Abstract Domains for Automated Reasoning about List-Manipulating Programs with Infinite Data [⋆]

A. Bouajjani[1], C. Drăgoi[2], C. Enea[1], and M. Sighireanu[1]

[1] LIAFA, Univ Paris Diderot & CNRS {abou,cenea,sighirea}@liafa.jussieu.fr
[2] IST Austria,cezarad@ist.ac.at

Abstract. We describe a framework for reasoning about programs with lists carrying integer numerical data. We use abstract domains to describe and manipulate complex constraints on configurations of these programs mixing constraints on the shape of the heap, sizes of the lists, on the multisets of data stored in these lists, and on the data at their different positions. Moreover, we provide powerful techniques for automatic validation of Hoare-triples and invariant checking, as well as for automatic synthesis of invariants and procedure summaries using modular inter-procedural analysis. The approach has been implemented in a tool called CELIA and experimented successfully on a large benchmark of programs.

1 Introduction

Reasoning about heap-manipulating programs can be quite complex and its automatization is a real challenge both from the theoretical and the practical point of view. Indeed, the specification of such a program (consider for instance a sorting algorithm), includes in general various types of constraints, for instance constraints on the structure of the heap (i.e., being a list, acyclic, etc.), on the (unbounded) sizes of the different parts of the heap (i.e., equality of the lengths of two lists), on the (muti)sets of elements stored in different parts of the heap (i.e., equality between the multisets of data stored in two different lists), as well as on the relations existing between the data (potentially ranging over infinite domains) stored in the heap (i.e., sortedness of a list).

For example, the procedure `quicksort` given in Fig. 1 sorts the input list pointed to by the variable `a`. The specification of `quicksort` includes (1) the sortedness of the output list pointed to by `res`, expressed by the formula:

$$\forall y_1, y_2.\ 0 \leq y_1 \leq y_2 < \mathtt{len}(\mathtt{res}) \Rightarrow \mathtt{data}(\mathtt{res}, y_1) \leq \mathtt{data}(\mathtt{res}, y_2) \qquad \text{(i)}$$

where y_1 and y_2 are interpreted as integers and used to refer to positions in the list pointed to by `res`, $\mathtt{len}(\mathtt{res})$ denotes the length of this list, and $\mathtt{data}(\mathtt{res}, y_1)$ denotes the integer stored in the element of `res` at position y_1, and (2) the preservation property saying that input and output lists have the same (multisets of) elements. This property is expressed by the equation

$$\mathtt{ms}(\mathtt{a}^0) = \mathtt{ms}(\mathtt{res}) \qquad \text{(ii)}$$

where $\mathtt{ms}(\mathtt{a}^0)$ (resp. $\mathtt{ms}(\mathtt{res})$) denotes the multiset of integers stored in the list pointed to by `a` at the beginning of the procedure (resp. `res` at the end of the procedure).

⋆ This work was partly supported by the French National Research Agency (ANR) project Veridyc (ANR-09-SEGI-016).

```
typedef struct list {
      struct list *next;
      int data;
      } list;

void split(list *a, int v, list **sm, list **gr){
  list *x=a;
  while (x != NULL){
    if (x->data <= v){
      ...
      /* adds the element pointed
      to by x to sm */
    }
    else{
      ...
      /* adds the element pointed
      to by x to gr */
    }
    x = x->next;
  } }
```

```
list* quicksort(list* a){
  list *left,*right,*pivot,*res,*start;
  int d;
  if (a == NULL || a->next == NULL)
    copy(a,res);
  else {
    d = a->data;
    alloc(&pivot,1);
    pivot->data = d;
    start = a->next;

    split(start,d,&left,&right);

    left = quicksort(left);
    right = quicksort(right);

    res = concat(left,pivot,right);
   }
  return res;
  }
```

Fig. 1: The `quicksort` algorithm on singly-linked lists.

Therefore, reasoning on the correctness of such programs requires designing formal frameworks where such kind of constraints (and their combinations) can be manipulated, i.e., expressed, proved valid, and synthesized.

From the expressiveness point of view, multi-sorted logics interpreted on labelled graphs over infinite alphabets can be naturally considered in this context. As said above, such a logic should allow expressing (1) structural properties on graphs using reachability predicates, as well as (2) constraints on (multi)sets of reachable elements: constraints on their sizes using some arithmetics like Presburger arithmetics for instance, equality/inclusion constraints on the multisets of data they are carrying, etc., and also (3) constraints on the data attached to the different nodes in the graph using some theory on the considered type of data, for instance in the case of integers, it would be possible to consider again Presburger arithmetics to express data constraints.

Given such an expressive specification language, the challenge then is to provide algorithmic techniques allowing to carry out automatically correctness proofs of programs w.r.t. some specifications. (Here we consider partial correctness proofs, i.e., checking safety properties.) This task is not trivial since of course the considered problem is undecidable in general for the considered class of programs and specifications. Nevertheless, our aim is to provide sound techniques that are powerful enough to handle most of the cases that arise in practice.

A first objective is to provide automatic support for pre/post-condition reasoning, assuming that we are given a program together with annotations specifying assumptions and requirements on the configurations at its different control points, including loop invariants and procedure specifications. The aim is to automatize each step in the correction proof using algorithms for checking the validity of Hoare triples, i.e., given a program statement s, a pre-condition ϕ and a post-condition ψ, check that starting from any configuration satisfying ϕ, executing s always leads to a configuration satisfying ψ. Phrased in the logic-based framework mentioned above, this corresponds to checking whether the formula $\mathtt{post}(\phi, St) \Rightarrow \psi$ is valid, where $\mathtt{post}(\phi, St)$ is supposed to be a formula that characterizes the set of all immediate successors of ϕ after executing

St. Therefore, we need to have (1) procedures for computing effectively the formula $\texttt{post}(\phi, St)$ for any given *St* and ϕ, and (2) algorithms for deciding entailments between two formulas in order to check that $\texttt{post}(\phi, St) \Rightarrow \psi$ holds.

Beyond that, a more ambitious objective is to provide algorithms for automatic synthesis of invariants and procedure summaries (i.e., assertions specifying the relations between the inputs and outputs of the procedures). This allows to augment the degree of automation since the user would not need to provide all the necessary annotations for the correctness proof, which is usually cumbersome and quite complex. Instead, he would be able to rely on synthesis techniques that can discover automatically the missing assertions (e.g., strong enough loop invariants) to complete the proof.

To achieve these goals, several problems must be faced. First, we must be able to decide the validity of the manipulated formulas. The problem is that it is very hard to define classes of formulas for which this is possible and that are expressive enough to cover relevant program properties such as those mentioned above, mixing complex constraints on the shape, sizes, and data. In fact, in many cases, the needed assertions are expressed using formulas that are outside the known decidable logics.

As for assertion synthesis, the additional problem is that the space of assertions is infinite, and it is hard to discover the relevant properties that hold for all possible configurations at some point in the program. Especially, it is important to have clever techniques for the generation of universally quantified formulas that capture such properties that may involve in general quite intricate relations between elements of the heap. Naive procedures would not be able to generate accurate enough assertions.

In this work, we propose an approach for addressing these issues based on the framework of abstract interpretation [11]. We focus on the case of (sequential) programs manipulating dynamic linked lists carrying integer numerical data.

First, we consider that constraints are expressed as elements of abstract domains, the latter being equipped with appropriate meet, join, and entailment operations. These operations correspond to approximations of the logical operations of conjunction, disjunction, and logical implication in the sense that the meet (resp. join) under-approximates conjunction (resp. over-approximates disjunction), and the entailment is a sound approximation of logical implication, i.e., if the entailment holds, then necessarily the implication holds also. In addition to these operations, abstract transformers are introduced allowing to define an over-approximation of $\texttt{post}(\phi, St)$, for every statement *St* and constraint ϕ in the considered abstract domain. Therefore, validating Hoare triples in this framework amounts to checking an entailment between two elements of some abstract domain. Notice that entailment checking in this context does not need to be complete in general. But then, the difficulty is of course in the design of the abstract domains (and the associated operations mentioned above) so that they allow expressing the kind of constraints that are needed for reasoning about significant classes of programs, and they offer powerful mechanisms for computing abstract post-images and for checking entailment that are accurate enough to be successful and efficient in practice.

Furthermore, invariant synthesis and procedure summary generation can naturally be done in this framework using intra/inter-procedural analyses. These analyses are defined as fixpoint computations using the abstract domains mentioned above. However, an additional, and quite delicate, issue that must be addressed in this case is how to guarantee termination while ensuring accuracy of the analyses. In particular, quite elaborate extrapolation (or widening) techniques are needed to generate universally quantified formulas that combine ordering and data constraints. Another important issue to address is scalability of the analyses. A natural approach for tackling this issue is to design modular inter-procedural analyses where the analysis of each procedure call is performed locally, by considering only the part of the heap that is accessible by the variables of the procedure. Then, a delicate problem arises which is how to maintain the relations that might exist between the elements of the local heap before and after the procedure call and the rest of the elements in the heap.

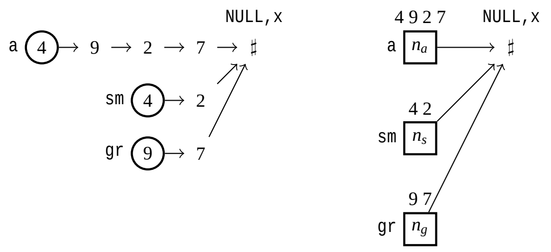
We propose in this paper abstract domains allowing to reason about the various kind of constraints that we have mentioned above, i.e., constraints on the shape of the heap, on the lengths of the lists starting at some locations, on the multisets of the data in these lists, and on the values of the data at different positions on these lists. We show that the proposed domains allow to reason accurately about complex constraints, in particular, our entailment checking techniques allow to establish the validity of formulas that are beyond the capabilities of the existing tools, including the currently most advanced SMT solvers such as CVC3 [2] and Z3 [14].

Moreover, we propose modular inter-procedural analysis techniques allowing to generate automatically invariants as well as procedure summaries. We show that in order to be accurate, modular reasoning requires nontrivial combinations of abstract analyses using different domains, in particular the domain of universally quantified formulas and the domain of multiset constraints. We have implemented the abstract domains and the techniques described in the paper in a tool called CELIA, and we have carried out a large set of experimentations showing the strength and the efficiency of our approach.

2 Programs

We consider a class of strongly typed sequential programs which manipulate singly linked lists. We suppose that all manipulated lists have the same type, i.e., pointer to a record called `list` consisting of one pointer field `next` and one data field `data` of integer type. The generalization to records with several data fields is straightforward.

Syntax: Let *PVar* be a set of variables of type pointer to `list` (*PVar* includes the constant `NULL`) and *DVar* a set of variables interpreted as integers. A *program* is defined by a set of procedures, each of them defined by a tuple $P = (\mathbf{fpi}, \mathbf{fpo}, \mathbf{loc}, G)$, where $\mathbf{loc} \subseteq PVar \cup DVar$ is the vector of local variables, $\mathbf{fpi} \subseteq \mathbf{loc}$ and $\mathbf{fpo} \subseteq \mathbf{loc}$ are the vectors of formal input, resp. output, parameters, and G is an *intra-procedural control flow graph* (CFG, for short). The edges of the CFG are labeled by (1) statements of the form p=`new`, p=q, p`->next=`q, p`->data=`dt, and $\mathbf{y}$=$Q(\mathbf{x})$, where $p,q \in PVar$, dt is a term representing an integer, Q is a procedure name, and $\mathbf{y},\mathbf{x} \subseteq PVar \cup DVar$, (2) boolean conditions on data built using predicates over $\mathbb{Z}$, (3) boolean conditions on pointers of the form p==q, where $p,q \in PVar$, or (4) statements `assert` φ and `assume` φ, where φ is a formula in the logic `SL3` defined in Section 3. The semantics assumes a garbage collector and consequently, the statement `free` is useless. We assume a call-by-value semantics for the procedure input parameters and that each procedure has its own set of local variables. We forbid pointers to procedures and pointer arithmetic.



$$
\begin{aligned}
&\exists n_1, n_s, n_g, \sharp.\\
&\quad \big(\mathtt{ls}(n_a,\sharp) * \mathtt{ls}(n_s,\sharp) * \mathtt{ls}(n_g,\sharp)\\
&\quad \wedge a(n_a) \wedge sm(n_s) \wedge gr(n_g) \wedge x(\sharp)\\
&\qquad \textit{Universally quantified formula:}\\
&\quad \wedge \mathtt{hd}(n_s) \leq v \wedge \mathtt{hd}(n_g) > v\\
&\quad \wedge \mathtt{len}(n_a) = \mathtt{len}(n_s) + \mathtt{len}(n_g)\\
&\quad \wedge \forall y.\, y \in \mathtt{tl}(n_s) \Rightarrow n_s[y] \leq v\\
&\quad \wedge \forall y.\, y \in \mathtt{tl}(n_g) \Rightarrow n_g[y] > v\\
&\qquad \textit{Multiset formula:}\\
&\quad \wedge \mathtt{ms}(n_a) = \mathtt{ms}(n_s) \cup \mathtt{ms}(n_g)\big)
\end{aligned}
$$

(a) (b) (c)

Fig. 2: Heap (a), heap decomposition (b), and `SL3` (c) representations of a program configuration in the procedure `split`.

Semantics: A program configuration consists of a valuation of the variables interpreted as integers and a configuration of the allocated memory. The latter is represented by a labeled directed graph where nodes represent list elements and edges represent values of the field `next` (every node has exactly one successor). The constant `NULL` is represented by the distinguished node $\sharp$. Nodes are labeled with values of the field `data` and program pointer variables. Such a representation is called a *heap*. For example, the valuation $[v \leftarrow 6]$ and the graph in Fig. 2(a) represents a program configuration of the procedure `split` from Fig. 1.

Definition 1 (Heap). *A* heap over *PVar* and *DVar is a tuple $H = (N, S, V, L, D)$ where: (1) N is a finite set of nodes which contains a distinguished node $\sharp$, (2) $S : N \rightharpoonup N$ is a successor partial function s.t. only $S(\sharp)$ is undefined, (3) $V : PVar \to N$ is a function associating nodes to pointer variables s.t. $V(\mathtt{NULL}) = \sharp$, (4) $L : N \rightharpoonup \mathbb{Z}$ is a partial function associating nodes to integers s.t. only $L(\sharp)$ is undefined, and (5) $D : DVar \to \mathbb{Z}$ is a valuation for the data variables.*

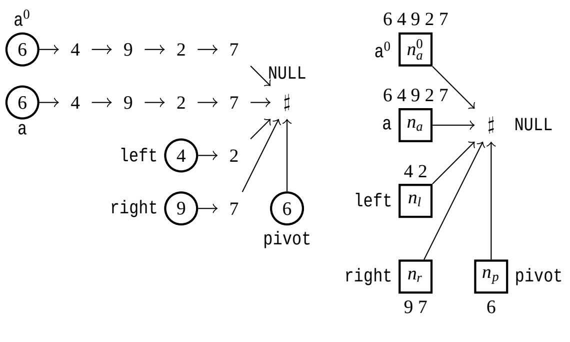
Definition 2 (Simple/Crucial node). *A node labeled with a pointer variable or which has at least 2 predecessors is called* crucial. *Otherwise, it's called a* simple node. □

For example, the circled nodes in Fig. 2(a) are crucial nodes. All the other nodes are simple. Since the semantics we consider is based on garbage collection, the heaps do not contain garbage, i.e., all the nodes of the graph are reachable from nodes labeled with pointer variables.

The *intra-procedural semantics* is defined by a mapping δ which associates to each control point c in the program a set of heaps over *PVar* and *DVar*, representing the set of program configurations reachable at c. As usual, the mapping δ is obtained as the least fixed point of a system of recursive equations. For any statement *St* and any set of heaps $\mathcal{H}$ over *PVar* and *DVar*, $\mathtt{post}_c(\mathcal{H}, St)$ denotes the concrete post-condition operator.

We consider an *inter-procedural semantics* based on relations between program configurations. To have a compositional semantics, we follow the approach of *local* *heap semantics* introduced in [25], where at each procedure call, the callee has access only to the part of the heap that is reachable from its actual parameters, called the *local heap*. For example, in Fig. 3(a), the local heap for the procedure call `quicksort(left)` contains only the nodes reachable from the node labeled by `left`. This approach simplifies the semantics since it avoids the representation of the call stack in the program configurations. However, its use is delicate because the nodes in the local heap of the callee may be shared with the local heaps of other procedures. If during the call these nodes become locally unreachable or deleted, the local heaps of the other procedures must also be updated accordingly. To solve this problem, [25] proposes to maintain for each procedure call the nodes of the local heap from which the shared paths start, but which are not pointed to by the procedure parameters. These nodes are called *cut-points*. Notice that, in general, the number of cut-points may be unbounded. However, there is a significant class of programs for which cut-points are never generated during the execution. This class, called *cut-point free programs* [26], includes programs such as sorting algorithms, traversal of lists, insertion, deletion, etc. In this paper, we consider cut-point free programs and we focus on the problems induced by data manipulation.

For any procedure $P = (\mathbf{fpi}, \mathbf{fpo}, \mathbf{loc}, G)$ and any control point c in P, we consider relations between a program configuration at the entry point of P and a program configuration at c. These relations are represented using a double vocabulary $\mathbf{loc} \cup \mathbf{loc}^0$, where $\mathbf{loc}^0 = \{v^0 \mid v \in \mathbf{loc}\}$ denote the values of the variables in $\mathbf{loc}$ at the entry point of P. A relation associated to P at c is represented by a heap over $\mathbf{loc} \cup \mathbf{loc}^0$ consisting of a valuation for the integer variables in $(\mathbf{loc} \cap DVar) \cup (\mathbf{loc} \cap DVar)^0$ and a graph which is the union of two sub-graphs: G^0 represents the local heap at the entry point of P and G represents the local heap at the control point c. For example, a relation associated to `quicksort` at line 33 is represented by the valuation $\left[d^0 \leftarrow 0, d \leftarrow 6\right]$ and the graph in Figure 3(a) (we suppose that integer variables are initialized to 0). The subgraph containing only the nodes reachable from the node labeled by $\mathtt{a}^0$ represents the input configuration while the rest of the graph represents the configuration at line 33.



(a) (b)

$$
\begin{array}{l}
\exists n_a^0, n_a, n_l, n_r, n_p, \sharp. \\
\quad \big(\mathtt{ls}(n_a^0, \sharp) * \mathtt{ls}(n_a, \sharp) * \mathtt{ls}(n_l, \sharp) \\
\quad * \mathtt{ls}(n_r, \sharp) * \mathtt{ls}(n_p, \sharp) \\
\quad \wedge \mathtt{a}^0(n_a^0) \wedge \mathtt{a}(n_a) \wedge \mathtt{left}(n_l) \\
\quad \wedge \mathtt{right}(n_r) \wedge \mathtt{pivot}(n_p) \\
\qquad \textit{Universally quantified formula:} \\
\quad \wedge \mathtt{hd}(n_l) \leq \mathtt{hd}(n_p) \wedge \mathtt{hd}(n_r) > \mathtt{hd}(n_p) \\
\quad \wedge d = \mathtt{hd}(n_p) \wedge \mathtt{len}(n_p) = 1 \\
\quad \wedge \mathtt{len}(n_a) = \mathtt{len}(n_l) + \mathtt{len}(n_r) + \mathtt{len}(n_p) \\
\quad \wedge \forall y.\, y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \leq \mathtt{hd}(n_p) \\
\quad \wedge \forall y.\, y \in \mathtt{tl}(n_r) \Rightarrow n_r[y] > \mathtt{hd}(n_p) \\
\quad \wedge eq_\forall(n_a, n_a^0) \\
\qquad \textit{Multiset formula:} \\
\quad \wedge \mathtt{ms}(n_a) = \mathtt{ms}(n_l) \cup \mathtt{ms}(n_r) \cup \mathtt{ms}(n_p) \\
\quad \wedge \mathtt{ms}(n_a^0) = \mathtt{ms}(n_a)\big)
\end{array}
$$

(c)

Fig. 3: Heap (a), heap decomposition (b), and `SL3` (c) representations of a relation between program configurations in the procedure `quicksort`.

The inter-procedural semantics is defined by a mapping ρ which associates to each control point c in the CFG of a procedure P a set of heaps over $\mathbf{loc} \cup \mathbf{loc}^0$. The mapping ρ is obtained as the least fixed point of a system of recursive equations [12, 29]. The extension of the postcondition operator $\mathtt{post}_c$ over relations is also denoted by $\mathtt{post}_c$.

3 Specification logic

We introduce hereafter *Singly-Linked List Logic* (SL3, for short) whose models are heaps. Its definition is based on a *decomposition* of heaps obtained as follows. Given a heap H, its decomposition $\overline{H}$ is defined by (1) keeping only some nodes from H but at least all the crucial nodes, (2) adding an edge between any two nodes which are reachable in H, and (3) labeling every node n with a sequence, which contains the integers on the path from H starting in n and ending in its successor in the new graph $\overline{H}$. The valuation for the program integer variables is unchanged. For example, Fig. 2(b), resp. Fig. 3(b), gives a decomposition for the heap in Fig. 2(a), resp. Fig. 3(a).

Syntax of SL3: Formulas in SL3 describe heap decompositions. Let *NVar* be a set of *node variables* interpreted as nodes of the decomposition. An SL3 formula is a disjunction of formulas of the form $\exists N.\ \varphi_G \wedge \varphi_P \wedge \varphi_D$, $N \subseteq NVar$, without free node variables:

- φ_G defines the edges of the decomposition; it contains a set of atomic formulas of the form $\mathtt{ls}(n,m)$ denoting an edge between the nodes n and m, which are connected using the $*$ operator (notation borrowed from separation logic [24]). The operator $*$ states that there is no sharing between the list segments represented by the edges of the decomposition;
- φ_P is a conjunction of formulas of the form $x(n)$ with $x \in PVar$ and $n \in NVar$, expressing the fact that x labels the node n;
- φ_D, called a *data formula*, is a first-order formula that describes the integer variables and the integer sequences labeling the nodes of the decomposition.

Syntax of data formulas: In the following, the sequence of integers labeling the node n is denoted also by n. The formula φ_D has the following form:

$$\Big(E \wedge \bigwedge_{G(\mathbf{y}) \in \mathcal{G}} \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})\Big) \quad \wedge \quad \Big(\bigwedge_i t_1^i = t_2^i\Big), \text{ where}$$

- E is a Presburger formula, called *existential constraint*, which characterizes the first elements of the sequences labeling the nodes of the decomposition (denoted by $\mathtt{hd}(n)$), the lengths of the integer sequences (denoted by $\mathtt{len}(n)$), and the values of the variables from *DVar*,
- $\mathbf{y}$ is a set of *position variables* interpreted as integers representing positions in the sequences labeling the nodes of the decomposition,
- $\mathcal{G}$ is a set of guards $G(\mathbf{y})$, which are conjunctions of (1) formulas that associate vectors of position variables with sequences ($\mathbf{y} \in \mathtt{tl}(n)$ means that the position variables from the vector $\mathbf{y}$ are interpreted as positions in the tail of the sequence n) and (2) a conjunction of linear constraints over the position variables that may use terms of the form $\mathtt{len}(n)$,

– $U(\mathbf{y})$ is a Presburger formula over terms of the form y, $n[y]$, denoting the integer at position y in the sequence n, $\mathtt{len}(n)$, and $\mathtt{hd}(n)$. A term $n[y]$ appears in $U(\mathbf{y})$ only if the guard $G(\mathbf{y})$ contains a constraint $\mathbf{y} \in \mathtt{tl}(n)$ with $y \in \mathbf{y}$. This restriction is used to avoid undefined terms. For instance, if n denotes a sequence of length 2 then the term $n[y]$ with y interpreted as 3 is undefined,
– t_1^i, t_2^i are multiset terms of the form $u_1 \cup \cdots \cup u_s$ ($s \geq 1$ and $\cup$ is the union of multisets) where basic terms u_i are of the form (1) $\mathtt{mhd}(n)$ (resp. d) representing the singleton containing the first integer of the sequence labeling n (resp. the value of d), or (2) $\mathtt{mtl}(n)$ representing the multiset containing all the integers of the sequence n except the first one. As a shorthand, $\mathtt{mhd}(n) \cup \mathtt{mtl}(n)$ is denoted by $\mathtt{ms}(n)$.

For example, the formula from Fig. 2(c) describes the decomposition from Fig. 2(b). Analogously, the formula from Fig. 3(c) describes the decomposition from Fig. 3(b), where the equality of sequences is described by:

$$\begin{aligned} eq_\forall(n,n^0) := \mathtt{hd}(n) = \mathtt{hd}(n^0) \wedge \mathtt{len}(n) = \mathtt{len}(n^0) \wedge \\ \forall y_1, y_2.\ (y_1 \in \mathtt{tl}(n) \wedge y_2 \in \mathtt{tl}(n^0) \wedge y_1 = y_2) \Rightarrow n[y_1] = n^0[y_2] \quad \text{(iii)} \end{aligned}$$

Semantics of SL3: For simplicity, we assume that any two distinct node variables represent two distinct nodes in the decomposition. Given a decomposition $\overline{H}$ and an SL3 formula φ, $\overline{H}$ satisfies φ if there exists a disjunct ψ of φ, which is of the form $\exists N.\ \varphi_G \wedge \varphi_P \wedge \varphi_D$, and an interpretation $\mathcal{I}$ of the node variables in ψ as nodes in $\overline{H}$ s.t. (1) $(\mathcal{I}(n), \mathcal{I}(m))$ is an edge in $\overline{H}$ iff φ_G contains the formula $\mathtt{ls}(n,m)$, (2) $\mathcal{I}(n)$ is labeled with $x \in PVar$ iff φ_P contains the atomic formula $x(n)$, and (3) the integer data in $\overline{H}$ satisfies the properties given by φ_D. Then, a heap H satisfies an SL3 formula φ if there exists a decomposition $\overline{H}$ of H that satisfies φ. The set of heaps satisfying an SL3 formula φ is denoted by $[\varphi]$.

Fragments of SL3: The fragment of SL3 which contains formulas without multiset constraints is denoted by SL3$^{\mathbb{U}}$ while the fragment of SL3 which describes the integer data using only multiset constraints is denoted by SL3$^{\mathbb{M}}$. An SL3 formula is called *succinct* if it describes heap decompositions that do not contain simple nodes.

4 Reasoning about programs without procedure calls

In this section, we present solutions based on abstraction for checking and synthesizing assertions for programs without procedure calls.

4.1 Pre/post condition reasoning

We describe a framework for pre/post-condition reasoning when the annotations are given in SL3. In general, the difficulty is to check entailments of the form $\mathtt{post}(\varphi_{pre}, St) \Rightarrow \varphi_{post}$, where $\mathtt{post}(\varphi_{pre}, St)$ is an SL3 formula that models exactly (over-approximates) the set of heaps $\mathtt{post}_c([\varphi_{pre}], St)$. In the following, we consider only entailments where the heap decompositions described by φ_{post} do not contain simple nodes, i.e., φ_{post} is succinct. This implies that the invariants and the post-conditions we can check must satisfy this restriction, which is usually the case in practice.

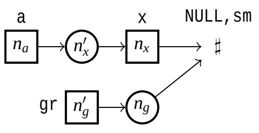
As a running example, we consider the problem of checking an invariant for the `while` loop in the procedure `split` from Fig. 1. This invariant, denoted by *Inv*, contains several disjuncts. Two of them, denoted by ψ_1 and ψ_2, are pictured in Fig. 4(c) and Fig. 4(d); the sub-formula that describes the edges and the labeling with pointer variables of the heap decomposition is represented by a graph. The disjuncts of *Inv* not represented in Fig. 4(c) are similar, i.e., they consider the cases where `x`, `sm`, or `gr` point to `NULL`. Instead of checking the validity of $\texttt{post}(Inv, St) \Rightarrow Inv$, where *St* is the body of the loop, we consider the problem of checking the simpler entailment $(\psi_1^p \vee \psi_2^p) \Rightarrow (\psi_1 \vee \psi_2)$, where ψ_1^p and ψ_2^p are given in Fig. 4(a) and Fig. 4(b), respectively (ψ_1^p is a sub-formula of $\texttt{post}(\psi_1, St)$ while ψ_2^p is a sub-formula of $\texttt{post}(\psi_2, St)$).

Let φ and φ' be two SL3 formulas and consider the problem of checking the validity of the entailment $\varphi \Rightarrow \varphi'$. To efficiently handle the disjunction, we check if for any disjunct ψ of φ there exists a disjunct ψ' of φ' such that $\psi \Rightarrow \psi'$. For example, $(\psi_1^p \vee \psi_2^p) \Rightarrow (\psi_1 \vee \psi_2)$ is valid if $\psi_1^p \Rightarrow \psi_1$ and $\psi_2^p \Rightarrow \psi_2$. This approach is complete only if both SL3 formulae φ and φ' are succinct and if any two disjuncts of φ' describe non-isomorphic heap decompositions (the isomorphism ignores the integer sequences).

Next, to check an entailment of the form $\psi \Rightarrow \psi'$, where ψ is of the form $\exists N.\ \varphi_G \wedge \varphi_P \wedge \varphi_D$ and ψ' is of the form $\exists N'.\ \varphi'_G \wedge \varphi'_P \wedge \varphi'_D$, a first approach is to check that the labeled graphs described by ψ and ψ' are isomorphic and that φ_D entails φ'_D. This check is complete only if both ψ and ψ' are succinct. Then, the entailment between φ_D and φ'_D is valid if (1) the existential constraint of φ_D implies the existential constraint of φ'_D, (2) the right part of any universally quantified implication in φ'_D is implied by the right part of an universally quantified implication in φ_D having a similar guard, and (3) the multiset constraints in φ_D imply the multiset constraints in φ'_D. A sufficient condition to test the validity of $\sqsubseteq_{\mathbb{M}}$ is: for every multiset equality in φ'_D of the form $t_1 = t_2$, φ_D contains the multiset equalities $t_1 = t_1^1 \cup t_1^2 \cdots \cup t_1^p$, $t_2 = t_2^1 \cup t_2^2 \cdots \cup t_2^p$, and for any $1 \leq i \leq p$, $t_1^i = t_2^i$. The approximation for the entailment that we obtain in this way is denoted by $\sqsubseteq$. For example, in Fig. 4, $\psi_2^p \sqsubseteq \psi_2$ and consequently, $\psi_2^p \Rightarrow \psi_2$.

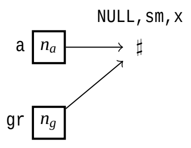
The operator $\texttt{fold}^{\#}$: To prove entailments of the form $\psi \Rightarrow \psi'$, where ψ is not succinct, we define an operator $\texttt{fold}^{\#}$, which computes a succinct SL3 formula that over-approximates ψ (i.e., it eliminates the existential node variables in ψ which represent simple nodes). The extension of $\texttt{fold}^{\#}$ to SL3 formulas is defined by $\texttt{fold}^{\#}(\bigvee_i \psi_i) = \bigvee_i \texttt{fold}^{\#}(\psi_i)$. Clearly, if $\texttt{fold}^{\#}(\psi) \sqsubseteq \psi'$ then $\psi \Rightarrow \psi'$. Such entailments arise naturally when checking loop invariants. Even if we consider a succinct invariant *Inv*, the post-condition operator `post` will unfold the structures and introduce simple nodes. Consequently, *Inv* describes heap decompositions that are not isomorphic to heap decompositions in $\texttt{post}(Inv, St)$ and $\texttt{post}(Inv, St) \sqsubseteq Inv$ does not hold. However, it may happen that $\texttt{fold}^{\#}(\texttt{post}(Inv, St)) \sqsubseteq Inv$ which is enough to prove $\texttt{post}(Inv, St) \Rightarrow Inv$. In the running example, we have that $\texttt{fold}^{\#}(\psi_1^p) \Rightarrow \psi_1$ which implies $\psi_1^p \Rightarrow \psi_1$.

Let ψ be the disjunct of some SL3 formula. In general, $\texttt{fold}^{\#}(\psi)$ is defined such that every maximal path $n_0, n_1, \ldots, n_{k-1}, n_k$ in the graph described by ψ between two crucial nodes n_0 and n_k is replaced by one edge between n_0 and n_k and the integer sequence labeling n_0 in the models of $\texttt{fold}^{\#}(\psi)$ is the concatenation of the integer sequences labeling n_0, n_1,…,n_{k-1} in ψ. For example, $\texttt{fold}^{\#}(\psi_1^p)$ is defined such that the paths n_a, n'_x, n_x and $n'_g, n_g, \sharp$ are replaced by an edge from n_a to n_x and an edge from n'_g



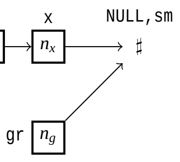
$\texttt{len}(n'_x) = \texttt{len}(n'_g) = 1 \wedge \texttt{hd}(n'_g) = \texttt{hd}(n'_x) > v$
$\wedge\, \texttt{hd}(n_g) > v \wedge \texttt{len}(n_a) = \texttt{len}(n_g)$
$\wedge\, \forall y.\, y \in \texttt{tl}(n_g) \Rightarrow n_g[y] > v$
$\wedge\, \texttt{mhd}(n'_x) = \texttt{mhd}(n'_g) \wedge \texttt{ms}(n_a) = \texttt{ms}(n_g)$

(a) The disjunct ψ_1^p



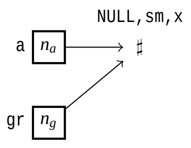
$\texttt{len}(n_a) = \texttt{len}(n_g) \wedge \texttt{hd}(n_g) > v$
$\wedge\, \forall y.\, y \in \texttt{tl}(n_g) \Rightarrow n_g[y] > v$
$\wedge\, \texttt{ms}(n_a) = \texttt{ms}(n_g)$

(b) The disjunct ψ_2^p



$\texttt{hd}(n_g) > v \wedge \texttt{len}(n_a) = \texttt{len}(n_g)$
$\wedge\, \forall y.\, y \in \texttt{tl}(n_g) \Rightarrow n_g[y] > v$
$\wedge\, \texttt{ms}(n_a) = \texttt{ms}(n_g)$

(c) The disjunct ψ_1



$\texttt{len}(n_a) = \texttt{len}(n_g) \wedge \texttt{hd}(n_g) > v$
$\wedge\, \forall y.\, y \in \texttt{tl}(n_g) \Rightarrow n_g[y] > v$
$\wedge\, \texttt{ms}(n_a) = \texttt{ms}(n_g)$

(d) The disjunct ψ_2

Fig. 4: Checking the invariant for the loop in the procedure `split`.

to $\sharp$, respectively. Also, the sequences labeling n_a and n'_g in $\texttt{fold}^{\#}(\psi_1^p)$ are the concatenation of the sequences labeling n_a, n'_x and n'_g, n_g, respectively. The multiset constraints are handled independently of the other constraints. Thus, $\texttt{fold}^{\#}(\psi_1^p)$ contains the multiset constraint $\texttt{ms}(n_a) = \texttt{ms}(n'_g)$, which is obtained by (1) applying an inference rule in ψ_1^p that infers the constraint $\texttt{ms}(n'_x) \cup \texttt{ms}(n_a) = \texttt{ms}(n'_g) \cup \texttt{ms}(n_g)$ (the hypotheses of the inference rule are $\texttt{mhd}(n'_x) = \texttt{mhd}(n'_g)$, $\texttt{ms}(n_a) = \texttt{ms}(n_g)$, and $\texttt{len}(n'_x) = \texttt{len}(n'_g) = 1$) and (2) substituting $\texttt{ms}(n'_x) \cup \texttt{ms}(n_a)$ with $\texttt{ms}(n_a)$ and $\texttt{ms}(n'_g) \cup \texttt{ms}(n_g)$ with $\texttt{ms}(n'_g)$.

The other type of constraints are computed as follows. The properties of the sequences labeling n_a in the models of $\texttt{fold}^{\#}(\psi_1^p)$ are easy to obtain because there are no universal formulas that describe the sequences labeling n_a and n'_x in ψ_1^p. We have to update only the length constraints, i.e., substitute $\texttt{len}(n_a)$ by $\texttt{len}(n_a) - \texttt{len}(n'_x)$ and project out the term $\texttt{len}(n'_x)$. Then, the properties of the sequences labeling n'_g in the models of ψ_1^p are obtained as follows:

– we update the length constraints as in the previous case, i.e., we substitute $\texttt{len}(n_g)$ by $\texttt{len}(n_g) - \texttt{len}(n'_g)$ and project out the term $\texttt{len}(n'_g)$.
– the universal formula that describes n'_g in $\texttt{fold}^{\#}(\psi_1^p)$ has the same guard as the one describing n_g in ψ_1^p. It is obtained by taking into consideration that the tail of n'_g in $\texttt{fold}^{\#}(\psi_1^p)$ is the concatenation between the head and the tail of n_g in ψ_1^p. Thus, we obtain a formula of the form $\forall y.\, y \in \texttt{tl}(n'_g) \Rightarrow (U_1 \vee U_2)$, where U_1 is the property of $\texttt{hd}(n_g)$ and U_2 is the property of $\texttt{tl}(n_g)$. The formula U_1 is $E\left[\texttt{hd}(n_g) \leftarrow n'_g[y]\right]$, where E is the existential constraint of ψ_1^p, and U_2 is obtained from the right part of $\forall y.\, y \in \texttt{tl}(n_g) \Rightarrow n_g[y] > v$ by substituting $n_g[y]$ with $n'_g[y]$, i.e., U_2 is $n'_g[y] > v$.

The relation `Closure`**:** In the example above, the input given to $\mathtt{fold}^{\#}$ contains only universally-quantified implications over one position variable. When these implications contain at least two position variables, the computation of the universally-quantified implications describing the concatenations is more involved. Let us consider the following formula expressing the fact that the sequences labeling n_1 and n_2 are sorted:

$$\psi_3 := \exists n_1, n_2.\ \big(\mathtt{ls}(n_1,n_2) * \mathtt{ls}(n_2,\sharp)\big) \wedge x(n_1) \wedge \mathtt{sorted}(n_1) \wedge \mathtt{less}(n_1) \wedge \mathtt{sorted}(n_2) \wedge \mathtt{less}(n_1)\big)$$
$$\mathtt{sorted}(n) := \forall y_1, y_2.\ ([y_1,y_2] \in \mathtt{tl}(n) \wedge y_1 \le y_2) \Rightarrow n[y_1] \le n[y_2]$$
$$\mathtt{less}(n) := \forall y.\ [y] \in \mathtt{tl}(n) \Rightarrow \mathtt{hd}(n) \le n[y].$$

In $\mathtt{fold}^{\#}(\psi_3)$, the sequence labeling n_1 should be the concatenation of the sequences labeling n_1 and n_2 in ψ_3 (n_2 represents a simple node in ψ_3). The universal formulas describing this sequence should have the same guards as the formulas in ψ_3, i.e., $G_1(y_1,y_2) = [y_1,y_2] \in \mathtt{tl}(n_1) \wedge y_1 \le y_2$ and $G_2(y) = y \in \mathtt{tl}(n_1)$. In the following, we focus on the first guard. An approach similar to the one used for guards of the form $y \in \mathtt{tl}(n)$ could take the union of the properties expressed using the guard $G_1(y_1,y_2)$ on each sequence (n_1 and n_2) and define it as a property of the concatenation. Unfortunately, this definition is unsound. The formula $\forall y_1,y_2.\ G_1(y_1,y_2) \Rightarrow n_1[y_1] \le n_1[y_2]$ is not implied by ψ because the concatenation of two sorted words is not always sorted.

The definition of $\mathtt{fold}^{\#}$ is based on a relation between guards and sets of guards, called `Closure` (see [1] for more details). If we go back to the formula ψ then $\mathtt{sorted}(n_1) \wedge \mathtt{less}(n_1)$ characterizes the data values in the first part of the concatenation and $\mathtt{sorted}(n_2) \wedge \mathtt{less}(n_2)$ characterizes the data values in the second part. But, out of two positions in the concatenation, one might be in n_1 (different from the first element of n_1) and the other one in n_2. Therefore, to define a sound $\mathtt{fold}^{\#}$ operator, we need a universally-quantified implication having as guard $G_3(y_1,y_2) = y_1 \in \mathtt{tl}(n_1) \wedge y_2 \in \mathtt{tl}(n_2)$. In fact, $\mathtt{Closure}(G_1(y_1,y_2))$ is the set of guards $\{G_1(y_1,y_2), G_2(y), G_3(y_1,y_2)\}$. The operator $\mathtt{fold}^{\#}$ combines universal formulas with guards from $\mathtt{Closure}(G_1(y_1,y_2))$ in order to compute the formula of the form $\forall y_1,y_2.\ G_1(y_1,y_2) \Rightarrow U$ (see [5, 1] for more details). If these formulas are not present in the input formula then $\mathtt{fold}^{\#}$ over-approximates it to *true*.

4.2 Invariant synthesis

We consider a static analysis for programs with singly-linked lists based on *abstract interpretation* [11]. We define in [5] a generic abstract domain whose elements represent sets of heaps. Two important instances are $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{U}})$ and $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{M}})$ (the parameter k may be omitted). The elements of $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{U}})$ are $\mathsf{SL3}^{\mathbb{U}}$ formulas and the elements of $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{M}})$ are $\mathsf{SL3}^{\mathbb{M}}$ formulas. The conjunctions of universally-quantified implications from $\mathsf{SL3}^{\mathbb{U}}$ formulas are elements of an abstract domain denoted by $\mathcal{A}_{\mathbb{U}}$ and the conjunctions of equalities between multiset terms from $\mathsf{SL3}^{\mathbb{M}}$ formulas are elements of an abstract domain denoted by $\mathcal{A}_{\mathbb{M}}$. The elements of $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{U}})$ and $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{M}})$ are also called *abstract heap sets*. The abstract values satisfy the following restrictions: (1) any two disjuncts describe non-isomorphic heap decompositions and (2) any disjunct describes a heap decomposition with at most k simple nodes. Also, $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{U}})$ has another two parameters which restrict the form of the universally-quantified formula describing the integer sequences. The first parameter is a set of guards $\mathbb{P}$, also called *guard patterns*, and the second one is a numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$ (such as the *Octagons* abstract domain [22], the *Polyhedra* abstract domain [13], etc.). Then, the formulas belonging to $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{U}})$ are disjunctions of formulas of the form:

$$\exists N.\ \Big(\varphi_G \wedge \varphi_P \wedge E \wedge \bigwedge_{G(\mathbf{y}) \in \mathbb{P}(N)} \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})\Big),$$

where (1) $\mathbb{P}(N)$ is a set of guards obtained from $\mathbb{P}$ by substituting all node variables with elements of N and (2) E and $U(\mathbf{y})$ are elements of the numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$. The order relation between elements of $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{U}})$ (resp. $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{M}})$) is exactly $\sqsubseteq$ restricted to $\mathsf{SL3}^{\mathbb{U}}$ (resp. $\mathsf{SL3}^{\mathbb{M}}$) formulas. If we ignore integer data, the number of heap decompositions without garbage and with at most k simple nodes is bounded. Consequently, the lattice $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{M}})$ is finite and there is no need to define a widening operator. The lattice $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{U}})$ is infinite due to the numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$. We define a widening operator which is parametrized by the widening operator of $\mathcal{A}_{\mathbb{Z}}$.

Unfolding/folding: The analysis over these abstract domains iterates the following two steps: (1) unfolding the structures in order to reveal the properties of some internal nodes in the lists, which makes necessary to introduce some simple nodes and then, (b) folding the structures, in order to keep the graphs finite, by eliminating the simple nodes and in the same time collecting the informations on these nodes using a formula that speaks about data sequences. To terminate, the widening operator is applied.

```
void initEven(list* head) {
   list *headi = head;
   int i = 0;
   while(headi != NULL) {
      headi->data = 2*i;
      headi = headi->next;
      i++;
   }
}
```

Fig. 5

We define sound abstract transformers for the statements in the class of programs we consider. The statements that dereference the `next` pointer field (`x=y->next` and `x->next=y`) introduce simple nodes. The folding step is applied every time the number of simple nodes becomes greater than k. It consists in applying the operator $\mathtt{fold}^{\#}$ described in Sec. 4.1. In particular, this is the crucial step that allows to generate universally quantified properties from a number of relations between a finite (bounded) number of nodes. To make the operator $\mathtt{fold}^{\#}$ precise, we should consider abstract domains $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{U}})$ parametrized by sets of guard patterns $\mathbb{P}$ which are closed under the relation `Closure`, i.e., they include $\mathtt{Closure}(G)$, for any G in $\mathbb{P}$.

We illustrate the unfold/fold mechanism on the procedure `initEven` from Fig. 5. We analyze this program using the abstract domain $\mathcal{A}_{\mathbb{HS}}(1, \mathcal{A}_{\mathbb{U}})$ parametrized by (1) a set of guard patterns consisting of one element $y \in \mathtt{tl}(n)$ and (2) the *Polyhedra* abstract domain. The analysis begins to unroll the loop of the procedure starting from the first $\mathsf{SL3}$ formula given in Fig. 6. This formula represents the set of all heaps that consist of a path between a vertex labeled by `head` and `headi`, and the distinguished node $\sharp$.

Every symbolic execution of the statement `headi=headi->next` in the loop generates a formula with two disjuncts: the first one corresponds to the case when `headi` points to `NULL` (the list traversal ends) and the second one unfolds the structure, i.e., introduces a new node which is pointed to by `headi`. The formulas obtained after unrolling once and thrice the loop are given in Fig. 6. An edge starting in some node n and labeled by 1 means that the formula contains the constraint $\mathtt{len}(n) = 1$. Also, a node n labeled by some integer v means that the formula contains the constraint $\mathtt{hd}(n) = v$.

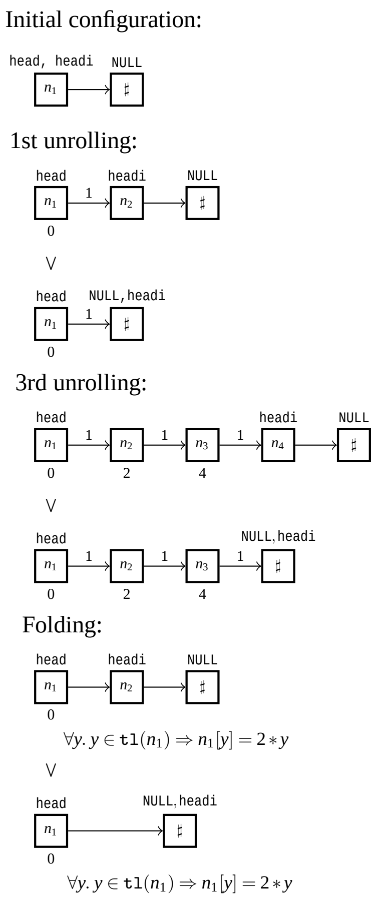


Fig. 6

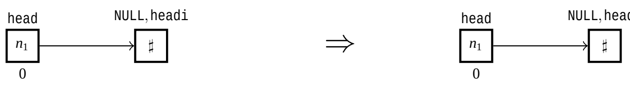
The size of the list pointed to by `head` is potentially unbounded, so the size of the graphs grows at each unrolling. In order to guarantee termination, the analysis manipulates graphs that contain at most one simple node (i.e., $k = 1$). Notice that after the third unrolling of the loop, the graphs contain two simple nodes. To keep the size of the abstract heaps bounded, the analysis eliminates these nodes but, before that, it collects the information that the unrolling of the loop revealed about them. This step is called folding the structure and consists in applying $\mathtt{fold}^{\#}$. We obtain a universal formula that describes the data properties of the nodes that have been eliminated. Because the analysis is parametrized by the pattern $\forall y.\, y \in \mathtt{tl}(n)$, $\mathtt{fold}^{\#}$ generates a universally quantified formula of the form $\forall y.\, y \in \mathtt{tl}(n_1) \Rightarrow U$. To this, it searches for all possible instantiations of the variable y that satisfy the pattern, in this case the nodes labeled by 2 and 4, and it applies the join in the numerical abstract domain between the constraints on these nodes, i.e., $\mathtt{dt}(y) = 2$ and $\mathtt{dt}(y) = 4$. The resulting formula is given in Fig. 6.

The unfolding and folding steps are repeated until the analysis reaches a fixed point. To ensure the convergence of the fixed point computation, apart from bounding the size of the graphs, we use the widening operator of the numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$. In the considered example, widening makes the length constraints converge to the fact that the list pointed to by `head` is greater than or equal to one. Consequently, the universally quantified formula from Fig. 6 is generalized to the entire list.

4.3 A sound decision procedure based on abstraction

In Sec. 4.1, we have shown that for any φ and φ' two $\mathsf{SL3}^{\mathbb{U}}$ formulas, the entailment $\varphi \Rightarrow \varphi'$ is valid if for any disjunct ψ' of φ' there exists a disjunct ψ of φ such that $\mathtt{fold}^{\#}(\psi) \sqsubseteq \psi'$. Notice that $\mathtt{fold}^{\#}(\psi) \sqsubseteq \psi'$ holds only if ψ' contains universally-quantified implications having the same guards as some universally-quantified implications in ψ. For example, the entailment $\psi_4 \Rightarrow \psi_5$ in Fig. 7 is valid but $\psi_4 \not\sqsubseteq \psi_5$ (because ψ_4 is succinct there is no need to apply the operator $\mathtt{fold}^{\#}$). This happens because ψ_4 does not contain an universally-quantified implication having as guard $[y_1, y_2] \in \mathtt{tl}(n_1) \wedge y_2 = y_1 + 1$.

The operator $\mathtt{convert}_{\mathbb{P}}$**:** In order to increase the precision of entailment checking between $\mathsf{SL3}^{\mathbb{U}}$ formulas, we define an operator $\mathtt{convert}_{\mathbb{P}}$ [6], parametrized by a set of



$\forall y.\, y \in \mathtt{tl}(n_1) \Rightarrow n_1[y] = 2 * y$

$$\forall y_1, y_2.\ ([y_1, y_2] \in \mathtt{tl}(n_1) \wedge y_2 = y_1 + 1) \Rightarrow n_1[y_2] = n_1[y_1] + 2$$

Fig. 7: An entailment between two formulas denoted by ψ_4 and ψ_5.

guard patterns $\mathbb{P}$. For any $\mathsf{SL3}^{\mathbb{U}}$ formula φ, $\mathtt{convert}_{\mathbb{P}}(\varphi)$ is an $\mathsf{SL3}^{\mathbb{U}}$ formula equivalent to φ which contains universally-quantified implications having as guards constraints from $\mathbb{P}$. Therefore, for any φ and φ' two $\mathsf{SL3}^{\mathbb{U}}$ formulas, if $\mathtt{convert}_{\mathbb{P}}(\varphi) \sqsubseteq \varphi'$ then $\varphi \Rightarrow \varphi'$. The operator $\mathtt{convert}_{\mathbb{P}}$ is defined as follows:

– we consider a program containing several `while` loops that traverse the list segments constrained by φ. For example, in the case of ψ_4, we consider the program:

```
list *headi = head;
while (headi != NULL)
  headi = headi->next;
```

– the program is analyzed using $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{U}})$ parametrized by a set of guard patterns $\mathbb{P}' = \mathbb{P} \cup \mathbb{P}_\varphi \cup \mathtt{Closure}(\mathbb{P} \cup \mathbb{P}_\varphi)$, where $\mathbb{P}_\varphi$ are the patterns in φ. The precondition is exactly φ. We denote by $\varphi_{\mathbb{P}}$ the postcondition (i.e., the formula describing the configurations reachable at the end of the program) synthesized using this analysis.
– $\mathtt{convert}_{\mathbb{P}}(\varphi)$ is the conjunction of φ and $\varphi_{\mathbb{P}}$.

The formula $\mathtt{convert}_{\mathbb{P}}(\varphi)$ is equivalent to φ because, by definition, $\varphi_{\mathbb{P}}$ is implied by φ. For example, $\mathtt{convert}_{\mathbb{P}_1}(\psi_1)$, where $\mathbb{P}_1$ consists of $y \in \mathtt{tl}(n_1)$, $[y_1, y_2] \in \mathtt{tl}(n_1) \wedge y_2 = y_1 + 1$, and the closure of these two patterns, is a formula which contains both universally quantified implications from Fig. 7 (see [6] for more details). The fact that $\mathtt{convert}_{\mathbb{P}_1}(\psi_1) \sqsubseteq \psi_2$ proves that $\psi_1 \Rightarrow \psi_2$ is valid.

5 Reasoning about programs with procedure calls

In this section, we extend the pre/post condition reasoning framework and the static analysis from the previous section to (recursive) programs with procedure calls.

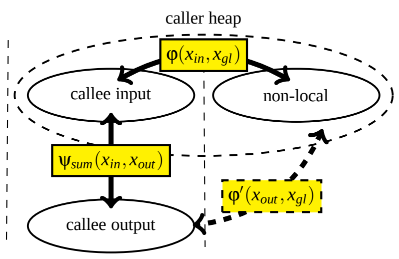
5.1 Pre/post condition reasoning

We assume that, besides loop invariants, each procedure is annotated by a precondition and a postcondition. Following the local heap semantics, they describe only the part of the heap relevant to the procedure. The precondition describes heaps where all nodes are reachable from the input parameters and the postcondition describes relations between the input and the output configurations, i.e., heaps over the double vocabulary $\mathbf{loc} \cup \mathbf{loc}^0$.

The validity of Hoare triples corresponding to procedure calls can be checked as follows. Let P be a procedure annotated by a precondition φ_{pre} and a postcondition φ_{post} and let $\{\varphi_1\}P(\mathbf{ai}, \mathbf{ao})\{\varphi_2\}$, be a Hoare triple, where $\mathbf{ai}$, resp. $\mathbf{ao}$, are the input, resp.

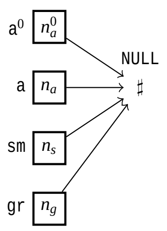
output, actual parameters (the validity of Hoare triples corresponding to $q = P(\mathbf{ai}, \mathbf{ao})$ is checked in a similar manner). This Hoare triple is valid if (1) for any heap H modeled by φ_1, the sub-graph of H containing all the nodes reachable from the actual input parameters **ai** satisfies φ_{pre} and (2) $\texttt{post}(\varphi, P(\mathbf{ai}, \mathbf{ao})) \Rightarrow \varphi_2$. The first condition holds if the entailment $local(\varphi_1) \Rightarrow \varphi_{pre}[\gamma]$ is valid, where $local(\varphi_1)$ is a sub-formula of φ_1 describing only nodes reachable from the actual parameters in **ai** and γ is a substitution that replaces formal parameters with actual parameters. For example, consider the Hoare triple in Fig. 10 for the first recursive call of the procedure `quicksort` in Fig. 1. The sub-formula $local(\varphi_1)$, where φ_1 is the formula in the left of Fig. 10, is $\texttt{ls}(n_l, \sharp) \land \texttt{left}(n_l) \land \texttt{hd}(n_l) \leq \texttt{hd}(n_p) \land \forall y.\, y \in \texttt{tl}(n_l) \Rightarrow n_l[y] \leq \texttt{hd}(n_p)$. Clearly, it implies the precondition of `quicksort`, which states that the input list is acyclic.

Then, $\texttt{post}(\varphi_1, P(\mathbf{ai}, \mathbf{ao}))$ is a disjunction of formulas obtained by combining a disjunct ψ_1 of φ_1 and a disjunct ψ_{post} of φ_{post} s.t. the decomposition of the input heap in ψ_{post} is isomorphic to the decomposition of the local heap in ψ_1 (the isomorphism is denoted by h). Thus, (1) we replace in ψ_1 the sub-formula that describes the local heap (without integer data) with the sub-formula that describes the output heap in ψ_{post} (without integer data), (2) we redirect all edges ending in nodes labeled by actual parameters (from ψ_1) to the nodes labeled by the corresponding formal parameters (from ψ_{post}), and (3) integer data is described by a formula of the form $\sigma = \exists N_0.\, (\varphi_D \land \varphi_{D,post}[h])$, where φ_D (resp. $\varphi_{D,post}$) is the sub-formula of ψ_1 (resp. ψ_{post}) that describes integer data and N_0 is the set of variables denoting nodes from the input heap in ψ_{post} (the isomorphism h is used as a substitution for node variables). Notice that the logic SL3 is extended by allowing existential quantification over node variables in the part that describes the integer data. For example, given the postcondition of `quicksort` in Fig. 9(b) and the formula φ_1 in the left of Fig. 10, $\varphi = \texttt{post}(\varphi_1, \texttt{left} = \texttt{quicksort}(\texttt{left}))$ is the formula in Fig. 11 ($\varphi_{D,\texttt{qst}}$ is given in Fig. 9).


caller heap
$\varphi(x_{in}, x_{gl})$
callee input
non-local
$\psi_{sum}(x_{in}, x_{out})$
$\varphi'(x_{out}, x_{gl})$
callee output
Fig. 8: Relation between caller and callee local heaps.

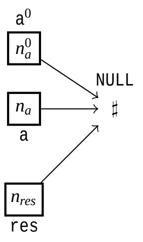
The approach based on local heaps can be too weak for proving the validity of Hoare triples corresponding to procedure calls. Elements in the local heap of the callee are linked at the call point to external elements by some data relation, φ, and the procedure is annotated by some postcondition ψ_{sum} that relates the input heap with the output heap. This situation is depicted in Fig. 8. The problem is how to recover the link φ' between the elements in the callee output heap and the external elements in the caller heap.

Annotations in SL3$^{\mathbb{U}}$ for `quicksort`: For the procedure `quicksort`, annotations in SL3$^{\mathbb{U}}$ are not sufficient to prove that it outputs a sorted list. This procedure takes the first element `d` of the input list `a` as the pivot, splits the tail of `a` into two lists `left` and `right`, where all the elements of `left`, resp. `right`, are smaller, resp. greater, than `d`, and then performs two recursive calls on the lists `left` and `right`, before composing the results, together with `d`, into a sorted list.



$$\begin{array}{l}\mathtt{hd}(n_s) \leq v \land \mathtt{hd}(n_g) > v\\ \land\, \mathtt{len}(n_a) = \mathtt{len}(n_s) + \mathtt{len}(n_g)\\ \land\, eq_{\forall}(n_a, n_a^0)\\ \land\, \forall y.\, y \in \mathtt{tl}(n_s) \Rightarrow n_s[y] \leq v\\ \land\, \forall y.\, y \in \mathtt{tl}(n_g) \Rightarrow n_g[y] > v\\ \land\, \mathtt{ms}(n_a^0) = \mathtt{ms}(n_a) = \mathtt{ms}(n_s) \cup \mathtt{ms}(n_g)\end{array}$$

(a) The postcondition of `split`

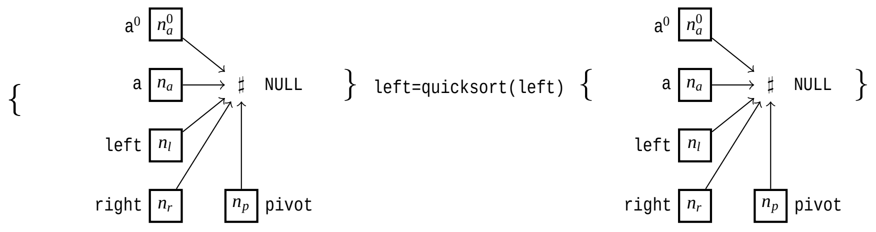


$$\underbrace{\begin{array}{l}\mathtt{len}(n_a) = \mathtt{len}(n_a^0) = \mathtt{len}(n_{res}) \geq 1\\ \land\, eq_{\forall}(n_a, n_a^0) \;\; \land\, sorted(n_{res})\\ \land\, \mathtt{ms}(n_a^0) = \mathtt{ms}(n_a) = \mathtt{ms}(n_{res})\end{array}}_{\varphi_{D,\mathtt{qst}}}$$

(b) The postcondition of `quicksort`

Fig. 9: Postconditions for `split` and `quicksort`.

Assume that the $\mathsf{SL3}^{\mathbb{U}}$ postcondition of `split`, resp. `quicksort`, is the formula in Fig. 9(a), resp. Fig. 9(b), without the multiset constraints. We show that the approach based on local heaps can not be used to prove the validity of the Hoare triple given in Fig.10; for the moment, we ignore the multiset constraints in φ_D. When computing $\varphi' = \mathtt{post}(\varphi'_1, \mathtt{left} = \mathtt{quicksort}(\mathtt{left}))$, where φ'_1 is the formula in the left of Fig.10 without the multiset equalities, the constraint that all the elements of `left` are less than or equal to the pivot is lost. The only constraint over the list pointed to by `left` in φ' is that the list is sorted. The reason for this is twofold: (1) the annotations of `quicksort` describe only the input and the output list and they don't refer to other variables from the context of the call (i.e., they don't contain the property that all the elements of the input list are less than or equal to the pivot) and (2) the postcondition of `quicksort` contains no relation between the elements of the input and the output list because $\mathsf{SL3}^{\mathbb{U}}$ cannot express the fact that a list is a permutation of another list.



$$\varphi_D \begin{cases}\mathtt{hd}(n_l) \leq \mathtt{hd}(n_p) \land \mathtt{hd}(n_r) > \mathtt{hd}(n_p)\\ \land\, d = \mathtt{hd}(n_p) \land \mathtt{len}(n_p) = 1 \land eq_{\forall}(n_a, n_a^0)\\ \land\, \mathtt{len}(n_a) = \mathtt{len}(n_l) + \mathtt{len}(n_r) + \mathtt{len}(n_p)\\ \land\, \forall y.\, y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \leq \mathtt{hd}(n_p)\\ \land\, \forall y.\, y \in \mathtt{tl}(n_r) \Rightarrow n_r[y] > \mathtt{hd}(n_p)\\ \land\, \mathtt{ms}(n_a) = \mathtt{ms}(n_l) \cup \mathtt{ms}(n_r) \cup \mathtt{ms}(n_p)\\ \land\, \mathtt{ms}(n_a^0) = \mathtt{ms}(n_a)\end{cases} \qquad \varphi_D \land sorted(n_l)$$

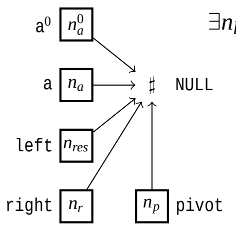
Fig. 10: A Hoare triple in $\mathsf{SL3}$ for the first recursive call in `quicksort`.

Combining universal formulas and multiset constraints: To be able to prove that `quicksort` outputs a sorted list, we must consider annotations with formulas from the full $\mathsf{SL3}$. That is, the list segments are now described by universally-quantified formulas *and* multiset constraints. The new postcondition for `split`, resp. `quicksort`, is the one in Fig. 9(a), resp. Fig. 9(b). Now, the difficulty is to reason in the combined theory.

With the new annotations, we have to check the validity of the Hoare triple from Fig.10 (multiset constraints are now taken into consideration). The crucial point in proving the validity of $\mathtt{post}(\varphi_1, \mathtt{left} = \mathtt{quicksort}(\mathtt{left})) \Rightarrow \varphi_2$, where φ_2 is the formula in the right of Fig.10, is to prove that the data constraints in Fig. 11 imply that all the elements of the sequence n_{res} (the new value of the list pointed to by `left`) are smaller than or equal to $\mathtt{hd}(n_p)$, i.e.,

$$\begin{aligned}\exists n_l.\ \Big(\mathtt{hd}(n_l) \le \mathtt{hd}(n_p) \wedge \forall y.\, y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \le \mathtt{hd}(n_p) \wedge \mathtt{ms}(n_{res}) = \mathtt{ms}(n_l)\Big)\\ \Rightarrow \Big(\mathtt{hd}(n_{res}) \le \mathtt{hd}(n_p) \wedge \forall y.\, y \in \mathtt{tl}(n_{res}) \Rightarrow n_{res}[y] \le \mathtt{hd}(n_p)\Big).\end{aligned} \qquad \text{(iv)}$$

In words, if the sequences n_l and n_{res} have the same multisets of elements and all elements of n_l are less than the pivot then, the latter also holds about the elements of n_{res}. Notice that the operator $\sqsubseteq$ from Sec. 4.1 is not precise enough to prove this entailment.



$$\left.\begin{aligned}\exists n_l.\ (\ &\mathtt{hd}(n_l) \le \mathtt{hd}(n_p)\\ &\wedge \forall y.\, y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \le \mathtt{hd}(n_p)\\ &\wedge \mathtt{ms}(n_{res}) = \mathtt{ms}(n_l))\\ &\wedge \ldots)\end{aligned}\right\}\exists n_l.\ (\varphi_D \wedge \varphi_{D,\mathtt{qst}}\left[n_a^0 \leftarrow n_l\right])$$

Fig. 11: The formula $\varphi = \mathtt{post}(\varphi_1, \mathtt{left} = \mathtt{quicksort}(\mathtt{left}))$.

We define an operator called `stregthen` [6] which can be used to prove such implications. It considers the same program as in $\mathtt{convert}_{\mathbb{P}}$, consisting of a sequence of loops that traverse the list segments. Then, it performs an analysis of this program using a partially reduced product [10] between the domain of abstract heap sets with universal formulas, $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$, and the domain of abstract heap sets with multiset constraints, $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$. The elements of this product are pairs from $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}) \times \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$. Almost all the abstract transformers are defined by $F^{\#}(A_1,A_2) = (F^{\#}_{\mathbb{U}}(A_1), F^{\#}_{\mathbb{M}}(A_2))$, for any $(A_1,A_2) \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}) \times \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$, where $F^{\#}_{\mathbb{U}}$ is the abstract transformer in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $F^{\#}_{\mathbb{M}}$ is the abstract transformer in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$. The only exception is the abstract transformer for `p=q->next`, denoted by $G^{\#}$, which is defined by $G^{\#}(A_1,A_2) = \sigma(G^{\#}_{\mathbb{U}}(A_1), G^{\#}_{\mathbb{M}}(A_2))$, where σ is a partial reduction operator that transfers information between the two abstract elements. To check the validity of $\varphi \Rightarrow \varphi_2$, the analysis starts from a precondition defined as a pair $(\varphi_{\mathbb{U}}, \varphi_{\mathbb{M}})$, where $\varphi_{\mathbb{U}}$ is obtained from φ by removing all multiset constraints and $\varphi_{\mathbb{M}}$ is obtained from φ by removing all universally-quantified implications. The output of `stregthen` is the conjunction between the input formula and the postcondition synthesized by the analysis. In this case, applying `stregthen` on the formula φ, we obtain $\varphi \wedge \mathtt{hd}(n_l) \le \mathtt{hd}(n_p) \wedge \forall y.\, y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \le \mathtt{hd}(n_p)$. Now, the fact that $\varphi \sqsubseteq \varphi_2$ holds proves the validity of $\varphi \Rightarrow \varphi_2$ which implies the validity of the Hoare triple from Fig. 10.

5.2 Synthesis of procedure summaries

For programs with procedure calls, we define a compositional analysis such that the summary of a procedure is computed only once and then reused whenever the procedure is called. Again, in order to solve the problems raised by the use of local heaps, we strengthen the analysis in the domain of universally-quantified formulas with the analysis in the domain of multiset constraints. Thus, we define an abstract domain which is a partial reduced product between $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$. The partial reduction operator is exactly `strengthen` and it is used in the abstract transformers for procedure returns and `assert` statements. The analysis over this partial reduced product is able for instance to synthesize the expected summary for the procedure `quicksort`.

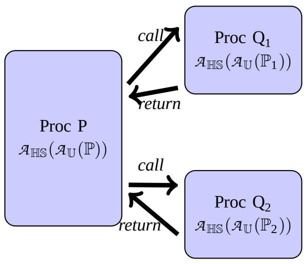

call
return
Proc P
$\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}(\mathbb{P}))$
Proc Q$_1$
$\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}(\mathbb{P}_1))$
call
return
Proc Q$_2$
$\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}(\mathbb{P}_2))$

Fig. 12

Another problem that we address for the design of a compositional analysis is due to the use of patterns for guards of universally-quantified implications. Indeed, the analysis of different procedures may need to use different sets of patterns and therefore, it is important to be able to localize the choice of these patterns to each procedure. Otherwise, it would be necessary to use a set of patterns that includes the union of all the sets that are used during the whole analysis. This would obviously make the analysis inefficient.

Consequently, during the analysis, at procedure calls and returns, we need to switch from an abstract domain of formulas parametrized by some set of patterns, say $\mathbb{P}$, to an abstract domain parametrized by another set of patterns $\mathbb{P}_1$ or $\mathbb{P}_2$ as shown in Figure 12 ($\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}(\mathbb{P}))$ denotes the domain of abstract heap sets with universally-quantified implications parametrized by the set of patterns $\mathbb{P}$). This transformation is defined using the operator $\texttt{convert}_{\mathbb{P}}$ (see [6] for more details).

6 Experimental results

We have implemented the inter-procedural analysis in a tool called CELIA [9]. CELIA is a plugin of the FRAMA-C platform [8], thus taking as input annotated C programs. CELIA instantiates the generic module FIXPOINT (`http://gforge.inria.fr/`) of fixpoint computation over control-flow graphs with the implementation of the abstract domains $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$ and their abstract transformers. The implementation of the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ domain considers the patterns $y \in \texttt{tl}(w)$, $(y_1,y_2) \in \texttt{tl}(w) \wedge y_1 \leq y_2$, $(y_1,y_2) \in \texttt{tl}(w) \wedge y_2 = y_1 + 1$, and $y_1 \in \texttt{tl}(w_1) \wedge y_2 \in \texttt{tl}(w_2) \wedge y_1 = y_2$ and it is generic on the numerical domain $\mathcal{A}_{\mathbb{Z}}$ used to represent data and length constraints. For this, we use the APRON platform [20] to access domains like octagons or polyhedra.

Benchmark: We have applied CELIA to a benchmark of C programs which is available on the web site of CELIA. The benchmark includes the basic functions that are used in usual libraries on singly-linked lists, for example the GTK `gslist` library which is part of the Linux distribution. These functions belong to several classes: (1) (recursive) functions performing elementary operations on list: adding/deleting the first/last element, initializing a list of some length, (2) (recursive) functions performing a traversal of one resp. two lists, without modifying their structures, but modifying their data, (3) functions computing from one resp. two input lists some output parameters of type list or integer, and (4) sorting algorithms on lists. The benchmark also contains programs which do several calls of the above functions on lists. For example, we handle some programs manipulating chaining hash tables. For that, we use abstraction techniques (slicing, unfolding fixed-size arrays) available through the Frama-C platform.

We have used CELIA for checking equivalence between sorting algorithms. The `strengthen` operation plays an essential role. Let P_1 and P_2 be two sorting procedures working on two input lists I_1 and I_2, and producing two outputs O_1 and O_2. The equivalence of P_1 and P_2 is reduced to the validity of the implication

$$\begin{aligned}\big(\mathit{equal}(I_1,I_2) \wedge\ \mathit{sorted}(O_1) \wedge \mathtt{ms}(I_1) = \mathtt{ms}(O_1)&\\ \wedge\ \mathit{sorted}(O_2) \wedge \mathtt{ms}(I_2) = \mathtt{ms}(O_2)\big)&\\ \Rightarrow \mathit{equal}(O_1,O_2),&\end{aligned} \tag{v}$$

where *equal* and *sorted* are expressed by universally quantified implications as in SL3. Our techniques are able to find that this formula is indeed valid. For instance, this entailment and the one in (iv) can not be proved using SMT solvers like CVC3 [2] and Z3 [14] (the multiset equality of two sequences $\mathtt{ms}(n_1) = \mathtt{ms}(n_2)$ is rewritten as $\exists m.\ \mathit{permutation}(m) \wedge \forall i.\ n_1[m[i]] = n_2[i]$, where $\mathit{permutation}(m)$ expresses the fact that the sequence m defines a permutation).

7 Conclusions and related work

The paper presents a logic-based framework the verification and the analysis of programs with lists and data. It introduces a family of abstract domains whose elements are first-order formulas that describe the shape/size of the allocated memory and the scalar data stored in the list cells. The latter is characterized using universal formulas or multiset constraints. The elements of these abstract domains can be used as annotations within pre/post-condition reasoning. In this context, we introduce sound procedures for checking the validity of Hoare triples. Then, we define an accurate inter-procedural analysis that is able to automatically synthesize invariants and procedure summaries. This analysis is compositional and it is based on unfolding/folding the program data structures. The precision is obtained using partial reduction operators, which allow to combine analyses over different abstract domains. Overall, our framework allows to combine smoothly pre-post condition reasoning with assertion synthesis.

Related Work: Assertion synthesis for programs with dynamic data structures has been addressed using different approaches, like constraint solving, e.g. [3], abstract interpretation, e.g., [7,12,19,15–18,23,28,26,27,30], Craig interpolants [21], and automata-theoretic techniques [4].

Several works [19,15,23] consider invariant synthesis for programs with uni-dimensional arrays of integers. The class of invariants they can generate is included in the one handled by our approach using $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$. These techniques are based on an automatically generated finite partitioning of the array indices. We consider a larger class of programs for which these techniques can not be applied. The analysis introduced in

[23] for programs with arrays can synthesize invariants on multisets of the elements in array fragments. This technique differs from ours based on the domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$ by the fact that it can not be applied directly to programs with dynamic lists.

In [18], a synthesis technique for universally quantified formulas is presented. Our technique differs from this one by the type of user guiding information. Indeed, the quantified formulas in [18] are of the form $\forall \mathbf{y}.\ F_1 \Rightarrow F_2$, where F_2 must be given by the user. In contrast, our approach fixes the formulas in left hand side of the implication and synthesizes the right hand side. The two approaches are in principle incomparable.

Concerning the approaches based on abstract interpretation which can handle procedure calls, most of them [7, 12, 17, 26, 27] focus on shape properties and do not consider constraints on sizes or data. The approach in [26] can synthesize procedure summaries that describe data if the instrumentation predicates which guide the abstraction speak about data. Providing patterns is simpler than providing instrumentation predicates on data because patterns contain only constraints between (universally-quantified) positions (in the left-hand-side of the implication) and no constraints on data. Actually, patterns are in many cases simple (ordering/equality constraints) and can be discovered using natural heuristics based on the program syntax or proposed by the user, whereas constraints on data can be more complex. Our approach allows to discover (maybe unpredictable) data constraints for given guard patterns. The analysis in [17] combines a numerical abstract domain with a shape analysis. It is not restricted by the class of data structures but the generated assertions describe only the shape and the size of the heap.

References

1. Cezara Drăgoi. *Automated verification of heap-manipulating programs with infinite data*. PhD thesis, University Paris Diderot - Paris 7, 2011.
2. C. Barrett and C. Tinelli. CVC3. In *Proc. of CAV*, pages 298–302, 2007.
3. Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Proc. of VMCAI*, pages 378–394, 2007.
4. Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *Proc. of CAV*, pages 517–531, 2006.
5. Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *Proc. of CAV*, pages 72–88, 2010.
6. Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proc. of PLDI*, pages 578–589, 2011.
7. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proc. of POPL*, pages 289–300, 2009.
8. CEA. *Frama-C Platform*. `http://frama-c.com`.
9. Celia plugin. http://www.liafa.jussieu.fr/celia.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL*, pages 269–282, 1979.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, pages 238–252, 1977.
12. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In *Proc. of IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277, 1977.

13. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL*, pages 84–96, 1978.
14. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, pages 337–340, 2008.
15. Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL*, pages 338–350, 2005.
16. Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Proc. of SAS*, pages 240–260, 2006.
17. Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *Proc. of POPL*, pages 239–251, 2009.
18. Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
19. Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.
20. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of CAV*, pages 661–667, 2009.
21. Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *Proc. of CAV*, pages 193–206, 2007.
22. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
23. Valentin Perrelle and Nicolas Halbwachs. An analysis of permutations in arrays. In *Proc. of VMCAI*, pages 279–294, 2010.
24. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*. IEEE Computer Society, 2002.
25. Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proc. of POPL*, pages 296–309, 2005.
26. Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proc. of SAS*, pages 284–302, 2005.
27. Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. In *Proc. of POPL*, pages 173–186, 2011.
28. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
29. Micha Sharir and Amir Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two approaches to interprocedural data flow analysis, pages 189–234. Prentice-Hall, 1981.
30. Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proc. of VMCAI*, pages 335–348, 2009.