On Inter-Procedural Analysis of Programs with Lists and Data

Ahmed Bouajjani Cezara Drăgoi Constantin Enea Mihaela Sighireanu
LIAFA, University Paris Diderot & CNRS
{abou,cezarad,cenea,sighirea}@liafa.jussieu.fr

Abstract

We address the problem of automatic synthesis of assertions on sequential programs with singly-linked lists containing data over infinite domains such as integers or reals. Our approach is based on an accurate abstract inter-procedural analysis. Program configurations are represented by graphs where nodes represent list segments without sharing. The data in these list segments are characterized by constraints in abstract domains. We consider a domain where constraints are in a universally quantified fragment of the first-order logic over sequences, as well as a domain constraining the multisets of data in sequences.

Our analysis computes the effect of each procedure in a local manner, by considering only the reachable part of the heap from its actual parameters. In order to avoid losses of information, we introduce a mechanism based on unfolding/folding operations allowing to strengthen the analysis in the domain of first-order formulas by the analysis in the multisets domain.

The same mechanism is used for strengthening the sound (but incomplete) entailment operator of the domain of first-order formulas. We have implemented our techniques in a prototype tool and we have shown that our approach is powerful enough for automatic (1) generation of non-trivial procedure summaries, (2) pre/post-condition reasoning, and (3) procedure equivalence checking.

1. Introduction

Automatic synthesis of valid assertions about programs, such as loop invariants or procedure summaries, is an important and highly challenging problem. In this paper, we address this problem for sequential programs manipulating singly-linked lists with unbounded data such as integers or reals. These programs may contain procedure calls, and actually they are in many cases naturally written using recursive procedures.

Assertions about these programs typically involve constraints on the shape of the structures, their sizes, the data values contained in the memory cells, the multisets of their data, etc. Consider for instance the algorithm `quicksort` in Figure 1 that sorts the input list pointed to by the variable `a` (and where the call `split(start,d,&left,&right)` copies all the cells of the list pointed to by `start` which have data larger than `d` in the list `right`, and all the other ones in the list `left`). The specification of `quicksort` includes (1) the sortedness of the output list pointed to by `res`, expressed by the formula:

$$\forall y_1,y_2.\ 0 \le y_1 \le y_2 < \mathtt{len}(\mathtt{res}) \Rightarrow \mathtt{data}(y_1) \le \mathtt{data}(y_2) \quad \text{(A)}$$

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 $\mathtt{data}(y_1)$ denotes the integer stored in the element of `res` at position y_1, and (2) the preservation property saying that the multiset of data of the input list `a` is equal to the multiset of data of the output list `res`. This property is expressed by

$$\mathtt{ms}(\mathtt{a}^0) = \mathtt{ms}(\mathtt{res}) \quad \text{(B)}$$

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).

```
typedef struct list {
  int data;
  struct list *next;
 } list;
list * quicksort(list *a){
  list  *left,*right,*pivot,*res,*start;
  int d;
  if (a == NULL || a->next == NULL)
    res = clone(a);
  else {
    d = a->data;
    pivot = create(1); // list of length 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; }
```

Figure 1. The `quicksort` algorithm on singly-linked lists.

We propose an approach for automatic assertion synthesis based on inter-procedural analysis within the framework of abstract interpretation [7], that is, we consider abstract domains for expressing constraints on relations between program configurations, and we define compositional techniques for computing procedure summaries concerning various aspects such as shapes, sizes, and data.

We build on the work by Bouajjani et al. in [2] where they define an accurate *intra*-procedural abstract analysis for synthesizing invariants of programs with lists and without procedure calls. In their approach, abstract domains are defined where elements are pairs composed of a heap backbone and an abstract data constraint. The heap backbone is an abstraction of the *heap graph* (the graph representing the allocated memory) where only a bounded number of nodes are kept, including all sharing nodes and all the nodes pointed to by program variables. An edge in the backbone represents a path (without sharing nodes) relating the source and target nodes in the original heap. The data constraint is given as an element of some abstract domain and allows to specify properties of the data sequences represented by the edges of the heap backbone. The provided analysis in that paper is based on (a) unfolding the structures

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06…$10.00

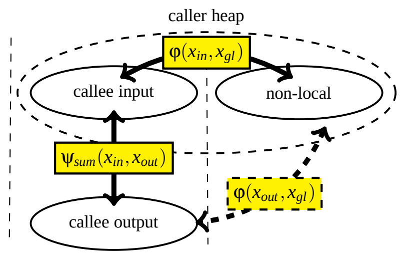

caller heap
$\varphi(x_{in}, x_{gl})$
callee input
non-local
$\psi_{sum}(x_{in}, x_{out})$
$\varphi(x_{out}, x_{gl})$
callee output

Figure 2. Relation between caller and callee local heaps.

in order to reveal the properties of some internal nodes in the lists, which makes necessary to introduce in the structures some nodes, called simple nodes, others than the sharing nodes or those pointed to by variables, and then (b) folding the structures by eliminating the simple nodes and in the same time collecting the informations on these nodes using a formula that speaks about sequences of data. The analysis is iterated several times, which may lead to additional unfoldings and foldings. Then, widening techniques on numerical domains are used in order to force termination. Several abstract domains are defined for the analysis, and in particular, domains where elements are (1) formulas in a universally quantified fragment of the first-order logic over data words, or (2) conjunctions of equality constraints between unions of multisets of data in words. The formulas in the first abstract domain contain a (quantified) universal part which is a conjunction of formulas $\forall \mathbf{y}.\ (P \Rightarrow U)$, where $\mathbf{y}$ is a vector of variables interpreted as positions in the words, P is a constraint on the positions (seen as integers) associated with the $\mathbf{y}$'s, and U is a constraint on the data values at these positions. It is assumed that P is obtained from a finite set of fixed *patterns* corresponding to, e.g., order constraints or difference constraints. While the techniques presented in [2] are strong enough to generate complex invariants for iterative programs, they cannot be applied for compositional computation of procedure summaries.

In this paper, we propose an extension of the approach in [2] by defining new techniques for accurate *inter*-procedural abstract analysis. This extension is not trivial due to many delicate problems that appear when addressing the compositionality issue. Indeed, in the spirit of [23], at each procedure call, the callee has only access to the part of the heap that is reachable from its actual parameters. The use of such local heaps is delicate due to the fact that there are relations between the elements of the local heap of the callee, and of the heaps of the procedures that are in the call stack. If these relations are lost during the analysis, this one can be unsound in some cases, or very imprecise in others. However, it is not feasible to maintain explicitly these relations during the analysis. Let us examine this crucial problem.

This problem has been addressed in [23] in a framework where data are not considered. In this case, the relations inter-local heaps are due to reachability: nodes in the local heap of the callee can be reachable from the other heaps through paths that do not contain nodes pointed to by the parameters of the procedure (the entry points of the local heap). If during the call these nodes become locally unreachable and deleted, the analysis becomes unsound. The solution to this proposed in [23] consists in maintaining along the calls the points, called cutpoints, where these (inter-local heap) paths enter local heaps. This is a tricky problem since in general the number of cutpoints may be unbounded. However, there is a significant class of programs for which cutpoints are never generated during the analysis. The class of such cutpoint free programs [24] includes programs such as sorting algorithms, traversal of lists, insertion, deletion, etc. In this paper, we consider cutpoint free programs and focus on the problems induced by data manipulation.

Indeed, even for cutpoint free programs, the problem above persists when data constraints are considered as we do in our framework. The reason is that elements in the local heap of a procedure can be related to elements in the rest of the heap with data constraints such as equality, ordering, etc. This situation is depicted in Figure 2. Elements in the local heap of the callee are linked at the call point to external elements by some data relation, φ, and the analysis generates a summary ψ_{sum} of the procedure relating the input heap with the output heap. Then, the problem is whether there is a link φ between the elements in the callee output heap and the external elements in the caller heap. This problem depends on the accuracy of the used summarization technique. Consider for instance the `quicksort` procedure that takes the first element `d` of the input list `a` as the pivot, splits the tail of the list `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. After the recursive call at line 16 on `left`, the information we obtain from the analysis with the domain of first-order formulas is that the output list `left'` is sorted. Since we had already the information that the elements of the input list `left` were all less than `d`, we must infer after the return from that call that the elements of `left'` are also less than `d`. But, since the link between `d` and the elements of `left` has not been passed to the recursive call, this information cannot be computed. This is because the used abstract domain cannot express the fact that a list is a permutation of another list (which requires formulas beyond the universally quantified fragment). Again, maintaining all the relations between the elements of the local heaps and the external elements is not feasible. Our solution to this problem is based on strengthening the analysis in the domain of first-order formulas with the analysis in the domain of multiset constraints. Indeed, for the `quicksort` example, knowing that `left` and `left'` have the same multisets of elements should allow to infer from the fact that all elements of `left` are less than `d`, that the same fact also holds about the elements of `left'`.

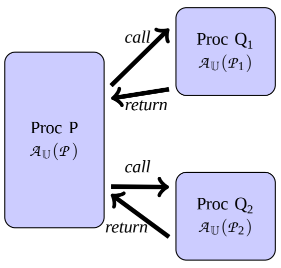

Proc P
$\mathcal{A}_{\mathbb{U}}(\mathcal{P})$
call
return
Proc Q$_1$
$\mathcal{A}_{\mathbb{U}}(\mathcal{P}_1)$
call
return
Proc Q$_2$
$\mathcal{A}_{\mathbb{U}}(\mathcal{P}_2)$

Figure 3. Compositional analysis with patterns.

Another problem that must be addressed for the design of compositional analysis is due to the use of patterns P for left-hand sides of implications in the first-order formulas. Indeed, the analysis of different procedures may need the use of 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 including the union of all the sets that are used during the whole analysis, and this would obviously make the analysis inefficient. For instance, the computation of the formula $(\forall y.\ 0 \le y < \texttt{len}(\texttt{left}) \Rightarrow \texttt{data}(y) \le d)\ \wedge\ (\forall y.\ 0 \le y < \texttt{len}(\texttt{right}) \Rightarrow \texttt{data}(y) > d)$ describing the effect of the procedure `split` called by `quicksort` does not need the use of the pattern $0 \le y_1 \le y_2 < \texttt{len}(\texttt{res})$ that is used for the generation of the sortedness property. 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 $\mathcal{P}$, to an abstract domain parametrized by another set of patterns $\mathcal{P}_1$ or $\mathcal{P}_2$ as shown in Figure 3 ($\mathcal{A}_{\mathbb{U}}(\mathcal{P})$ denotes the domain of first-order formulas parametrized by the set of patterns $\mathcal{P}$).

The two problems exposed above show that in order to define a compositional and accurate inter-procedural analysis, we need to define an operation for composing abstract domains (e.g., first-order formulas with multiset constraints, or first-order formulas of different types). We propose in this paper a mechanism which allows to solve these problems. This mechanism is based on unfolding/folding operations which can be used, at procedure calls and returns, to (1) compute an over-approximation of the intersection between a first-order formula and a multiset constraint and (2) to convert universal formulas defined over a set of patterns $\mathcal{P}_1$ to formulas defined over a set of patterns $\mathcal{P}_2$.

Beyond compositional summary computation, the operation we define for combining abstract domains allows to tackle two other interesting problems. First, it allows to define an entailment operation on combined constraints of the form $\phi \wedge \psi \Rightarrow \phi'$ where ϕ and ϕ' are two universal first-order formulas (potentially over different sets of patterns), and ψ is a multiset constraint. This provides a lightweight sound (but not complete) decision procedure for such kind of formulas, which is useful for carrying out pre-post condition reasoning. Furthermore, our techniques are accurate enough to be used for automatic procedure equivalence checking. It is easy to see that this problem can be reduced to inter-procedural analysis, provided that it is possible to express equality between structures, and derive such properties. This is not trivial in general since reasoning about combined abstract domains is needed. For instance, to check that two sorting procedures P_1 and P_2 are equivalent, it is possible to call each of them on two identical input lists I_1 and I_2, and then assert that the two outputs O_1 and O_2 are equal. Assuming that the summary of P_i is $sorted(O_i) \wedge \mathtt{ms}(I_i) = \mathtt{ms}(O_i)$, for $i \in \{1,2\}$, this amounts to check the validity of the formula:

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

where the predicates *equal* and *sorted* are expressed by first-order formulas. Our techniques are able to find that this formula is indeed valid.

We have implemented our techniques and carried out several experiments showing the strength of our approach.

2. Programs

Let *PVar* be a set of variables of type reference to a record type called `list` defined by a single reference field `next` and one data field `data` of integer type. We consider that *PVar* includes the constant `NULL`. Also, let *DVar* be a set of variables interpreted as integers. The generalization to record types with several data fields and data fields of different basic types is straightforward.

Syntax: A *program* is defined by a set of procedures, each of them represented by its *intra-procedural control flow graph* (CFG, for short). Formally, a procedure P is a tuple $(\mathbf{fpi},\mathbf{fpo},\mathbf{loc},G)$, 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 its CFG. W.l.o.g., we suppose that the CFG of P contains a unique *entry point* s_P and a unique *exit point* e_P. Its edges 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 (built over terms of the form $d \in DVar$ and p`->data` using operations over $\mathbb{Z}$), 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 of the form `assert` φ and `assume` φ, where φ is a formula (see Section 4 for more details concerning the syntax of φ). The semantics assumes a garbage collector and consequently, the statement `free` is useless. We assume a call-by-value semantics for the procedure parameters and that each procedure has its own set of local variables. We forbid pointers to procedures and pointer arithmetic.

The *inter-procedural control flow graph* (ICFG, for short) of a program is defined as usual by replacing edges labeled by procedure calls with (1) a *call to start* edge labeled by $\langle \mathbf{call}\ \mathbf{y}{=}Q(\mathbf{x})\rangle$, and (2) an *exit to return* edge labeled by $\langle \mathbf{ret}\ \mathbf{y}{=}Q(\mathbf{x})\rangle$.

Semantics: A program configuration is a valuation of the variables interpreted as integers together with a configuration of the allocated memory. The latter is represented by a labeled directed graph where nodes correspond to objects of type `list` and edges correspond to values of the reference field `next`. The nodes are labeled by the values of the field `data` and by the program pointer variables which are pointing to the corresponding objects. The constant `NULL` is represented by a distinguished node $\sharp$. Such a representation for a program configuration is called a *heap.*

DEFINITION 2.1 **(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. *$S : N \times N$ is a set of edges such that S contains at most one edge (n,n'), for any $n \in N$, and S does not contain an edge $(\sharp,n)$ with $n \in N$,*
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 2.2 **(Simple/crucial node).** *A node which is labeled by a pointer variable or which has at least two predecessors is called a* crucial node. *Otherwise, it is called a* simple node.

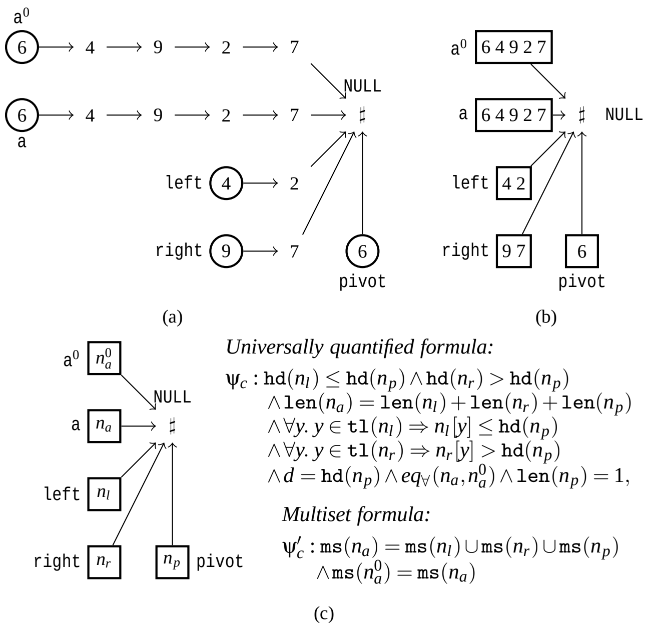
For any procedure $P = (\mathbf{fpi},\mathbf{fpo},\mathbf{loc},G)$ and any control point c in P, we consider relations between program configurations at the entry point of P and program configurations at c. These relations can be represented using a double vocabulary $\mathbf{loc} \cup \mathbf{loc}^0$, 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 $\mathbf{loc} \cup \mathbf{loc}^0$ containing a valuation for the integer variables in $(\mathbf{loc} \cap DVar) \cup (\mathbf{loc} \cap DVar)^0$ together with a graph which is the union of two sub-graphs: G^0 represents the heap at the entry point of P and G represents the heap at the control point c. For example, a relation associated to `quicksort` at line 16 is represented by the valuation $\left[d^0 = 0, d = 6\right]$ and the labeled graph in Figure 4(a) (we suppose that local integer variables are initialized to 0). The nodes which correspond to objects pointed to by program variables are circled. The subgraph containing only nodes reachable from the node labeled by $\mathtt{a}^0$ represents the input configuration while the rest of the graph represents the heap configuration at line 16.

The concrete semantics defines a mapping ρ which associates to each control point c in the CFG of a procedure P a set of heaps over $\mathbf{loc} \cup \mathbf{loc}^0$. As usual, the mapping ρ is defined by a system of recursive equations [8, 28] of the form:

$$R_{init} \in \rho(s_P) \text{ and } \rho(c) = \rho(c) \cup \bigcup_{c' \xrightarrow{St} c} \mathtt{post}(St, \rho(c')), \tag{D}$$

where (1) R_{init} is a heap containing two copies of the initial configuration of some procedure $P = (\mathbf{fpi},\mathbf{fpo},\mathbf{loc},G)$, one copy over $\mathbf{loc}^0$, and one copy over $\mathbf{loc}$, (2) $c' \xrightarrow{St} c$ is an edge from the ICFG labeled by St, and (3) `post` is the postcondition operator.

For any statement, except for procedure calls and returns, `post` affects only the part of the heap reachable from nodes labeled by



Universally quantified formula:

$$\begin{aligned}\psi_c : &\ \mathtt{hd}(n_l) \le \mathtt{hd}(n_p) \wedge \mathtt{hd}(n_r) > \mathtt{hd}(n_p)\\ &\wedge \mathtt{len}(n_a) = \mathtt{len}(n_l) + \mathtt{len}(n_r) + \mathtt{len}(n_p)\\ &\wedge \forall y.\ y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \le \mathtt{hd}(n_p)\\ &\wedge \forall y.\ y \in \mathtt{tl}(n_r) \Rightarrow n_r[y] > \mathtt{hd}(n_p)\\ &\wedge d = \mathtt{hd}(n_p) \wedge eq_\forall(n_a, n_a^0) \wedge \mathtt{len}(n_p) = 1,\end{aligned}$$

Multiset formula:

$$\begin{aligned}\psi'_c : &\ \mathtt{ms}(n_a) = \mathtt{ms}(n_l) \cup \mathtt{ms}(n_r) \cup \mathtt{ms}(n_p)\\ &\wedge \mathtt{ms}(n_a^0) = \mathtt{ms}(n_a)\end{aligned}$$

Figure 4. Relations between program configurations.

variables in **loc**. For simplicity, we assume that (1) heaps don't contain garbage, i.e., all the nodes are reachable from nodes labeled by pointer variables, and (2) each pointer assignment `p->next=q`, resp. `p=q`, is preceded by `p->next=NULL`, resp. `p=NULL`. The semantics of the procedure calls and returns is based on *local heaps* [23], i.e. heaps containing only objects reachable from the actual parameters. For example, in Figure 4(a), the local heap for the procedure call `quicksort(left)` contains only the nodes reachable from the node labeled by `left`. As we consider only cutpoint-free programs [24], the definition of this semantics is straightforward.

3. Abstractions of program relations

In this section, we recall the abstract domains defined in [2].

Preliminaries on abstract interpretation: Let $\mathcal{C} = (C, \subseteq)$ and $\mathcal{A} = (A, \sqsubseteq)$ be two lattices ($\subseteq$, resp. $\sqsubseteq$, are order relations on C, resp. A). The lattice $\mathcal{A}$ is an *abstract domain* for $\mathcal{C}$ [7, 10] if there exists a monotonic function $\gamma : A \to C$. In the following, an abstract domain $\mathcal{A}$ is denoted by $\mathcal{A} = (A, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, where $\sqcap$ denotes its greatest lower bound (meet) operator, $\sqcup$ denotes its lowest greater bound (join) operator, $\top$ its top element and $\bot$ its bottom element. Moreover, as usual in the abstract interpretation framework, ∇ represents the widening operator. Let $\mathcal{F}_\mathcal{C}$ be a set of concrete transformers, that is, of functions from C into C. If $\mathcal{A}$ is an abstract domain for $\mathcal{C}$, the set of its abstract transformers, denoted $\mathcal{F}^{\#}_{\mathcal{A}}$, contains a function $f^{\#} : A \to A$ for each $f \in \mathcal{F}_\mathcal{C}$. The transformer $f^{\#}$ is *sound* if $f(\gamma(a)) \subseteq \gamma(f^{\#}(a))$, for any $a \in A$.

3.1 Abstract domains for heaps

The abstraction of a labeled graph H representing (relations between) program configurations is defined starting from a *decomposition* obtained 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 the initial graph, and (3) labeling every node n with a word over $\mathbb{Z}$ which contains the integers on the path from H between n and its successor in the new graph. For example, Figure 4(b) gives a decomposition for the graph in Figure 4(a).

Abstractions of program relations can be defined by representing the values of the integer program variables and the words over $\mathbb{Z}$ as formulas in some logic. The logics we use capture various aspects such as constraints on the sizes, the multisets of their letters, or the data at different positions of the words. For example, the graph and the formula ψ_c in Figure 4(c) represent an abstraction of the program relation in Figure 4(a) obtained starting from the decomposition in Figure 4(b). The same holds for the graph and the multiset formula ψ'_c in Figure 4(c). Such a pair between a graph and a formula is called *abstract heap*. In the formula ψ_c, n_a^0, n_a, n_l, n_r, and n_p are variables interpreted as words over $\mathbb{Z}$, $\mathtt{hd}(n_l)$ denotes the first letter of the word denoted by n_l, $\mathtt{len}(n_a)$ denotes the length of the word denoted by n_a, y is a variable interpreted as a position in some word (as usual, the positions of a word $a_0\, a_1 \ldots a_n$ are the integers from 0 to n), $y \in \mathtt{tl}(n_l)$ means that y belongs to the *tail* of n_l (i.e., the suffix starting with the second letter), and $n_l[y]$ is a term interpreted as the letter at the position y of n_l. The formula $eq_\forall(n_a, n_a^0)$ states that the words denoted by n_a and n_a^0 are identical. We distinguish the first letter of a word from its tail because programs assignments can update only this first letter (the statement `p->data=...` updates the first letter of the word associated to the node labeled by p). In the multiset formula, $\mathtt{ms}(n_a)$ denotes the multiset containing all the letters of n_a.

Let $\mathbb{Z}^+$ denote the set of non-empty sequences over $\mathbb{Z}$ and let *DWVar* be a set of *data word variables* interpreted as elements of $\mathbb{Z}^+$. A *data words logic* is a (possibly infinite) set of formulas from a first order logic $\mathsf{FO}(\mathit{DWVar}, \mathit{DVar}, \mathbb{O}, \mathbb{P})$ over the variables $\mathit{DWVar} \cup \mathit{DVar}$, where $\mathbb{O}$ is a set of operation symbols and $\mathbb{P}$ is a set of predicate symbols. In order to express program transformations we suppose that $\mathbb{O}$ contains at least the concatenation operator $\cdot$, a function symbol $\mathtt{len} : \mathbb{Z}^+ \to \mathbb{Z}$ which returns the length of the input sequence and the equality predicate between words *eq*. However, we do not require that these symbols are used in the set of formulas belonging to some data words logic.

Definition 3.1 (**$\mathbb{LDW}$-domain).** *A* logical data words abstract domain over *DWVar* and *DVar ($\mathbb{LDW}$-domain, for short) is a lattice $\mathcal{A}_\mathbb{W} = \left(A^\mathbb{W}, \sqsubseteq^\mathbb{W}, \sqcap^\mathbb{W}, \sqcup^\mathbb{W}, \top^\mathbb{W}, \bot^\mathbb{W}\right)$, where $A^\mathbb{W}$ is a data words logic, which is an abstract domain for the lattice of sets of pairs (L, D) with $L : \mathit{DWVar} \to \mathbb{Z}^+$ and $D : \mathit{DVar} \to \mathbb{Z}$.*

Notice that $\sqsubseteq^\mathbb{W}$ is a sound approximation of the logical implication $\Rightarrow$ between formulas (i.e., if $\varphi_1 \sqsubseteq^\mathbb{W} \varphi_2$ then $\varphi_1 \Rightarrow \varphi_2$).

Definition 3.2 (**Abstract heap).** *An* abstract heap over *PVar*, *DVar*, and an $\mathbb{LDW}$-domain $\mathcal{A}_\mathbb{W}$ *is a tuple $\widetilde{H} = (N, S, V, \widetilde{W})$ where N, S, V are as in the definition of heaps, and $\widetilde{W}$ is a formula in $\mathcal{A}_\mathbb{W}$ over the data word variables $N \setminus \{\sharp\}$ (we assume that for each node in N there exists a data word variable with the same name) and the data variables DVar. A k*-abstract heap *is an abstract heap with at most k simple nodes.*

Two abstract heaps are *isomorphic* if their underlying graphs are isomorphic. Let $\mathcal{C}_\mathbb{H}$ denote the lattice of sets of heaps. We define $\mathcal{A}_\mathbb{H}(k, \mathcal{A}_\mathbb{W})$ an abstract domain for $\mathcal{C}_\mathbb{H}$ whose elements are k-abstract heaps over $\mathcal{A}_\mathbb{W}$ s.t. (1) for any two isomorphic abstract heaps, the lattice operators are obtained by applying the corresponding operators between the values from $\mathcal{A}_\mathbb{W}$, and (2) the join and the widening (resp. meet) of two non-isomorphic abstract heaps is $\top^\mathbb{H}$ (resp. $\bot^\mathbb{H}$). Notice that $\nabla^\mathbb{H}$ is a widening operator because the heaps generated by the programs which manipulate only singly-linked lists contain a bounded number of crucial nodes [19].

The domains used in the analysis are finite powerset domains over $\mathcal{A}_\mathbb{H}(k, \mathcal{A}_\mathbb{W})$. Their elements are called *k-abstract heap sets.*

Definition 3.3 (**Abstract heap set).** *A k*-abstract heap set over *PVar*, *DVar*, and an $\mathbb{LDW}$-domain $\mathcal{A}_\mathbb{W}$ *is a finite set of non-*

isomorphic k-abstract heaps over PVar, DVar, and $\mathcal{A}_{\mathbb{W}}$. The abstract domain of k-abstract heap sets is denoted by $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{W}}) = \left(A^{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{W}}), \sqsubseteq^{\mathbb{HS}}, \sqcap^{\mathbb{HS}}, \sqcup^{\mathbb{HS}}, \top^{\mathbb{HS}}, \bot^{\mathbb{HS}}\right)$, where $A^{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{W}})$ is the set of all k-abstract heap sets over PVar, DVar, and $\mathcal{A}_{\mathbb{W}}$.

The operators from $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{W}})$ and the widening operator are obtained from those of $\mathcal{A}_{\mathbb{H}}(k,\mathcal{A}_{\mathbb{W}})$ as usual [9]. For example, the join of two abstract heap sets is computed by taking the union of these sets and by applying the join operator between any two isomorphic abstract heaps.

3.2 An $\mathbb{LDW}$-domain with universally quantified formulas

We present the abstract domain $\mathcal{A}_{\mathbb{U}} = \left(A^{\mathbb{U}}, \sqsubseteq^{\mathbb{U}}, \sqcap^{\mathbb{U}}, \sqcup^{\mathbb{U}}, \top^{\mathbb{U}}, \bot^{\mathbb{U}}\right)$ whose elements are first-order formulas with free variables in *DWVar* $\cup$ *DVar*. This domain is parametrized by a set of constraints on position variables (which are interpreted as positions in the words), called *guard patterns*. Guard patterns are conjunctions of:

- formulas that associate vectors of position variables with data word variables ($\mathbf{y} \in \mathtt{tl}(\omega)$ means that the position variables from the vector $\mathbf{y}$ are interpreted as positions in the tail of ω),
- formulas that impose a total order between the values of the position variables associated with the same data word variable ($y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m$, where $\prec_i \in \{\leq, <, <_1\}$ with $y <_1 y'$ iff $y' = y+1$), and
- a linear constraint on the values of the position variables which are the first in the order constraints considered above.

Let $\mathcal{V} \subseteq DWVar$ and let $\mathcal{P}$ be a set of guard patterns. Also, let $\mathcal{P}(\mathcal{V})$ be a set of formulas obtained from guard patterns in $\mathcal{P}$ by substituting all data word variables with variables from $\mathcal{V}$. An element of $A_{\mathbb{U}}$ is a formula of the form:

$$\widetilde{W}(\mathcal{V}) ::= E(\mathcal{V}) \wedge \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V})} \forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U_g \qquad \text{(E)}$$

where $E(\mathcal{V})$ is a quantifier-free arithmetical formula over *DVar* and terms $\mathtt{hd}(w)$, $\mathtt{len}(w)$ with $w \in \mathcal{V}$, $g(\mathbf{y})$ is a guard over the vector of position variables $\mathbf{y}$, and U_g is a quantifier-free arithmetical formula over the terms in $E(\mathcal{V})$ together with $w[y]$ and y, for any $w \in \mathcal{V}$ and $y \in \mathbf{y}$. It is assumed that the term $w[y]$ appears in U_g only if $g(\mathbf{y})$ associates y with w. Also, E and U_g represent elements of some numerical abstract domain $\mathcal{A}_{\mathbb{Z}} = \left(A^{\mathbb{Z}}, \sqsubseteq^{\mathbb{Z}}, \sqcap^{\mathbb{Z}}, \sqcup^{\mathbb{Z}}, \top^{\mathbb{Z}}, \bot^{\mathbb{Z}}\right)$ which is a parameter of $\mathcal{A}_{\mathbb{U}}$.

Lattice operators: The value $\top^{\mathbb{U}}$ (resp. $\bot^{\mathbb{U}}$) is defined by the formula in which E and all U_g are $\top^{\mathbb{Z}}$ (resp. $\bot^{\mathbb{Z}}$). Let

$$\widetilde{W}(\mathcal{V}_1) = E(\mathcal{V}_1) \wedge \textstyle\bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1)} \forall \mathbf{y}.\, (g(\mathbf{y}) \Rightarrow U_g) \text{ and}$$
$$\widetilde{W}'(\mathcal{V}_2) = E'(\mathcal{V}_2) \wedge \textstyle\bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_2)} \forall \mathbf{y}.\, (g(\mathbf{y}) \Rightarrow U'_g).$$

Before applying any lattice operator we add to $\widetilde{W}$ (resp. $\widetilde{W}'$) universally quantified formulas $\forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow \top^{\mathbb{Z}}$, for any $g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1) \setminus \mathcal{P}(\mathcal{V}_2)$ (resp. $g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_2) \setminus \mathcal{P}(\mathcal{V}_1)$).

Then, $\widetilde{W} \sqsubseteq^{\mathbb{U}} \widetilde{W}'$ iff (1) $E \sqsubseteq^{\mathbb{Z}} E'$, and (2) for each guard $g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1) \cup \mathcal{P}(\mathcal{V}_2)$ which associates the vector of position variables $\mathbf{y_i}$ with the data word variable w_i, for all $1 \leq i \leq n$, the following holds: if $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(w_i) \geq |\mathbf{y}_i| + 1$, for all $1 \leq i \leq n$, then $E \sqcap^{\mathbb{Z}} U_g \sqsubseteq^{\mathbb{Z}} U'_g$.

Also, $\widetilde{W} \sqcup^{\mathbb{U}} \widetilde{W}'$ is defined by

$$\left(E \sqcup^{\mathbb{Z}} E'\right) \wedge \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1) \cup \mathcal{P}(\mathcal{V}_2)} \forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow \left(U_g \sqcup^{\mathbb{Z}} U'_g\right).$$

The operators $\sqcap^{\mathbb{U}}$ and $\nabla^{\mathbb{U}}$ are defined in a similar way.

3.3 An $\mathbb{LDW}$-domain with multiset formulas

We present the abstract domain $\mathcal{A}_{\mathbb{M}} = \left(A^{\mathbb{M}}, \sqsubseteq^{\mathbb{M}}, \sqcap^{\mathbb{M}}, \sqcup^{\mathbb{M}}, \top^{\mathbb{M}}, \bot^{\mathbb{M}}\right)$ whose elements are multiset constraints with free variables in *DWVar* $\cup$ *DVar*. A *basic multiset term* is $u ::= \mathtt{mhd}(n) \mid \mathtt{mtl}(n) \mid d$, where $n \in DWVar$ and $d \in DVar$. The term $\mathtt{mhd}(n)$ (resp. d) represents the singleton containing the first letter of the word associated to n (resp. the value of d). The term $\mathtt{mtl}(n)$ represents the multiset containing all the letters of the word associated to n except for the first one. As a shorthand, $\mathtt{ms}(n)$ denotes the term $\mathtt{mhd}(n) \cup \mathtt{mtl}(n)$.

A *multiset constraint* is a conjunction of formulas of the form $u_1 \cup u_2 \cup \cdots \cup u_s = v_1 \cup v_2 \cup \cdots \cup v_t$, where $s \geq 1$, $t \geq 1$, and for all $1 \leq i \leq s$ and $1 \leq j \leq t$, u_i and v_j are pairwise distinct basic multiset terms, and $\cup$ is the usual union operator between multisets.

Multiset constraints can be represented by a polyhedron with a dimension for any basic term: a formula $u_1 \cup u_2 \cup \cdots \cup u_s = v_1 \cup v_2 \cup \cdots \cup v_t$ is represented by the linear constraint $u_1 + u_2 + \cdots + u_s = v_1 + v_2 + \cdots + v_t$. The entailment relation between the multiset constraints is defined by the entailment relation between the corresponding polyhedra. The lattice of multiset constraints is finite (for finite *DWVar* and *DVar*) and consequently, there is no need for a widening operator.

4. Abstract semantics

The abstract semantics defines a mapping $\rho^{\#}$, which associates to each control point of the program an abstract heap set. It is defined by the system of equations obtained from the one defining the concrete semantics in (D), by replacing `post` with a *sound abstract transformer* $\mathtt{post}^{\#}$ over abstract heap sets. The termination of the fixpoint computation is guaranteed by applying the widening operator at the start of each intra-procedural loop and at the entry and exit point of each recursive procedure.

The analysis that we have implemented tabulates the input configurations of the computed procedure summaries. In this way, we avoid to compute twice the same procedure summary. For any label *St* of some ICFG edge, we define $\mathtt{post}^{\#}$ as follows:

- if *St* is different from `assume`, `assert`, procedure call and procedure return, we first define the abstract transformer $\mathtt{post}^{\#}$ over abstract heaps. Then, $\mathtt{post}^{\#}(St, HS)$, where HS is some abstract heap set, is the join in $\mathcal{A}_{\mathbb{HS}}$ of $\mathtt{post}^{\#}(St, \tilde{H})$, for each $\tilde{H} \in HS$;
- otherwise, we define $\mathtt{post}^{\#}$ directly over abstract heap sets.

The statement** p=`NULL`**: The abstract transformer corresponding to this statement begins by moving the label p to the node $\sharp$. It may happen that the obtained abstract heap contains more than k simple nodes. If it is the case, the transformer calls a procedure called $\mathtt{fold}^{\#}$ which, for any $\tilde{H} \in \mathcal{A}_{\mathbb{H}}$, returns an abstract heap $\tilde{H}'$ with no simple nodes which over-approximates $\tilde{H}$ (i.e., the concretization of $\tilde{H}'$ includes the concretization of $\tilde{H}$).

Let $\tilde{H} = (N, S, V, \widetilde{W})$ be an abstract heap and let $n_1, \ldots, n_t$ be all its simple nodes. For simplicity, suppose that they are on the same path between two crucial nodes n and m. We define $\mathtt{fold}^{\#}(\tilde{H}) = (N', S', V, \widetilde{W}')$, where (N', S') are obtained by removing the nodes $n_1, \ldots, n_t$ and by adding the edge (n, m) and $\widetilde{W}'$ is a formula in $\mathcal{A}_{\mathbb{W}}$ which is an over-approximation of the following formula

$$\left(\exists n \exists n_1 \ldots \exists n_t.\, \left(\overline{n} = n \cdot n_1 \cdot \ldots \cdot n_t \wedge \widetilde{W}\right)\right)[\overline{n} \leftarrow n]. \qquad \text{(F)}$$

The formula in (F) belongs to $\mathcal{A}_{\mathbb{W}}$ extended with quantification over data word variables, and a concatenation operation symbol "$\cdot$". Intuitively, (F) says that the word associated to n in $\mathtt{fold}^{\#}(\tilde{H})$ (denoted in (F) by $\overline{n}$) is the concatenation of the words associated to $n, n_1, \ldots, n_t$ in $\tilde{H}$.

The transformer $\mathtt{concat}^{\#}$ from [2] computes an over-approximation of (F) in $\mathcal{A}_{\mathbb{U}}$. It is precise for domains $\mathcal{A}_{\mathbb{U}}$ parametrized by *closed sets of patterns*. Roughly, a set of patterns $\mathcal{P}$

is closed if whenever it contains a pattern P over the set of position variables $\mathbf{y}$ it also contains all the projections of P (in Presburger arithmetics) on subsets of $\mathbf{y}$. In $\mathcal{A}_{\mathbb{M}}$, an over-approximation of (F) can be obtained by:

- updating in $\widetilde{W}$, $\mathtt{mtl}(n_1)$ to
$$\mathtt{mtl}(n_1) \cup \mathtt{mhd}(n_2) \cup \mathtt{mtl}(n_2) \cup \cdots \cup \mathtt{mhd}(n_t) \cup \mathtt{mtl}(n_t)$$
- removing all the constraints on the variables $n_2, \ldots, n_t$.

Statements involving reference fields: For a statement St of the form p`->next=NULL` (resp. q`=`p`->next`), the concrete `post` modifies the edge (resp. the target of the edge) starting in the node labeled by p. The abstract transformer $\mathtt{post}^{\#}(St, \tilde{H})$, where $\tilde{H} = (N, S, V, \widetilde{W})$, can not directly modify this edge because it may correspond to a path of arbitrary length in the concrete heap. Thus, $\mathtt{post}^{\#}(St, \tilde{H})$ begins by calling a procedure $\mathtt{unfold}^{\#}(\tilde{H}, p)$ which returns two abstract heaps $\tilde{H}_1$ and $\tilde{H}_2$ such that the concretization of $\tilde{H}$ is included in the union of the concretizations of $\tilde{H}_1$ and $\tilde{H}_2$. The abstract heap $\tilde{H}_1$ (resp. $\tilde{H}_2$) corresponds to the case when the length of the word associated to the node n labeled by p in $\tilde{H}$ is 1 (resp. strictly greater than 1). Thus, $\tilde{H}_1 = (N, S, V, \widetilde{W}')$ where $\widetilde{W}'$ is a formula in $\mathcal{A}_{\mathbb{W}}$ which over-approximates $\varphi_1 : \widetilde{W} \wedge \mathtt{len}(n) = 1$ (i.e., $\varphi_1 \Rightarrow \widetilde{W}'$). Then, $\tilde{H}_2 = (N', S', V, \widetilde{W}'')$ corresponds to the case $\mathtt{len}(n) > 1$, where (N', S') is obtained by adding a new node n' between n and its successor n'' in $\tilde{H}$ and $\widetilde{W}''$ is a formula in $\mathcal{A}_{\mathbb{W}}$ that over-approximates the following formula (given in an extension of the logic of $\mathcal{A}_{\mathbb{W}}$ with quantification over word variables and with a concatenation operation symbol "$\cdot$"):

$$\left(\exists n.\ \left(n = \overline{n} \cdot n' \wedge \mathtt{len}(\overline{n}) = 1 \wedge \widetilde{W} \wedge \mathtt{len}(n) > 1\right)\right) [\overline{n} \leftarrow n] \quad \text{(G)}$$

This formula says that the word associated to n in $\tilde{H}_2$ is only the first letter of the word associated to n in $\tilde{H}$ and the word associated to n' in $\tilde{H}_2$ is the tail of the word associated to n in $\tilde{H}$. The abstract transformer $\mathtt{split}^{\#}$ given in [2] computes this over-approximation in the domain of universal formulas. If $\widetilde{W}$ is a multiset constraint in $\mathcal{A}_{\mathbb{M}}$, an over-approximation for (G) can be obtained by substituting $\mathtt{mhd}(n)$ with $\mathtt{mhd}(n) \cup \mathtt{mhd}(n') \cup \mathtt{mtl}(n')$.

The statements `assume` and `assert`: The assertions φ we consider for these statements are disjunctions of formulas of the form $\varphi_G \wedge \varphi_{\mathbb{W}}$, where φ_G describes a unique graph (N, S, V) as in the definition of heaps and $\varphi_{\mathbb{W}}$ is a formula in the data words logic of $\mathcal{A}_{\mathbb{W}}$ over the data word variables N. For simplicity, we assume that φ_G is a separation logic formula [22] defined as a separating conjunction of atomic formulas of the form $\mathtt{ls}(n, n')$, where n, n' are variables interpreted as distinct nodes in the graph. A formula $\mathtt{ls}(n, n')$ is true in some abstract heap iff there exists an edge between the nodes represented by n and n' (i.e a path connecting them in the concrete heap). We require that φ_G (1) contain at most one atomic formula $\mathtt{ls}(n, n')$, for every n, and (2) induce a graph that does not contain simple nodes. Then, let A_H be an abstract heap set from $\mathcal{A}_{\mathbb{H}}(\mathcal{A}_{\mathbb{W}})$ and let φ be a formula $\varphi_G \wedge \varphi_{\mathbb{W}}$ that describes an element A'_H from the same abstract domain $\mathcal{A}_{\mathbb{H}}(\mathcal{A}_{\mathbb{W}})$. Thus, $\mathtt{post}^{\#}(\mathtt{assume}\ \varphi, A_H) = A_H \sqcap^{\mathbb{HS}} A'_H$. Also, $\mathtt{post}^{\#}(\mathtt{assert}\ \varphi, A_H) = A_H$ if $\mathtt{fold}^{\#}(A_H) \sqsubseteq^{\mathbb{HS}} A'_H$ and $\mathtt{post}^{\#}(\mathtt{assert}\ \varphi, A_H) = \perp^{\mathbb{HS}}$, otherwise. The extension of this definition to disjunctions of such formulas is straightforward.

Procedure calls: The abstract transformer corresponding to a *call to start* edge, **call** $\mathbf{y} = P(\mathbf{x})$, where $P = (\mathbf{fpi}, \mathbf{fpo}, \mathbf{loc}, G)$, computes abstract heap sets over $\mathbf{loc}^0 \cup \mathbf{loc}$. For any abstract heap $\tilde{R}^c = (N^c, S^c, V^c, \widetilde{W}^c)$ from the input abstract heap set it returns an abstract heap $\tilde{R}^s = (N^s, S^s, V^s, \widetilde{W}^s)$, where:

- the local heap corresponding to the call $P(\mathbf{x})$, denoted $\mathtt{local}(\tilde{R}^c, P(\mathbf{x})) = (N^{\mathbf{x}}, S^{\mathbf{x}}, V^{\mathbf{x}}, \widetilde{W}^x)$, is defined by (1) $(N^{\mathbf{x}}, S^{\mathbf{x}}, V^{\mathbf{x}})$ is the sub-graph of $\tilde{R}^c$ containing only nodes reachable from the nodes labeled by variables in $\mathbf{x}$ in which actual parameters are replaced by the corresponding formal parameters and (2) $\widetilde{W}^{\mathbf{x}}$ is an over-approximation of $\exists (N^c \setminus N^{\mathbf{x}}).\ \widetilde{W}^c$(where $\exists (N^c \setminus N^{\mathbf{x}})$ denotes an existential quantification over all the word variables in $N^c \setminus N^{\mathbf{x}}$).
- the graph (N^s, S^s, V^s) contains two copies of $(N^{\mathbf{x}}, S^{\mathbf{x}}, V^{\mathbf{x}})$ (the nodes of one copy are labeled by variables in $\mathbf{loc}^0$) and $\widetilde{W}^s$ is an over-approximation in $\mathcal{A}_{\mathbb{W}}$ of the formula:
$$\widetilde{W}^{\mathbf{x}} \wedge \widetilde{W}^{\mathbf{x}} \left[n \leftarrow n^0 \mid n \in N^{\mathbf{x}}\right] \wedge \bigwedge_{\substack{n^0 \in N^0,\ n \in N \\ n \text{ is a copy of } n^0}} eq(n, n^0),$$
where $eq(n, n^0)$ is expressing the equality between the data of the words denoted by n and n^0. In $\mathcal{A}_{\mathbb{U}}$, $eq(n, n^0)$ is expressed precisely by
$$\begin{aligned} eq_{\forall}(n, n^0) :\ & \mathtt{hd}(n) = \mathtt{hd}(n^0) \wedge \mathtt{len}(n) = \mathtt{len}(n^0) \\ & \wedge \big(\forall y_1, y_2.\ (y_1 \in \mathtt{tl}(n^0) \wedge y_2 \in \mathtt{tl}(n) \wedge y_1 = y_2) \\ & \qquad \Rightarrow n^0[y_1] = n[y_2]\big), \end{aligned} \quad \text{(H)}$$
while $\mathcal{A}_{\mathbb{M}}$ can represent only an over-approximation by
$$eq_m(n, n^0) :\ \mathtt{mhd}(n) = \mathtt{mhd}(n^0) \wedge \mathtt{mtl}(n) = \mathtt{mtl}(n^0). \quad \text{(I)}$$

Procedure returns: Consider now an *exit to return* edge, labeled by **ret** $\mathbf{y} = P(\mathbf{x})$, from the exit point of $P = (\mathbf{fpi}, \mathbf{fpo}, \mathbf{loc}, G)$, denoted e_P, to some control point r in the CFG of a procedure Q. Let c be the call point associated to r and $\tilde{R}^c = (N^c, S^c, V^c, \widetilde{W}^c) \in \rho^{\#}(c)$. The abstract transformer for **ret** $\mathbf{y} = P(\mathbf{x})$ computes an abstract heap set whose elements are denoted by $\tilde{R}^r = (N^r, S^r, V^r, \widetilde{W}^r)$. To this, it uses the (pre) computed summary for this call of the procedure P, represented by an abstract heap set whose elements are denoted $\tilde{R}^e = (N^e, S^e, V^e, \widetilde{W}^e)$. The summary must satisfy the property that its input configuration R^e_{in} is implied by the constraints on the actual parameters, i.e. $\mathtt{local}(\tilde{R}^c, P(\mathbf{x})) \sqsubseteq^{\mathbb{H}} \tilde{R}^e_{in}$. To compute $\tilde{R}^r = (N^r, S^r, V^r, \widetilde{W}^r)$:

- we take the union between the call graph (N^e, S^e, V^e) and the exit graph $(N^{out}, S^{out}, V^{out})$ of the output configuration in the summary R^e,
- we eliminate the input version of the parameters of P,
- we define $\widetilde{W}^r$ by (a) defining $\mathtt{Combine}(\widetilde{W}^c, \widetilde{W}^e)$ which applies the conjunction between $\widetilde{W}^c$ and $\widetilde{W}^e$ after it ensures that the nodes in N^c belonging to the local heap of this call have the same name as the corresponding nodes in N^e representing the input configuration and (b) computing an over-approximation of $\exists N^0.\ \mathtt{Combine}(\widetilde{W}^c, \widetilde{W}^e)$, where N^0 is the set of nodes in N^e representing the input configuration.

In $\mathcal{A}_{\mathbb{U}}$ and $\mathcal{A}_{\mathbb{M}}$, over-approximations of existentially quantified formulas can be defined by, roughly, removing sub-formulas containing the existentially quantified data word variables.

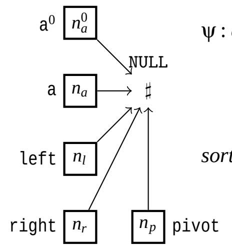
5. Combining abstract domains

Let us take a closer look to the analysis of the `quicksort` procedure from Figure 1 with the domain of universally-quantified formulas, $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$, over the set of guard patterns

$$\begin{gathered} y \in \mathtt{tl}(\omega), \qquad y_1, y_2 \in \mathtt{tl}(\omega) \wedge y_1 \leq y_2, \\ y_1 \in \mathtt{tl}(\omega_1) \wedge y_2 \in \mathtt{tl}(\omega_2) \wedge y_1 = y_2. \end{gathered}$$

The analysis manipulates universally-quantified implications where the left part is one of the formulas above, where the ω's are data word variables. Typically, in the case of recursive procedures, the analysis starts by computing procedure summaries for input lists of length 1 and then, for input lists of length 2, and so on until it reaches a fixpoint (to terminate it applies the widening operator).

The analysis is able to compute a procedure summary $\psi^{\mathbb{U}}_{sum}$: $sorted(n'_o) \wedge \mathtt{len}(n_l) \leq 2$ for input lists of length at most 2, where n'_o represents the list pointed to by the output formal parameter `res` of `quicksort`. In the next iteration, the context of the first recursive call "`left=quicksort(left)`" is given by the graph and the formula ψ_c in Figure 4(c). The abstract transformer for procedure return computes $\exists n_l.\ (\psi_c \wedge \psi^{\mathbb{U}}_{sum})$. This projection will remove all the constraints on n_l which is the actual input parameter. The same holds for the list pointed to by `right` and the second recursive call. The relation obtained after the two calls is given in Figure 5. We have lost the property that all the elements of `left` are less than the pivot. Consequently, after calling `concat`, we can't obtain that the list pointed to by `res` is sorted.



$$\psi : d = \mathtt{hd}(n_p) \wedge \mathtt{len}(n_p) = 1 \wedge eq_{\forall}(n_a, n_a^0) \wedge sorted(n_l) \wedge \mathtt{len}(n_l) \leq 3 \wedge sorted(n_r) \wedge \mathtt{len}(n_r) \leq 3,$$

$$sorted(n) : \forall y.\ y \in \mathtt{tl}(n) \Rightarrow \mathtt{hd}(n) \leq n[y] \wedge \forall y_1, y_2.\ ((y_1, y_2) \in \mathtt{tl}(n) \wedge y_1 \leq y_2) \Rightarrow n[y_1] \leq n[y_2]$$

Figure 5. The relation synthesized at line 17 of `quicksort`.

Next, we give a solution for this problem based on a combination of abstract analyses. Thus, at the return from the first recursive call, the abstract transformer computes

$$\exists n_l.\ \left(\mathtt{strengthen}_{\mathbb{M}}\left(\psi_c \wedge \psi^{\mathbb{U}}_{sum}, \psi^{\mathbb{M}}_{sum}\right)\right), \qquad \text{(J)}$$

where $\psi^{\mathbb{M}}_{sum} : \mathtt{ms}(n'_o) = \mathtt{ms}(n_l)$ is the summary for `quicksort` computed in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$ and $\mathtt{strengthen}_{\mathbb{M}} : \mathcal{A}_{\mathbb{U}} \times \mathcal{A}_{\mathbb{M}} \to \mathcal{A}_{\mathbb{U}}$. The operator $\mathtt{strengthen}_{\mathbb{M}}$ returns

$$\psi_c \wedge \psi^{\mathbb{U}}_{sum} \wedge \mathtt{hd}(n'_o) \leq \mathtt{hd}(n_p) \wedge \forall y.\ y \in \mathtt{tl}(n'_o) \Rightarrow n'_o[y] \leq \mathtt{hd}(n_p),$$

and consequently, by eliminating the constraints on n_l (due to the existential quantification), we preserve the fact that the elements of the list pointed to by `left` are smaller than the pivot.

The function $\mathtt{strengthen}_{\mathbb{M}}$ is an instance of a more general procedure denoted $\mathtt{strengthen}_{\mathbb{W}}$.

Given $\mathcal{A}_{\mathbb{W}}$ an $\mathbb{LDW}$-domain, the definition of $\mathtt{strengthen}_{\mathbb{W}}$ uses a procedure $\mathtt{infer}_{\mathbb{W}}$ which, for any two abstract values $\widetilde{W}_{in} \in \mathcal{A}_{\mathbb{U}}$ and $\widetilde{W}_{aux} \in \mathcal{A}_{\mathbb{W}}$, returns an over-approximation of their conjunction expressed in $\mathcal{A}_{\mathbb{U}}$.

DEFINITION 5.1. *Let $\mathcal{A}_{\mathbb{W}}$ be some $\mathbb{LDW}$-domain and $\mathtt{infer}_{\mathbb{W}} : \mathcal{A}_{\mathbb{U}} \times \mathcal{A}_{\mathbb{W}} \to \mathcal{A}_{\mathbb{U}}$ such that*

$$\left(\widetilde{W}_{in} \wedge \widetilde{W}_{aux}\right) \Rightarrow \mathtt{infer}_{\mathbb{W}}(\widetilde{W}_{in}, \widetilde{W}_{aux}).$$

The function $\mathtt{strengthen}_{\mathbb{W}} : \mathcal{A}_{\mathbb{U}} \times \mathcal{A}_{\mathbb{W}} \to \mathcal{A}_{\mathbb{U}}$ is defined by

$$\mathtt{strengthen}_{\mathbb{W}}(\widetilde{W}_{in}, \widetilde{W}_{aux}) = \widetilde{W}_{in} \sqcap^{\mathbb{U}} \mathtt{infer}_{\mathbb{W}}(\widetilde{W}_{in}, \widetilde{W}_{aux}).$$

Before giving the formal definition of $\mathtt{infer}_{\mathbb{W}}$ let us consider two examples, one for computing $\mathtt{infer}_{\mathbb{M}}$ and one for computing $\mathtt{infer}_{\mathbb{U}}$.

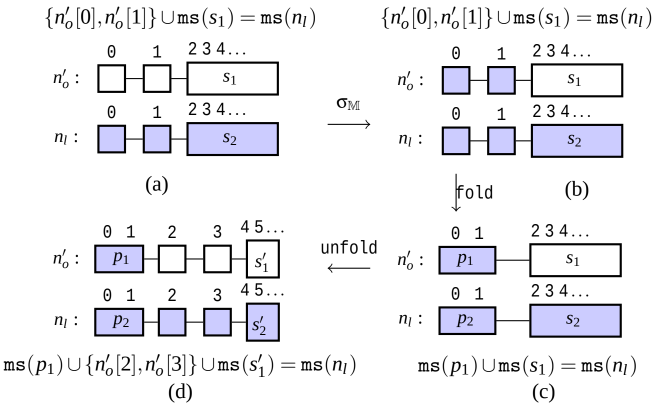
Computing* $\mathtt{infer}_{\mathbb{M}}$*: Consider the application of $\mathtt{strengthen}_{\mathbb{M}}$ from (J). From its arguments, we should only remember the multiset constraint $\mathtt{ms}(n'_o) = \mathtt{ms}(n_l)$ and the following formula:

$$\varphi(n_l) : \mathtt{hd}(n_l) \leq \mathtt{hd}(n_p) \wedge \forall y.\ y \in \mathtt{tl}(n_l) \Rightarrow n_l[y] \leq \mathtt{hd}(n_p) \qquad \text{(K)}$$

which says that all the elements of the word n_l are less than $\mathtt{hd}(n_p)$.

The result of $\mathtt{strengthen}_{\mathbb{M}}$ is computed using $\mathtt{infer}_{\mathbb{M}}$ applied on the same inputs. Roughly, to compute $\mathtt{infer}_{\mathbb{M}}$ (1) we unfold a bounded length prefix p_1, resp. p_2, of the word n'_o, resp. n_l; thus, $n'_o = p_1 \cdot s_1$ and $n_l = p_2 \cdot s_2$, where $\cdot$ denotes the concatenation of words, (2) we infer the properties of the nodes in p_1 and p_2 implied by the conjunction between the constraint in the domain of universally-quantified formulas and the constraint in the multiset domain and (3) we fold the prefixes p_1 and p_2 and collect the informations on these nodes using a universally-quantified formula. Then, we continue to apply the same transformations on the words s_1 and s_2 until we reach a fixpoint.

This unfolding/folding mechanism reduces the initial problem (of inferring universally-quantified constraints implied by the conjunction of the inputs) to the problem of inferring *quantifier-free* constraints implied by the conjunction of the inputs.



$\{n'_o[0], n'_o[1]\} \cup \mathtt{ms}(s_1) = \mathtt{ms}(n_l)$
0 1 2 3 4...
n'_o :
s_1
0 1 2 3 4...
n_l :
s_2
(a)
$\sigma_{\mathbb{M}}$
$\{n'_o[0], n'_o[1]\} \cup \mathtt{ms}(s_1) = \mathtt{ms}(n_l)$
(b)
fold
0 1 2 3 4...
p_1
p_2
$\mathtt{ms}(p_1) \cup \mathtt{ms}(s_1) = \mathtt{ms}(n_l)$
(c)
unfold
0 1 2 3 4 5...
s'_1
s'_2
$\mathtt{ms}(p_1) \cup \{n'_o[2], n'_o[3]\} \cup \mathtt{ms}(s'_1) = \mathtt{ms}(n_l)$
(d)

Figure 6. Computing $\mathtt{infer}_{\mathbb{M}}$.

Some steps from the computation of $\mathtt{infer}_{\mathbb{M}}$ are given in Figure 6. The unfolding of a prefix of length 2 is given in Figure 6(a). Above each sub-word we give the positions from the initial word it contains. The sub-words are colored if their elements satisfy the property from $\varphi(n_l)$. At this step, only the sub-words of n_l are colored. In the multiset constraint, for any $i \geq 0$, $\{n'_o[i]\}$ denotes the singleton multiset containing the element of n'_o at position i. This syntax is not exactly the one used in the formulas manipulated by our analysis but we use it for the sake of simplicity. On this unfolding we apply a *partial reduction operator* [9], denoted $\sigma_{\mathbb{M}}$, which deduces new properties on the unfolded prefix based on the multiset constraints. Here, it deduces that, for any $0 \leq i \leq 1$,

$$n'_o[i] \in \mathtt{ms}(n_l) \wedge \varphi(n_l) \text{ implies } n'_o[i] \leq \mathtt{hd}(n_p).$$

The result of applying $\sigma_{\mathbb{M}}$ is given in Figure 6(b). Now, we can apply a folding operation in the two abstract domains whose result is given in Figure 6(c). Then, we continue by unfolding another prefix of length 2 from the sub-word of n_o that starts with position 2 (this is pictured in Figure 6(d)). We repeat these steps until the fixpoint computation terminates. The result will be:

$$\mathtt{infer}_{\mathbb{M}}\left(\psi_c \wedge \psi^{\mathbb{U}}_{sum}, \psi^{\mathbb{M}}_{sum}\right) = \psi_c \wedge \psi^{\mathbb{U}}_{sum} \wedge \varphi(n'_o).$$

Computing* $\mathtt{infer}_{\mathbb{U}}$*: Let $\mathcal{A}^1_{\mathbb{U}}$, resp. $\mathcal{A}^2_{\mathbb{U}}$, be a domain with universal formulas parametrized by a set of patterns $\mathcal{P}_1$, resp. $\mathcal{P}_2$. Using a similar mechanism, we compute, for any $\widetilde{W} \in \mathcal{A}^1_{\mathbb{U}}$, an over-approximation of $\widetilde{W}$ in $\mathcal{A}^2_{\mathbb{U}}$, denoted $\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2)(\widetilde{W})$. For example, let

$$\begin{array}{rcl}\mathcal{P}_1 & = & \{y \in \mathtt{tl}(\omega), \qquad y_1,y_2 \in \mathtt{tl}(\omega) \wedge y_1 \leq y_2\} \\ \mathcal{P}_2 & = & \{y \in \mathtt{tl}(\omega) \wedge y = 1, \qquad y \in \mathtt{tl}(\omega) \wedge y = \mathtt{len}(\omega) - 1, \\ & & y_1,y_2 \in \mathtt{tl}(\omega) \wedge y_1 <_1 y_2\},\end{array}$$

and let $\widetilde{W}$ be the sortedness property $sorted(n)$ in Figure 5. Computing $\mathtt{convert}(\mathcal{P}_1,\mathcal{P}_2)(\widetilde{W})$ allows us to prove that $\widetilde{W}$ implies

$$\gamma(n) : \forall y_1,y_2.\ (y_1,y_2 \in \mathtt{tl}(n) \wedge y_1 <_1 y_2) \Rightarrow n[y_1] \leq n[y_2]$$

which can not be done using the entailment relation $\sqsubseteq^{\mathbb{U}}$ defined in $\mathcal{A}^1_{\mathbb{U}}$ or $\mathcal{A}^2_{\mathbb{U}}$. We start with the formula $\widetilde{W}' : true$ and we apply a procedure $\mathtt{infer}_{\mathbb{U}}$ to strengthen it using the information from $\widetilde{W}$. Again, we unfold a bounded-length prefix of n. For instance, if we unfold a prefix of length 3, the formula $\widetilde{W}'$ remains unchanged and the formula $\widetilde{W}$ is transformed into a formula $\widetilde{W}_1$ given by:

$$n[0] \leq n[1] \wedge n[1] \leq n[2] \wedge n[2] \leq n[3] \wedge sorted(n[3]),$$

where $sorted(n[3])$ denotes the fact that the formula $sorted$ is true starting from the third position of n. Then, we apply a partial reduction operator, $\sigma_{\mathbb{U}}$, that takes the existential part of $\widetilde{W}_1$ and adds it to $\widetilde{W}'$. Afterwards, we apply a folding operation and the word n is split into a word n' containing only its first 3 elements and a word n'' containing all the other elements. The formula in $\mathcal{A}^2_{\mathbb{U}}$ becomes

$$\begin{array}{c}\forall y.\ (y \in \mathtt{tl}(n') \wedge y = 1) \Rightarrow \mathtt{hd}(n') \leq y \\ \wedge \forall y_1,y_2.\ (y_1,y_2 \in \mathtt{tl}(n') \wedge y_1 <_1 y_2) \Rightarrow n'[y_1] \leq n'[y_2] \\ \wedge \forall y.\ (y \in \mathtt{tl}(n') \wedge y = \mathtt{len}(n') - 1) \Rightarrow y \leq \mathtt{hd}(n'').\end{array}$$

As in the previous case, by iterating these two steps into a fixpoint computation that traverses all the elements of n, we obtain $\mathtt{convert}(\mathcal{P}_1,\mathcal{P}_2)(\widetilde{W})$ which is given by

$$\begin{array}{c}\forall y.\ (y \in \mathtt{tl}(n) \wedge y = 1) \Rightarrow \mathtt{hd}(n) \leq y \\ \wedge \forall y_1,y_2.\ (y_1,y_2 \in \mathtt{tl}(n) \wedge y_1 <_1 y_2) \Rightarrow n[y_1] \leq n[y_2].\end{array}$$

Clearly, $\mathtt{convert}(\mathcal{P}_1,\mathcal{P}_2)(\widetilde{W}) \sqsubseteq^{\mathbb{U}} \gamma(n)$ which finishes our proof.

5.1 The procedure $\mathtt{infer}_{\mathbb{W}}$

The output of $\mathtt{infer}_{\mathbb{W}}$ is defined by the analysis of a program without procedures with an abstract domain which is a *partially reduced product* [9] between $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$. The elements of this abstract domain are pairs from $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}) \times \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$. The analysis computes an invariant for the reachable program configurations at each control point. Almost all the abstract transformers in this analysis are defined by:

$$\mathtt{post}^{\#}(St,(HS_{in},HS_{aux})) = (\mathtt{post}^{\#}{}_{\mathbb{U}}(St,HS_{in}),\mathtt{post}^{\#}{}_{\mathbb{W}}(St,HS_{aux})),$$

where $\mathtt{post}^{\#}{}_{\mathbb{U}}(St,HS_{in})$ is the abstract transformer in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $\mathtt{post}^{\#}{}_{\mathbb{W}}(St,HS_{aux})$ is the abstract transformer in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$. The only exception is the statement p=q->next whose abstract transformer calls a *a partial reduction operator* [9] $\sigma_{\mathbb{W}} : \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}) \times \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}}) \rightarrow \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}) \times \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$. This operator propagates information between the two abstract domains and its output should satisfy the following: for any $A_H \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $A'_H \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$,

$$\begin{array}{c}\sigma^1_{\mathbb{W}}(A_H,A'_H) \sqsubseteq^{\mathbb{HS}} A_H,\ \sigma^2_{\mathbb{W}}(A_H,A'_H) \sqsubseteq^{\mathbb{HS}} A'_H, \text{ and} \\ \gamma(\sigma^1_{\mathbb{W}}(A_H,A'_H)) \cup \gamma(\sigma^2_{\mathbb{W}}(A_H,A'_H)) = \gamma(A_H) \cup \gamma(A'_H),\end{array}$$

where $\sigma^i_{\mathbb{W}}(A_H,A'_H)$ is the projection of $\sigma_{\mathbb{W}}(A_H,A'_H)$ on the ith component, for any $1 \leq i \leq 2$. If St': p=q->next then

$$\begin{array}{l}\mathtt{post}^{\#}(St',(HS_{in},HS_{aux})) = \\ \qquad \sigma_{\mathbb{W}}(\mathtt{post}^{\#}{}_{\mathbb{U}}(St',HS_{in}),\mathtt{post}^{\#}{}_{\mathbb{W}}(St',HS_{aux})).\end{array} \tag{L}$$

```
while((zm!=NULL) &&
      (zn!=NULL))
{ zm = zm->next;
  zn = zn->next; }
while(zm!=NULL)
  zm = zm->next;
while(zn!=NULL)
  zn = zn->next;
```

Figure 7. A program for computing $\mathtt{infer}_{\mathbb{W}}$.

Let M denote the set of data word variables in $\widetilde{W}_{aux}$. The program that computes $\mathtt{infer}_{\mathbb{W}}$ depends on M. In practice, we can heuristically choose to consider only some of the data word variables in $\widetilde{W}_{aux}$. The result of $\mathtt{infer}_{\mathbb{W}}$ is still sound.

For example, let $M = \{m,n\}$. The program used in $\mathtt{infer}_{\mathbb{W}}$ is given in Figure 7 and consists in `while` loops that traverse the lists represented by the nodes in M.

The initial configuration of the program is a pair of abstract heaps $(\tilde{H}_{in},\tilde{H}_{aux})$ which contain the same graph (N,S,V) such that (1) the graphs contain one node for each data word variable in $\widetilde{W}_{in}$ or $\widetilde{W}_{aux}$ with an edge towards $\sharp$, (2) each node $n \in N$ is labeled by at least two variables, one being `zn`, (3) $\tilde{H}_{in} = (N,S,V,\widetilde{W}_{in})$ and $\tilde{H}_{aux} = (N,S,V,\widetilde{W}_{aux})$. The first loop traverses simultaneously the two list segments. At each iteration, the pointer variables `zm` and `zn` are advanced to the next element. Since the lists may not have the same length, we add another two `while` loops to continue the traversing of the unfinished list starting from where the previous loop stopped.

Let $(A_H,A'_H) \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}}) \times \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ be the postcondition of this program (i.e, the pair of abstract heap sets associated to the last control point). Remark that $\mathtt{fold}^{\#}(A_H) = \{\tilde{H}\}$ and $\mathtt{fold}^{\#}(A'_H) = \{\tilde{H}'\}$, where $\tilde{H}$ and $\tilde{H}'$ are abstract heaps that contain exactly the same graph as the abstract heaps from the precondition. We define $\mathtt{infer}_{\mathbb{W}}(\widetilde{W}_{in},\widetilde{W}_{aux}) = \widetilde{W}$, where $\widetilde{W}$ is the $\mathcal{A}_{\mathbb{U}}$ formula from $\tilde{H}$ projected on the variables from $\widetilde{W}_{in}$.

The partial reduction operator $\sigma_{\mathbb{W}}$***:*** The definition of $\sigma_{\mathbb{W}}$ over abstract heap sets uses a similar operator on abstract heaps which again, uses a similar operator on $\mathbb{LDW}$-domains. The operator $\sigma_{\mathbb{W}}(A_H,A'_H)$ on abstract heap sets takes the join of $\sigma_{\mathbb{W}}(\tilde{H},\tilde{H}')$, for any two isomorphic abstract heaps $\tilde{H} \in A_H$ and $\tilde{H} \in A'_H$. Then, let $\tilde{H} = (N,S,V,\widetilde{W}) \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $\tilde{H}' = (N,S,V,\widetilde{W}') \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ be two isomorphic abstract heaps (we suppose that the nodes related by the isomorphism have the same name). We define

$$\sigma_{\mathbb{W}}(\tilde{H},\tilde{H}') = \left(\left(N,S,V,\sigma^1_{\mathbb{W}}(\widetilde{W},\widetilde{W}')\right),\left(N,S,V,\sigma^2_{\mathbb{W}}(\widetilde{W},\widetilde{W}')\right)\right).$$

The procedure $\mathtt{infer}_{\mathbb{U}}$***:*** When $\mathcal{A}_{\mathbb{W}}$ is some abstract domain $\mathcal{A}'_{\mathbb{U}}$, we obtain an instantiation of $\mathtt{infer}_{\mathbb{W}}$, denoted $\mathtt{infer}_{\mathbb{U}}$. To define $\mathtt{infer}_{\mathbb{U}}$, we have to provide only the definition of the partial reduction operator $\sigma_{\mathbb{U}} : \mathcal{A}_{\mathbb{U}} \times \mathcal{A}'_{\mathbb{U}} \rightarrow \mathcal{A}_{\mathbb{U}} \times \mathcal{A}'_{\mathbb{U}}$. Thus, let $\widetilde{W}$, resp. $\widetilde{W}'$, be an abstract value in $\mathcal{A}_{\mathbb{U}}$, resp. $\mathcal{A}'_{\mathbb{U}}$. We define $\sigma^1_{\mathbb{U}}(\widetilde{W},\widetilde{W}') = \widetilde{W} \wedge E'$, where E' is the quantifier-free part of $\widetilde{W}'$, and $\sigma^2_{\mathbb{U}}(\widetilde{W},\widetilde{W}') = \widetilde{W}'$.

In the following, we will describe another instantiation of $\mathtt{infer}_{\mathbb{W}}$, when $\mathcal{A}_{\mathbb{W}}$ is the domain of multiset constraints.

5.2 Combining multiset constraints and universal formulas

Let $\widetilde{W} \in \mathcal{A}_{\mathbb{U}}$ and $\widetilde{W}' \in \mathcal{A}_{\mathbb{M}}$ be two formulas over the set of data word variables N. The $\mathcal{A}_{\mathbb{U}}$ formula $\sigma^1_{\mathbb{M}}(\widetilde{W},\widetilde{W}')$ is obtained from $\widetilde{W}$ by adding new constraints on the values of $\mathtt{hd}(n)$ with $n \in N$ based on the multiset constraint $\widetilde{W}'$. The $\mathcal{A}_{\mathbb{M}}$ formula $\sigma^2_{\mathbb{M}}(\widetilde{W},\widetilde{W}')$ is obtained by adding to $\widetilde{W}'$ an equality $\mathtt{mhd}(n) = \mathtt{mhd}(n')$ for any equality $\mathtt{hd}(n) = \mathtt{hd}(n')$ implied by the first output. In the following, we describe the computation of $\sigma^1_{\mathbb{M}}(\widetilde{W},\widetilde{W}')$.

$$\frac{\mathtt{mhd}(n) \subseteq \mathtt{mtl}(n') \text{ and } \widetilde{W} \sqsubseteq^{\mathbb{U}} \forall y.\, y \in n' \Rightarrow \varphi}{\exists y.\ (\varphi[n'[y] \leftarrow \mathtt{hd}(n))]} \qquad \frac{\mathtt{mhd}(n) \subseteq \mathtt{mhd}(n')}{\mathtt{hd}(n) = \mathtt{hd}(n')}$$

Figure 8. Inference rules for $\sigma^1_{\mathbb{M}}$.

To help the intuition, we start by an example. Let $\widetilde{W}:\ \forall y.\, y \in \mathtt{tl}(n) \Rightarrow n[y] > 5$ and $\widetilde{W}':\ \mathtt{mhd}(n_1) \cup \mathtt{mtl}(n_2) = \mathtt{mtl}(n)$.

By an abuse of notation we can rewrite $\widetilde{W}$ as $\forall val.\ val \in \mathtt{mtl}(n).\ val < 5$, where *val* is a variable interpreted as an integer. Notice that $\widetilde{W}'$ implies that $\mathtt{hd}(n_1)$ belongs to $\mathtt{mtl}(n)$ and consequently we obtain that $\widetilde{W} \wedge \widetilde{W}'$ implies $\mathtt{hd}(n_1) < 5$. This deduction can be done using the first inference rule in Figure 8.

In general, a multiset constraint induces multiple choices w.r.t. the multisets to which the singletons belong to. For example if $\widetilde{W}' ::= \mathtt{mhd}(n_1) \cup \mathtt{mtl}(n_2) = \mathtt{mtl}(n) \cup \mathtt{mtl}(m)$ then $\widetilde{W}'$ implies that either $\mathtt{hd}(n_1) \in \mathtt{mtl}(n)$ or $\mathtt{hd}(n_1) \in \mathtt{mtl}(m)$. In each case, the property on $\mathtt{hd}(n_1)$ added to the formula $\widetilde{W}$ might be different. However, there are a finite number of choices. For each of them, we construct a strengthening of $\widetilde{W}$ and then we define $\sigma^1_{\mathbb{M}}(\widetilde{W}, \widetilde{W}')$ as the join of all of these strengthenings.

Formally, the formula $\sigma^1_{\mathbb{M}}(\widetilde{W}, \widetilde{W}')$ is built as follows:

1. we deduce the set of all conjunctions of the form
$$\psi ::= \mathtt{mhd}(n_1) \subseteq bt_1 \wedge \ldots \wedge \mathtt{mhd}(n_k) \subseteq bt_k, \qquad \text{(M)}$$
where bt_i is a basic multiset term, for any $1 \leq i \leq k$, such that (1) $\widetilde{W}'$ implies the disjunction of all formulas ψ as above (in the logic obtained from $\mathcal{A}_{\mathbb{M}}$ by adding the usual inclusion operator between multisets $\subseteq$) and (2) ψ contains exactly once all the terms of the form $\mathtt{mhd}(n_i)$ from $\widetilde{W}'$. Note that if, for example, $bt_1 = \mathtt{mhd}(n')$ then the conjunction will not contain any other atomic formula over the term $\mathtt{mhd}(n')$.
2. for every such conjunction, we use its atomic formulas to apply the inference rules in Figure 8 and deduce new facts on the values of $\mathtt{hd}(n)$ with $n \in N$. These facts are conjuncted to $\widetilde{W}$.
3. $\sigma^1_{\mathbb{M}}(\widetilde{W}, \widetilde{W}')$ is the join of all abstract values in $\mathcal{A}_{\mathbb{U}}$ obtained in the previous step.

6. Applications

In this section, we describe several applications of the procedure $\mathtt{strengthen}_{\mathbb{W}}$.

6.1 Changing the set of patterns

To obtain a compositional analysis with the abstract domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$, we need to transform an abstract value A_1 in a domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}^1_{\mathbb{U}})$ over a set of patterns $\mathcal{P}_1$ into an abstract value A_2 in a domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}^2_{\mathbb{U}})$ over a set of patterns $\mathcal{P}_2 \neq \mathcal{P}_1$.

Thus, we define an operator $\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2) : \mathcal{A}^1_{\mathbb{U}} \to \mathcal{A}^2_{\mathbb{U}}$ parameterized by a pair of pattern sets $(\mathcal{P}_1, \mathcal{P}_2)$, such that $\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2)(\widetilde{W}_1)$ is an over-approximation of $\widetilde{W}_1$ in the domain $\mathcal{A}^2_{\mathbb{U}}$. Intuitively, $\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2)(\widetilde{W}_1)$ returns a formula $\widetilde{W}_2$ which contains (1) constraints from $\widetilde{W}_1$ using the patterns in $\mathcal{P}_1 \cap \mathcal{P}_2$ and (2) constraints using the patterns in $\mathcal{P}_2 \setminus \mathcal{P}_1$ implied by $\widetilde{W}_1$ (in $\mathsf{FO}(DWVar, DVar, \mathbb{O}, \mathbb{P})$). Thus,
$$\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2)(\widetilde{W}_1) = \mathtt{strengthen}_{\mathbb{U}^2}(\top^{\mathbb{U}^2}, \widetilde{W}_1),$$
where $\top^{\mathbb{U}^2}$ is the top element in $\mathcal{A}^2_{\mathbb{U}}$.

The extension of $\mathtt{convert}$ to an operator on the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ domains is done in a straightforward manner: the graph is kept the same and only constraints associated to the graph are converted. This operator is the base ingredient for the following applications.

6.2 Computing procedure summaries

In order to be able to compute procedure summaries parametrized by different sets of guard patterns, we modify the definition of the abstract transformers corresponding to procedure calls and returns as follows. Let Q_1 be a procedure for which a set of patterns $\mathcal{P}_1$ has been fixed. Suppose that Q_1 calls a procedure Q_2 associated with the set of patterns $\mathcal{P}_2$. The transformer $\mathtt{post}^\#(\textbf{call } \mathbf{y} = Q_2(\mathbf{x}), \tilde{R}^c)$, where $\tilde{R}^c$ is over the patterns $\mathcal{P}_1$, computes the local graph $\mathtt{local}(\tilde{R}^c, Q_2(\mathbf{x}))$ (see Section 4) over patterns in $\mathcal{P}_1$ and then applies $\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2)$ to obtain an over-approximation of the local graph for Q_2. Similarly, the transformer $\mathtt{post}^\#(\textbf{ret } \mathbf{y} = Q_2(\mathbf{x}))$ is modified by applying $\mathtt{convert}(\mathcal{P}_2, \mathcal{P}_1)$ to the combination between the context of the call and the summary for Q_2.

As shown for the $\mathtt{quicksort}$ example, the $\mathtt{strengthen}_{\mathbb{M}}$ operator allows to increase the precision of the analysis in the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ domains. For this, the analysis computes two over-approximations for the mapping ρ defining the semantics of a program: a mapping $\rho^\#_{\mathbb{U}}$ with values in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and a mapping $\rho^\#_{\mathbb{M}}$ with values in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$. The abstract values at the initial control point of the CFG, c_0, are such that $\rho^\#_{\mathbb{M}}(c_0)$ is an over-approximation in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$ of $\rho^\#_{\mathbb{U}}(c_0)$. The recursive equations for $\rho^\#_{\mathbb{U}}$ and $\rho^\#_{\mathbb{M}}$ remain unchanged. The only difference is the definition of the abstract transformer in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ corresponding to procedure returns.

Consider the edge $e_{Q_2} \xrightarrow{\textbf{ret } \mathbf{y} = Q_2(\mathbf{x})} r$ in the CFG of some procedure Q_1. Also, let c be the call point associated to r. The abstract transformer $\mathtt{post}^\#(\textbf{ret } \mathbf{y} = Q_2(\mathbf{x}), \tilde{R}^e)$, where $\tilde{R}^e = (N^e, S^e, V^e, \widetilde{W}^e) \in \rho^\#_{\mathbb{U}}(e_{Q_2})$, is an abstract heap set obtained by composing any relation $\tilde{R}^c = (N^c, S^c, V^c, \widetilde{W}^c) \in \rho^\#_{\mathbb{U}}(c)$ with $\tilde{R}^e$ and a relation $\tilde{R}^e_{aux} = (N^e_{aux}, S^e_{aux}, V^e_{aux}, \widetilde{W}^e_{aux}) \in \rho^\#_{\mathbb{M}}(e_{Q_2})$. This composition is applied only if the three relations correspond to the same call. If it is the case, the composition is an abstract heap $\tilde{R}^r = (N^r, S^r, V^r, \widetilde{W}^r)$, where (N^r, S^r, V^r) is built as in Section 4, and $\widetilde{W}^r$ is built by replacing $\mathtt{Combine}(\widetilde{W}^c, \widetilde{W}^e)$ with $\mathtt{strengthen}_{\mathbb{M}}(\mathtt{Combine}(\widetilde{W}^c, \widetilde{W}^e), \widetilde{W}_{aux})$.

6.3 Assertion checking

Let A^1_H be the abstract value in a domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}^1_{\mathbb{U}})$ computed at the control point of the statement $\mathtt{assert}\ \varphi$, where φ describes an element $A^2_H \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}^2_{\mathbb{U}})$. If $\mathcal{P}_1 = \mathcal{P}_2$, checking that A^1_H satisfies φ is done using the entailment operator of $\mathcal{A}_{\mathbb{HS}}$, i.e. $\mathtt{fold}^\#(A^1_H) \sqsubseteq^{\mathbb{HS}} A^2_H$. To improve the precision of this entailment checking, we can apply $\mathtt{strengthen}_{\mathbb{M}}(A^1_H, A^{aux}_H)$, where A^{aux}_H is the assertion synthesized by $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$ at the same control point, to obtain A'_H in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}^1_{\mathbb{U}})$ and then check that $\mathtt{fold}^\#(A'_H) \sqsubseteq^{\mathbb{HS}} A^2_H$ in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}^1_{\mathbb{U}})$.

When $\mathcal{P}_1 \neq \mathcal{P}_2$, we modify the abstract transformer of $\mathtt{assert}\ \varphi$ such that, for any abstract heap set $A^1_H \in \mathcal{A}_{\mathbb{HS}}(\mathcal{A}^1_{\mathbb{U}})$, $\mathtt{post}^\#(\mathtt{assert}\ \varphi, A^1_H) = A^1_H$ if $\mathtt{fold}^\#(\mathtt{convert}(\mathcal{P}_1, \mathcal{P}_2)(A^1_H)) \sqsubseteq^{\mathbb{HS}} A^2_H$, i.e., the output of $\mathtt{fold}^\#$ for the over-approximation of A^1_H in $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}^2_{\mathbb{U}})$ entails A^2_H.

6.4 Equivalence checking

Let P_1 and P_2 be two procedures having the same input and output formal parameters. Then, P_1 and P_2 are *equivalent* if they return exactly the same heap when they receive the same input. We use the program in Figure 9 to obtain a sound procedure for equivalence checking. In every configuration of this program, the heap contains two disjoint regions, each region representing the heap configuration of one of these procedures. Initially, the program assumes that the heap contains two copies of the same input configuration. For that, we use the predicate $equal(\mathbf{fpi_1}, \mathbf{fpi_2})$ where $\mathbf{fpi_1}$ and $\mathbf{fpi_2}$ are two copies of the input parameters. We consider that $equal(\mathbf{fpi_1}, \mathbf{fpi_2})$ holds for an abstract heap if and only if it is

```
assume(equal(fpi_1,fpi_2));
list *y_1, *y_2;
y_1=P_1(fpi_1);
y_2=P_2(fpi_2);
assert(equal(y_1,y_2));
```

Figure 9. Procedure equivalence checking.

formed of two sub-graphs G_1 and G_2 s.t. (1) G_1, resp. G_2, contains only nodes reachable from pointer variables in $\mathbf{fpi_1}$, resp. $\mathbf{fpi_2}$, (2) G_1 and G_2 are isomorphic, and (3) for any two nodes related by the isomorphism, the integer sequences attached to them are equal. Equality of integer sequences can be expressed using the universally-quantified formula $eq_\forall$ given by the equation (H) (page 6). The two procedures are called on this input configuration. Notice that the procedure P_1 (resp. P_2) can modify only the graph G_1 (resp. G_2). The inter-procedural analysis applied to this program computes an invariant I describing the configurations reachable after returning from the two procedures (line 5). This invariant is formed of a set of abstract heaps. The two procedures are equivalent if for any abstract heap in I, the regions reachable from the output parameters of P_1, $\mathbf{y_1}$, and from the output parameters of P_2, $\mathbf{y_2}$, are equal. To express this equality we use the predicate $equal(\mathbf{y_1},\mathbf{y_2})$.

Notice that the program in Figure 9 can be used in the intra-procedural setting by inlining the procedures. However, the advantage of the compositional inter-procedural analysis is that each procedure is analyzed independently (using the local heap semantics).

7. Experimental results

Implementation details: We have implemented our inter-procedural analysis in a plugin called CELIA [5] of the FRAMA-C platform [4] for C program analysis. CELIA takes as input the ICFG built by FRAMA-C from the C program. The implementation of CELIA invokes/adapts (1) the heap abstract domains $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ and $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{M}})$ provided by the CINV tool [2], (2) the numerical domains of the APRON platform [17], and (3) the generic module of fixpoint computation over control-flow graphs due to B. Jeannet [16]. It has been carried out by implementing in C the abstract transformers including the abstract domain combination/strengthening.

Benchmark: We have applied CELIA to a benchmark of C programs which is available on the web site of CELIA. The benchmark includes all 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. Table 1 gives a sample of functions in this benchmark, split in six classes. The class **sll** includes C functions performing elementary operations on list: *add*ing/*del*eting the *f*irst/*l*ast element, *init*ializing a list of some length. The classes **map** and **map2** include C functions performing a traversal of one resp. two lists, without modifying their structures, but modifying their data. The classes **fold** and **fold2** include C functions computing from one resp. two input lists some output parameters of type list or integer. Finally, the **sort** class includes sorting algorithms on lists. The procedures in classes **map*** and **fold*** are tail recursive, thus we consider for them both iterative and recursive versions. The third column of Table 1 specifies the versions considered (iterative/recursive) and the number of nested loops or recursive calls.

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. Also, the benchmark includes programs allowing to test the applications discussed in the previous section and which we detail in the following.

Computing procedure summaries: Table 1 describes some of our experimental results on the synthesis of procedure summaries.

Column 5 indicates the set of patterns used for the analysis with $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$. These patterns are:

$$P_=(x,x') \equiv \forall y_1 \in \mathtt{tl}(x), y_2 \in \mathtt{tl}(x').\, y_1 = y_2,$$
$$P_1(x) \equiv \forall y \in \mathtt{tl}(x), \qquad P_2(x) \equiv \forall y_1, y_2 \in \mathtt{tl}(x).\, y_1 \le y_2$$

The pattern $P_=$ is used by default for each analysis since it is needed to capture the relation of equality between actual and formal function parameters. The choice of P_1 and P_2 is made according to a heuristics that is based on syntactical criteria such as the number of nested loops or the number of recursive calls in the body of the program. (These numbers are reported in column 3 of Table 1.) The pattern P_1 is used for programs with at least one loop (resp. recursive call) and one iteration variable over lists. The pattern P_2 is used for nested loops, more than one recursive call, or two iteration variables.

Column 6 of Table 1 shows samples of procedure summaries that CELIA can synthesize. (We use the & sign to denote, like in C, the output parameters.)

Columns 4–5 provide the global running times for the analysis, including calls to the APRON libraries. All experiments have been done on an Intel i3-370M with 2.4 GHz and 2 GB of RAM.

All examples in our benchmark corresponding to common functions for list manipulation (classes **sll**–**fold2** in Table 1, except the function `merge`) are analyzed in less than 1 second. During the analysis of these programs, the manipulated relations are represented using at most 6 abstract heaps, each of them having at most 16 nodes. For the rest of the examples, these relations have at most 18 abstract heaps. The sorting algorithms are time consuming due to (1) the use of widening operators (we have implemented) that are more accurate than the standard ones available in APRON, and (2) the frequent use of the `strengthen` operation in examples such as `quicksort`.

Besides dealing with recursion, compositional inter-procedural allows to have a much more scalable analysis. For instance, consider a program that calls the *init(v)* function on 10 different lists. Our analysis computes once the summary of this function and reuse it, while the analysis after inlining computes successively the effect of all the calls. Thus, the inter-procedural analysis is ten times faster for this example than the intra-procedural analysis.

Combination of abstract domains: The use of the `strengthen` operation is needed in many examples of programs with procedure calls. For instance, as we have seen throughout the paper, the analysis of the recursive sorting algorithm `quicksort` requires combining universal formulas with multiset constraints. Without this combination, the `quicksort` procedure must be transformed to have two parameters (the first and the last element of the list), like in [24]. Therefore, the pivot is given as a parameter which helps to recover at the return from the recursive calls the property that all elements are less/greater than the pivot. Actually, the techniques of [24] cannot handle the version of `quicksort` given in Figure 1, which is the standard implementation of the `quicksort`.

Non-recursive programs may also need strengthening operations for their analysis due to the fact that different sets of patterns may be used for different procedure calls. To experiment that, we have considered programs performing multiple calls to procedures given in Table 1, taking $\{P_=, P_1, P_2\}$ as set of patterns for the analysis of the main procedure. For example, we have considered a procedure that calls `bubblesort` on a list x, and then copies it in a variable y using the procedure `clone` (the procedure `bubblesort` is analyzed using $\{P_=, P_1, P_2\}$ and the procedure `clone` using $\{P_=\}$). For the call to `clone`, we obtain that the two lists x and y are equal, but the sortedness property of x is not transferred to y. However, this property can be recovered at the return of `clone` (using the

class	*fun*	*nesting*	$\mathcal{A}_{\mathbb{M}}$	$\mathcal{A}_{\mathbb{U}}$		*Examples of summaries synthesized*
		(loop,rec)	t (s)	$\mathcal{P}$	t (s)	
sll	*create*	$(0,-)$	0.013	$P_=, P_1$	0.021	
	addfst	–	0.003	$P_=$	0.002	
	addlst	$(0,1)$	0.031	$P_=$	0.033	$\rho^{\#}_{\mathbb{U}}(create(\&x,\ell)) :\ \mathtt{hd}(x) = 0 \wedge \mathtt{len}(x) = \ell \wedge \forall y \in \mathtt{tl}(x) \Rightarrow x[y] = 0$
	delfst	–	0.001	$P_=$	0.001	
	dellst	$(0,1)$	0.034	$P_=$	0.042	
map	*init*(v)	$(0,1)$	0.024	$P_=, P_1$	0.034	$\rho^{\#}_{\mathbb{U}}(init(v,x)) :\ \mathtt{len}(x^0) = \mathtt{len}(x) \wedge \mathtt{hd}(x) = v \wedge \forall y \in \mathtt{tl}(x).\ x[y] = v$
	initSeq	$(0,1)$	0.024	$P_=, P_1$	0.034	$\rho^{\#}_{\mathbb{U}}(add(v,x)) :\ \mathtt{len}(x^0) = \mathtt{len}(x) \wedge \mathtt{hd}(x) = \mathtt{hd}(x^0) + v \wedge$
	add(v)	$(0,1)$	0.021	$P_=$	0.032	$\forall y_1 \in \mathtt{tl}(x), y_2 \in \mathtt{tl}(x^0).\ y_1 = y_2 \Rightarrow x[y_1] = x^0[y_2] + v$
map2	*add*(v)	$(0,1)$	0.089	$P_=$	0.517	$\rho^{\#}_{\mathbb{U}}(add(v,x,z)) :\ \mathtt{len}(x^0) = \mathtt{len}(x) \wedge \mathtt{len}(z^0) = \mathtt{len}(z) \wedge eq_{\forall}(x,x^0) \wedge$
	copy	$(0,1)$	0.063	$P_=$	0.078	$\forall y_1 \in \mathtt{tl}(x), y_2 \in \mathtt{tl}(z).\ y_1 = y_2 \Rightarrow x[y_1] + v = z[y_2]$
fold	*del*Pred	$(0,1)$	0.062	$P_=, P_1$	0.145	$\rho^{\#}_{\mathbb{M}}(split(v,x,\&l,\&u)) :\ \mathtt{ms}(x) = \mathtt{ms}(x^0) = \mathtt{ms}(l) \cup \mathtt{ms}(u)$
	max	$(0,1)$	0.031	$P_=, P_1$	0.048	$\rho^{\#}_{\mathbb{U}}(split(v,x,\&l,\&u)) :\ eq_{\forall}(x,x^0) \wedge \mathtt{len}(x) = \mathtt{len}(l) + \mathtt{len}(u) \wedge$
	clone	$(0,1)$	0.071	$P_=$	0.315	$l[0] \leq v \wedge \forall y \in \mathtt{tl}(l) \Rightarrow l[y] \leq v \wedge$
	split	$(0,1)$	0.245	$P_=, P_1$	0.871	$u[0] > v \wedge \forall y \in \mathtt{tl}(u) \Rightarrow u[y] > v$
fold2	*equal*	$(0,1)$	0.127	$P_=$	0.261	$\rho^{\#}_{\mathbb{M}}(merge(x,z,\&r)) :\ \mathtt{ms}(x) \cup \mathtt{ms}(z) = \mathtt{ms}(r) \wedge \mathtt{ms}(x^0) = \mathtt{ms}(x) \wedge \ldots$
	concat	$(0,1)$	0.217	$P_=, P_1, P_2$	0.806	$\rho^{\#}_{\mathbb{U}}(merge(x,z,\&r)) :\ eq_{\forall}(x,x^0) \wedge eq_{\forall}(z,z^0) \wedge sorted(x^0) \wedge sorted(z^0) \wedge$
	merge	$(0,1)$	1.014	$P_=, P_1, P_2$	2.306	$sorted(r) \wedge \mathtt{len}(x) + \mathtt{len}(z) = \mathtt{len}(r)$
sort	*bubble*	$(1,-)$	0.387	$P_=, P_1, P_2$	2.190	
	insert	$(1,-)$	0.557	$P_=, P_1, P_2$	3.292	$\rho^{\#}_{\mathbb{M}}(quicksort(x)) :\ \mathtt{ms}(x) = \mathtt{ms}(x^0) = \mathtt{ms}(res)$
	quick	$(-,2)$	1.541	$P_=, P_1, P_2$	121.1	$\rho^{\#}_{\mathbb{U}}(quicksort(x)) :\ eq_{\forall}(x,x^0) \wedge sorted(res)$
	merge	$(-,2)$	1.547	$P_=, P_1, P_2$	95.94	

Table 1. Experimental results for functions in our benchmark.

`strengthen` operation) from the fact that y is equal to x and that x is sorted.

Equivalence checking: We have experimented this approach for checking equivalence between sorting algorithms. The `strengthen` operation plays an essential role. To explain this, consider the example from introduction which considers the equivalence checking of two sorting procedures P_1 and P_2 working on two input lists I_1 and I_2, and producing two outputs O_1 and O_2. The problem is reduced to checking the validity of the implication (C) (page 3) because, for the template of Figure 9, we have that:

- $equal(I_1, I_2)$ corresponds to the assume statement at line 1,
- $sorted(O_1)$ (resp. $\mathtt{ms}(I_1) = \mathtt{ms}(O_1)$) is the summary of P_1 in the $\mathcal{A}_{\mathbb{U}}$ (resp. $\mathcal{A}_{\mathbb{M}}$) domain,
- $sorted(O_2)$ (resp. $\mathtt{ms}(I_2) = \mathtt{ms}(O_2)$) is the summary of P_2 in the $\mathcal{A}_{\mathbb{U}}$ (resp. $\mathcal{A}_{\mathbb{M}}$) domain,
- $equal(O_1, O_2)$ corresponds to the assert statement at line 5.

As explained in Section 6.4, the $equal(I_1, I_2)$ annotation is translated in the $\mathcal{A}_{\mathbb{U}}$ domain into a $eq_{\forall}$ formula (equation H) and approximated in the $\mathcal{A}_{\mathbb{M}}$ domain into the eq_m formula (equation I). To check the validity of (C), we call $\mathtt{strengthen}(\psi^{\mathbb{U}}, \psi^{\mathbb{M}})$ where:

$$\begin{array}{lcl} \psi^{\mathbb{U}} & : & eq_{\forall}(I_1, I_2) \wedge sorted(O_1) \wedge sorted(O_2) \\ \psi^{\mathbb{M}} & : & \mathtt{ms}(I_1) = \mathtt{ms}(I_2) \wedge \mathtt{ms}(I_1) = \mathtt{ms}(O_1) \wedge \mathtt{ms}(I_2) = \mathtt{ms}(O_2), \end{array}$$

and obtain the universally quantified formula $\psi^{\mathbb{U}} \wedge eq_{\forall}(O_1, O_2)$.

For all experiments, the time needed to check the validity of (C) is negligible compared with the time to compute the procedure summaries.

8. Related work

Automatic synthesis of assertions about programs with dynamic data structures has been addressed using different approaches including abstract interpretation [2, 3, 6, 8, 11–13, 15, 20, 21, 23–27, 29], constraint solving [1, 14], Craig interpolants [18].

In the intra-procedural case, several works consider invariant synthesis for programs that manipulate dynamic data structures. The generated invariants are either universally-quantified first-order formulas [2, 13, 15, 20] or multiset constraints [2, 21].

Concerning the approaches based on abstract interpretation which can handle procedure calls, most of them [3, 8, 23, 25] focus on shape properties and do not consider constraints on sizes and data. The approach in [24] 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. For example, in [24] the predicate $dle(v, u)$ allows to synthesize the summary for a procedure that sorts in ascending order, but cannot be used for a procedure that sorts in descending order. However, using the pattern $y1 \leq y2$ allows with our approach to synthesize the summaries for both kind of procedures. The same pattern may also allow to discover other properties than sortedness. Actually, patterns are in many cases simple (ordering/equality constraints) and can be discovered using natural heuristics based on the program syntax or proposed/guessed by the user, whereas constraints on data can be more complex. Our approach allows to discover (maybe unpredictable) data constraints for given guard patterns. To establish the fact that a procedure preserves the data values in the input list, the method used in [24] is based on reachability, that is, every cell in the input list remains reachable in the output list. This method can be applied only for programs that never modify/permute the contents of data fields. In our approach, using the multiset domain, we can handle programs that can permute positions of cells in the list or modify/permute the contents of their data fields.

The approach in [11] considers abstract domains where the elements are pairs formed of a graph and a constraint on data. The inter-procedural analysis based on these domains can not synthesize constraints in form of universally-quantified formulas as our analysis can do. In [26], the authors introduce *trace partitioning abstract domains* which start from a partition of the set of traces and compute an invariant for each class. The partitioning can be static (usually based on the control structure of the program) or dynamic. From this point of view, the approach in [26] considers mainly statically-defined partitions. The abstract domain in our paper, based on the unfolding/folding operations, can be seen as an instance of a trace partitioning abstract domain with a dynamic partitioning. The corresponding partitioning puts in the same class all the traces for which the number of dereferences of the `next` pointer field is the same modulo some fixed constant k (which is a parameter of the analysis). The approach in [26] considers mainly numerical abstract domains and it is not faced to the difficulties raised by a compositional analysis on programs manipulating dynamic data structures. The analysis in [12] combines a numerical abstract domain with a shape analysis. It is not restricted by the class of data structures but it considers only properties related to the shape and to the size of the memory.

9. Conclusion

We have defined an accurate inter-procedural analysis for programs with lists and data. The key contribution of this paper is a technique for combining the analysis in different abstract domains and its use in compositional analysis techniques that are able to infer non trivial procedure summaries.

The combination mechanism we propose, based on an unfolding/folding technique combined with partial reduction operators, could be applied for other abstract domains than those considered in this paper. In particular, other abstract domains based on first-order formulas, e.g., the one defined in [13], can be used to strengthen the analysis in our domain of universal formulas.

Another interesting aspect of our work is that it allows to manipulate constraints without requirement of decidability, contrary to many works based on decision procedures. Our abstract domains allow actually to express verification conditions that appear in pre-post condition reasoning. These conditions are typically implications, and then, our entailment checking can be used to check safely their validity. Therefore our framework allows to combine smoothly pre-post condition reasoning with assertion synthesis.

Future work includes the generalization of our framework to structures such as multi-linked lists, trees, and nested structures.

References

[1] D. Beyer, T.A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394. Springer, 2007.

[2] A. Bouajjani, C. Drăgoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *CAV*, volume 6174 of *LNCS*, pages 72–88. Springer, 2010.

[3] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300. ACM, 2009.

[4] CEA. *Frama-C Platform*. `htp://frama-c.com`.

[5] Celia plugin. `http://www.liafa.jussieu.fr/celia`.

[6] B.-Y.E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260. ACM, 2008.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979.

[10] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[11] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL*, pages 157–168. ACM, 1990.

[12] S. Gulwani, T. Lev-Ami, and S. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251. ACM, 2009.

[13] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.

[14] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS*, volume 5505 of *LNCS*, pages 262–276. Springer, 2009.

[15] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[16] B. Jeannet. *Fixpoint*. `http://gforge.inria.fr/`.

[17] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[18] R. Jhala and K.L. McMillan. Array abstractions from proofs. In *CAV*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[19] R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, volume 3385 of *LNCS*, pages 181–198. Springer, 2005.

[20] B. McCloskey, T.W. Reps, and S. Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, volume 6337 of *LNCS*, pages 71–99. Springer, 2010.

[21] V. Perrelle and N. Halbwachs. An analysis of permutations in arrays. In *VMCAI*, volume 5944 of *LNCS*, pages 279–294, 2010.

[22] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[23] N. Rinetzky, J. Bauer, T.W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309. ACM, 2005.

[24] N. Rinetzky, S. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, volume 3672 of *LNCS*, pages 284–302. Springer, 2005.

[25] X. Rival and B.-Y.E. Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186. ACM, 2011.

[26] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29, 2007.

[27] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[28] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. New York University, 1981.

[29] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.