, we have that the basic relational analysis or the extended relational analysis inferred a strictly stronger property than the state analysis. We discuss some of cases in the next paragraphs.

When the result of the basic relational analysis is stronger than the state analysis. In this case, we consider the functions `head_insertion` and `deep_copy`:

- `head_insertion`: the basic relational analysis inferred the following abstract heap relation:
$$[\alpha_0 \mapsto \alpha_1 \dashrightarrow \alpha_0 \mapsto \beta] *_{\mathrm{R}} [\mathbf{emp} \dashrightarrow \beta \cdot \mathtt{next} \mapsto \alpha_1]$$
$$*_{\mathrm{R}} [\mathbf{emp} \dashrightarrow \beta \cdot \mathtt{data} \mapsto \delta] *_{\mathrm{R}} \mathtt{Id}(\mathbf{list}(\alpha_1))$$
It describes exactly the effects of the insertion of a new allocated element at the head of a given list. Indeed, it expresses explicitly that the input list (abstracted by the inductive predicate $\mathbf{list}(\alpha_1)$) has not been modified by the function. Moreover, it expresses the allocation of a new list element β that `next` field points to the input list. This abstract relation is clearly more expressive than the result of the state analysis, that cannot capture the relational properties described above. However, adding abstract heap transformation predicates to the relational analysis did not add more interesting relational properties: in this case, the basic relational analysis is already precise enough.
- `deep_copy`: this function traverses a list and copies its `data` fields in a new list. The inferred basic abstract heap relation by the basic relational analysis is:
$$\mathtt{Id}(\mathbf{list}(\alpha)) *_{\mathrm{R}} [\mathbf{emp} \dashrightarrow \mathbf{list}(\beta)]$$
It indicates that the input list $\mathbf{list}(\alpha)$ has not been modified and that a new list $\mathbf{list}(\beta)$ has been freshly allocated. These properties cannot be inferred by the state analysis. Neverless, the abstract heap transformation predicate $(\subseteq^\sharp, \{\mathtt{data}, \mathtt{next}\})$ inferred by the extended relational analysis does not provide more information. Indeed, the predicate $\subseteq^\sharp$ is useless because we already knew that $\mathbf{list}(\beta)$ has been allocated (its abstract input heap is **emp**) and the predicate $\{\mathtt{data}, \mathtt{next}\}$ indicates that both the `data` and the `next` fields may have been modified.

When the result of the extended relational analysis is stronger than the basic relational analysis. The second case is when the result of the basic relational analysis is not stronger than the result of the state analysis but when the result of the extended relational analysis is stronger than the result of the basic relational analysis. We discuss the `map`, `reverse`, `filter` and `partition` functions that are in that situation:

- `map`: this function traverses a list and increments each `data` field. The inferred abstract heap relation for this function is $[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]$. It just indicates that the input and output lists start at the same address. We have no more information compared to the state

analysis. However, using abstract heap transformation predicates, we obtained the predicate $(=^\sharp, \{\texttt{data}\})$ for the previous transform-into relation. It indicates that the footprint of the two lists is the same and that only the `data` fields may have been modified. This describes much more accurately the behavior of this function.

- `reverse`: this function reverses (in place) the order of the elements of a list. The inferred abstract heap relation of this function is $[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$. It does not express any interesting relation compared to the state analysis (only the transformation of a list into another). However, the inferred abstract heap transformation predicate $(=^\sharp, \{\texttt{next}\})$ expresses a permutation in place of the list, without modifying the `data` fields. This is the same abstraction obtained for the sort function of Figure 3.
  We remark that this is not the most precise property that we may think of (it does not imply that the elements of the list were properly reversed; it just says they were re-ordered). This is due to the fact that the transformation predicates that we have used cannot capture this most precise property, and can at most state that the footprint and values were preserved. Of course, one may design a specific transformation predicate to account for the reversal operation.
- `filter`: this function deallocates all the negative elements of the input list. The inferred abstract heap relation is the same as the function reverse but the inferred abstract heap transformation predicates is $(\supseteq^\sharp, \{\texttt{next}\})$. This means that some deallocations may occur and no `data` field has been modified.
- `partition`: this function partitions (in place) the input list into two lists. The first one contains all the positive elements of the input list and the second all the negative elements. The inferred abstract heap relation of this function $[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta_1) *_{\mathrm{S}} \mathbf{list}(\beta_2)]$ only indicates the presence of two well formed linked lists in the output state, but nothing more than the state analysis does. However, the inferred abstract heap transformation predicate $(=^\sharp, \{\texttt{next}\})$ indicates that these two output lists are composed by the elements of the input list, and that the `data` fields of the latter have not been modified.

6.1.2 Runtime Comparison

We now compare the runtime of the relational analysis and of the state analysis. We observe in most cases that the relational analysis is slower than the state analysis, although the slow down factor is reasonable. Indeed, the time of relational analysis rarely exceeds the double of the state analysis (this is the case for the functions `partition`, `filter` and tree `insert`). An exception is the list `sort`, which is approximately 5 times slower. This is explained by the fact that this function contains a condition test in a nested loop and another condition test in the main loop. This implies to perform an important number of abstract joins and widenings. Conversely, the function `head_insertion`, that does not perform either abstract join or widening, avoids any slowdown. Moreover, we believe that we can optimize the implementation of these operators in our prototype analyzer, using a better strategy to detect the rules to apply. To do that, we could draw inspiration from the work in [35] that proposes an elegant solution to solve this problem.

While these test cases are not large, these results show that the relational analyses have a reasonable overhead and that they bring additional information compared to a classical state analysis. The difference time between the analyses is due to the fact that the relational analysis manipulates pairs of states whereas the state analysis manipulates only one state. In general, the relational analysis infers stronger properties.

In order to be able to analyze large programs, we consider for future works to perform a *compositional analysis* where abstract relations are used as *function summaries* and composed

at call sites. This avoids to reanalyze a function every time it is called, which is an advantage for scalability.

6.2 On The List Module of The Operating System Contiki

In this section, we run experiments to evaluate the ability of our relational analysis to infer and verify desired function contracts about procedures manipulating linked data structures. We also check that our analysis is sound, i.e. raises an alarm when a program contains a bug. Finally, we evaluate the effort one can gain by using a fully-automated analysis like ours.

For this, we compare our approach with the one of Blanchard et al. [4] for the verification of the linked list module of the operating system Contiki [26]. Their work performs a deductive verification of this list module and is based on a parallel view of a linked list via a companion ghost array. In particular, the modifications of the lists performed by the function are described as relations between ghost arrays in the input and output states.

This approach requires the user to specify both the pre and post conditions of each function, to annotate the source code with many invariants (mostly loop invariants), to write ghost functions and to prove different lemmas using SMT solvers or the Coq proof assistant. In total, for about 176 lines of C code in the list module, they wrote 46 lines of for ghost functions and about 1400 lines of annotations. Their verification has generated 798 goals to prove. Among these goals, 770 have been proven automatically by SMT solvers, 4 interactively and 24 proven using the Coq proof assistant. By comparison we only need to write one or two lines of annotations for preconditions, which only state that the input of the function is a well-formed linked list, and, when there is another argument, whether this argument is separated or belongs to that list.

Since the contracts of these functions were written manually in [4], they provide an accurate description of the properties that one wants to prove about these functions. For instance, the contract for `list_length` states that the function inputs a list and returns its length but does not modify the list, which is the expected contract for a function with this name.

We aimed to check if our relational abstract domain was able to express and verify the same properties than those stated in these contracts, but *more automatically*. More precisely, we assessed whether our approach could verify the exact shape properties (e.g., a list head insertion function preserves the existing elements of the list and adds an element at the head), under the data of the abstract pre-condition. Due to the radical approach difference, we need to use a different formalism than theirs (while they use ghost arrays, we rely on relational separation logic). Therefore, the properties that we check are *equivalent* to theirs, in the sense that the input/output states described by their contracts correspond to those infered by our analysis. Note that as we do not need to reason over ghost arrays, the contracts cannot be strictly the same.

We have analyzed all the functions given in their paper, using the source code that they provide. These functions are listed in Table 4. It shows that for all of the functions (except for `list_length`), the contracts inferred by our analysis are equivalent to those given by Blanchard et al. For `list_length` we do infer the relational shape property that the list is not modified, but we cannot prove that the returned integer corresponds to the length of the list. We did not analyze the files that only manipulate arrays such as `array_pop.c`, as our relational abstract domain does not require ghost array companions.

An important feature in this list module is that each element of a list has to be unique. So if a function adds an element into a list, and if this element is already in this list, the element first has to be removed from the list and then added at the desired position. This is why all the functions

Function	Equivalent contract	Num. of pre-conditions	Times (in ms)
`list_add`	yes	2	211 + 207
`list_chop`	yes	1	204
`list_copy`	yes	1	207
`list_head`	yes	1	201
`list_init`	yes	1	203
`list_insert`	yes	3	208 + 203 + 204
`list_item_next`	yes	1	202
`list_length`	relational shape only	1	203
`list_pop`	yes	1	200
`list_push`	yes	2	208 + 204
`list_remove`	yes	2	201 + 202
`list_tail`	yes	1	201

Table 4: Experimental results for the linked list module of the operating system Contiki. It indicates if our relational analysis infered a function contract equivalent to the one manually specified by Blanchard et al.'s. The third column indicates with how many different pre-conditions we run the analysis of the function. The last column indicates the execution times (in ms) for each pre-condition of the function

that add an element into a list (`list_push`, `list_add`) call the function `list_remove`. This latter removes the given list element from the input list if it is inside, or leaves the list unchanged otherwise. For these functions we run our analysis twice with these 2 different preconditions: when the input list contains the given element and when it does not contain it. We could have to run the analysis once with only one pre-condition consisting of a disjunction of these to previous pre-conditions, but these disjuncts may have been joined during a widening. This would have begotten a too high loss of precision.

The function `list_insert` inserts a given list element into a list at a given position. We analyzed this function with three different pre-conditions: when the element to insert is not in the list, when the element is in the list but before the position it is supposed to be inserted, and when the element is in the list but after the desired position for insertion. This function actually contains a bug. Indeed, if the element is already in the list (no matter before or after the position of insertion), the function adds directly the element at the given position without removing it from the list. This breaks the structure of the list. Like Blanchard et al., we found this bug with our analysis.

7 Related Works

7.1 Extension of Separation Logic

Separation logic is used in program verification systems that can be separated in three kinds:

- Verification tools based on interactive proof assistants (e.g. [3]) requires the tool to apply or check the use of the separation logic inference rules [39].
- Semi-automated tools based on deductive verification (e.g. [36, 29]) avoids the need to write many intermediate proof steps, by additionally providing automated entailment checking procedure.
- Fully-automated tools based on abstract interpretation (e.g. [8, 10]) can automatically infer shape properties as separation logic formulas; for this the separation logic formulas must be manipulable as elements of an abstract domain (e.g. a join and widening operators must

be defined). A sound (but not complete) automated entailment checking is also provided as an inclusion operator of the abstract domain.

Our work takes place in the context of the latter, and extends it to infer not only shape invariants on the memory, but relational invariants (e.g. function contracts) between different program locations, using an extension of separation logic.

In the past, several works have enriched separation logic in order to perform specific analyses. To our knowledge, our work is the first to propose logical connectives based on separation logic that support inductive predicates and describe input-output heap relations.

Extensions expressing immutability Classical separation logic cannot directly express immutability, and the classical way to express such a property in a Hoare triple is to state that the values stored in memory in the pre- and post-condition are the same. Several authors have proposed extensions to separation logic to work around this limitation.

In the context of interactive proof, Charguéraud et al. [12] introduced the temporary read-only permission through a new connective for separation logic. This connective offers read-only access to any heap fragment described with a separation logic formula. David et al. [21] introduced a similar concept in the context of deductive verification.

Like our relational connectives, this read-only connective can express that some part of the heap is left unchanged, as it can only be read. Note however that immutability is only one example among many of the relations that we can express: for instance the footprint predicate domain of Section 4.3.1 allows expressing relational properties that are not immutability properties, such as whether a sorting algorithm is in-place.

Costea et al. [14] introduced read-only permission at the field level, which is similar in expressivity to our Fields predicate domain of Section 4.3.2.

Another important difference with these works is that our analysis is able to infer post-conditions for programs in the presence of loops; whereas the first work focuses on manual proofs, and the second and third on the verification (based on automatic entailment checking procedures) of user-supplied invariants.

Other extensions of separation logic An important extension of separation logic is the concurrent separation logic [37], which allows independent reasoning about threads that access separate storage. The new connective $\|$ allows to evaluate two terms of separation logic in parallel.

Like the read-only permission extensions, an important difference of these separation logic extensions compared to ours is that they modify directly the original separation logic; whereas in our extension, the relational connectives encompasses the terms of separation logic, without modifying them.

Fu et al. [27] introduced extension of separation logic to specify historical execution traces of heaps in the context of concurrent programs. They also have a new separating conjunction connective but for disjoint traces, whereas our relational separating conjunction expresses independent transformations. Desynchronized separation [20] 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.

7.2 Relational Analysis

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 [19], 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, [38] 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 [40] in [30]. 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 technique as mentioned above. Our analysis also has a notion of overlaid old / new predicates, but these are described heap regions, inside separation logic formulas.

Several other techniques have been used to specify memory properties. For instance, [42] uses temporal logic to specify temporal properties of heap evolutions and [41] checks structural properties of codes using a specification language.

In the context of concurrent programs, [1] verifies linearizability of concurrent objects (unbounded linked list) maintaining isomorphism between two instances of memory layout. Also, [23] uses an extension of temporal linear temporal logic and a tableau-based model-checking algorithm to specify the dynamic evolution of pointer structures. This latter technique has been applied in [24] to prove the correctness of concurrent programs manipulating linked lists.

In the context of functional languages, [31] allows to write down relations between function inputs and outputs, and relies on a solver to verify that constraints hold and [43] computes shape specification by learning.

Regarding to shape analyses based on separation logic, [5] infers combined list-data relations and has been extended in [6] for inter-procedural analysis. They can infer precise relations between the data contained in a list, such as the sum of all data in a list is inferior to the length of this list. They also use a multi-set to represent the data of a list. For example, to prove a function sort, they compare the multi-set of the input list with the multi-set of the output list. If these multi-sets are the same (this means a permutation), and if the output list is sorted, then the sort function is proven. However, they do not have this notion of physical equality between the different memory cells that our relational domain can express. Consequently, they cannot capture whether a program treats some data in-place or not. Moreover, our relational properties do not focus on a specific data structure, but aims at being generic for any data structure.

Modular analyses that compute invariants by separate analysis of program components [13, 22, 9] use various sorts of abstractions for the behavior of program components. A common pattern uses tables of pairs made of an abstract pre-condition and a corresponding abstract post-condition, effectively defining a sort of cardinal power abstraction [18]. This technique has been used in several shape analyses based on separation logic [8, 28, 33, 7]. We believe this tabular approach could benefit from abstractions of relations such as ours to infer stronger properties, and more concise summaries.

8 Conclusion

While relational properties are harder to abstract than state properties, they are intrinsically more expressive and they offer the ability to make the analysis modular and compositional. In the context of data structures, shape analyses based on separation logic rely on the separating conjunction ($*$) that ensures that two memory regions are disjoint, and on inductive predicates that describe precise structural invariant over complex dynamic data structures. However, separation logic formulas describe a set of states, they cannot describe relations. In this paper, we have introduced a set of logical connectives inspired by separation logic, to describe input-output relations rather than states. We have built upon this logic an abstract domain, and a static analysis based on abstract interpretation that computes conservative abstract relations. We also have extended the relational abstract domain to express more specific and stronger relational properties, in a modular way. Experiments prove its ability to infer expressive relational properties for basic libraries of data structures. Furthermore, it would be more adapted to a compositional inter-procedural analysis.

Acknowledgments. We acknowledge the anonymous reviewers for their constructive comments.

References

1. Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *Conference on Computer Aided Verification (CAV)*, pages 477–490. Springer, 2007.
2. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C specification language, 2008.
3. Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! a framework for higher-order separation logic in coq. In *International Conference on Interactive Theorem Proving (ITP)*, pages 315–331. Springer, 2012.
4. Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for lists: A critical module of contiki verified in frama-c. In *NASA Formal Methods Symposium (NFM)*. Springer, 2018.
5. Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *Conference on Computer Aided Verification (CAV)*, pages 72–88. Springer, 2010.
6. Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 578–589. ACM, 2011.
7. 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.
8. 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.
9. 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.
10. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 247–260. ACM, 2008.
11. Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Electronic Proceedings in Theoretical Computer Science*, pages 161–185. OPA, 2013.
12. Arthur Charguéraud and François Pottier. Temporary read-only permissions for separation logic. In *European Symposium on Programming (ESOP)*, pages 260–286. Springer, 2017.
13. 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.

14. Andreea Costea, Asankhaya Sharma, and Cristina David. Hipimm: Verifying granular immutability guarantees. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 189–193, New York, NY, USA, 2014. ACM.
15. 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.
16. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.
17. 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.
18. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1979.
19. Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Conference on Compiler Construction (CC)*, pages 159–179. Springer, 2002.
20. 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.
21. Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 359–374, New York, NY, USA, 2011. ACM.
22. 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.
23. 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.
24. 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.
25. 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.
26. Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.
27. Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *International Conference on Concurrency Theory (ICC)*, pages 388–402. Springer, 2010.
28. 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.
29. Bart Jacobs and Frank Piessens. The verifast program verifier. Technical report, Department of Computer Science, KU Leuven, Belgium, 2008.
30. 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.
31. Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. In *International Conference on Functional Programming (ICFP)*, pages 311–324. ACM, 2014.
32. 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.
33. 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.
34. 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.
35. Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. In *Symposium on Principles of Programming Languages (POPL)*, volume 52, pages 32–45. ACM, 2017.
36. 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.
37. Peter W O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.
38. Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Conference on Advances in computer science: secure software and related issues*, pages 331–345. Springer, 2006.

39. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics In Computer Science (LICS)*, pages 55–74. IEEE, 2002.
40. 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.
41. Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering (ASE)*, 14(1):87–121, 2007.
42. Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In *European Symposium on Programming (ESOP)*, pages 204–222. Springer, 2003.
43. 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.