Invariant Synthesis for Programs Manipulating Lists with Unbounded Data [⋆]

A. Bouajjani[1], C. Drăgoi[1], C. Enea[1], A. Rezine[2], and M. Sighireanu[1]
[1] LIAFA, University of Paris Diderot and CNRS, 75205 Paris 13, France,
{abou,cezarad,cenea,sighirea}@liafa.jussieu.fr
[2] Uppsala University, Sweden, rahmed@it.uu.se

Abstract. We address the issue of automatic invariant synthesis for sequential programs manipulating singly-linked lists carrying data over infinite data domains. We define for that a framework based on abstract interpretation which combines a specific finite-range abstraction on the shape of the heap with an abstract domain on sequences of data, considered as a parameter of the approach. We instantiate our framework by introducing different abstractions on data sequences allowing to reason about various aspects such as their sizes, the sums or the multisets of their elements, or relations on their data at different (linearly ordered or successive) positions. To express the latter relations we define a new domain whose elements correspond to an expressive class of first order universally quantified formulas. We have implemented our techniques in an efficient prototype tool and we have shown that our approach is powerful enough to generate non-trivial invariants for a significant class of programs.

1 Introduction

Invariant synthesis is an essential ingredient in various program verification and analysis methodologies. In this paper, we address this issue for sequential programs manipulating singly-linked lists carrying data over infinite data domains such as integers or reals. Specifications of such programs typically involve constraints on various aspects such as the sizes of the lists, the multisets of their elements, as well as relations between data at their different positions, e.g., ordering constraints or even more complex arithmetical constraints on consecutive elements, or combining relations between the sizes, the sum of all elements, etc., of different lists.

Consider for instance the procedure Dispatch3 given in Figure 1(b). It puts all the cells of the input list which have data larger than 3 to the list grt, and it puts all the other ones to the list less. Naturally, the specification of this procedure (at line 10) includes (1) the property expressed by the universally quantified first-order formula

$$\forall y.\ \mathtt{grt} \xrightarrow{*} y \Rightarrow \mathtt{data}(y) \geq 3 \quad \wedge \quad \forall y.\ \mathtt{less} \xrightarrow{*} y \Rightarrow \mathtt{data}(y) < 3 \qquad \text{(A)}$$

which says that every cell y reachable from grt (resp. less) have data greater (resp. smaller) than 3, and (2) the preservation property saying that the multiset of the input list is equal to the union of the multisets of the two output lists. This property is expressed by

$$\mathtt{ms\_init} = \mathtt{ms}(\mathtt{grt}) \cup \mathtt{ms}(\mathtt{less}) \qquad \text{(B)}$$

[⋆] This work was partly supported by the French National Research Agency (ANR) projects Averiss (ANR-06-SETIN-001) and Veridyc (ANR-09-SEGI-016).

where the variable $\mathtt{ms\_init}$ represents the multiset of the elements of the input list, and $\mathtt{ms}(\mathtt{grt})$ (resp. $\mathtt{ms}(\mathtt{less})$) denotes the multiset of the elements of $\mathtt{grt}$ (resp. $\mathtt{less}$). A weaker property is length preservation, expressed by:

$$\mathtt{len\_init} = \mathtt{len}(\mathtt{grt} \xrightarrow{+} \mathtt{null}) + \mathtt{len}(\mathtt{less} \xrightarrow{+} \mathtt{null}), \qquad \text{(C)}$$

where $\mathtt{len\_init}$ is the length of the input list.

```
   procedure Fibonacci(list* head)
1: { list *x=head;
2:   int m1=1;
3:   int m2=0;
4:   while (x != null)
5:   { x->data=m1+m2;
6:     m1=m2;
7:     m2=x->data;
8:     x=x->next;
9:   }
10:}
```

(a)

```
   procedure Dispatch3(list* head)
1: { list *tmp=null, grt=null, less=null;
2:   while (head != null)
3:   { tmp=head->next;
4:     if (head->data >= 3)
5:     { head->next=grt; grt=head; }
6:     else
7:     { head->next=less; less=head; }
8:    head=tmp;
9:   }
10:}
```

(b)

Fig. 1. Procedures Fibonacci and Dispatch3.

The specification of sorting algorithms is similar since it includes an ordering constraint on the output list that is easily expressible using a universally quantified first-order formula, and a preservation constraint saying that the input and output lists have the same elements that is expressible using multiset constraints.

Moreover, an interesting property of the procedure Dispatch3 above is that the sum of all the elements in the list grt is larger than 3 times the size of that list, i.e.

$$\sum_{\mathtt{grt} \xrightarrow{*} y} \mathtt{data}(y) - 3 \times \mathtt{len}(\mathtt{grt} \xrightarrow{+} \mathtt{null}) \geq 0 \qquad \text{(D)}$$

Consider now the procedure Fibonacci given in Figure 1(a). It takes a list as an input and initializes its elements following the Fibonacci sequence. The natural specification for the procedure (at line 10) is expressed by the universally-quantified formula

$$\forall y_1, y_2, y_3.\ \mathtt{head} \xrightarrow{*} y_1 \to y_2 \to y_3 \Rightarrow \mathtt{data}(y_3) = \mathtt{data}(y_2) + \mathtt{data}(y_1) \qquad \text{(E)}$$

which corresponds precisely to the definition of the Fibonacci sequence. Moreover, an interesting property of the Fibonacci sequence $\{f_i\}_{i\geq 1}$ is that $\sum_{i=1}^{i=n} f_i = 2f_n + f_{n-1} - 1$. This can be expressed (again at line 10) by the following constraint

$$\sum_{\mathtt{head} \xrightarrow{*} y} \mathtt{data}(y) = 2 \times \mathtt{m2} + \mathtt{m1} - 1 \qquad \text{(F)}$$

The automatic synthesis of invariants like those shown above is a challenging problem since it requires combining in a nontrivial way different analysis techniques. This paper introduces a uniform framework based on abstract interpretation for tackling this problem. We define a generic abstract domain $\mathcal{A}_{\mathbb{HS}}$ for reasoning about dynamic lists with unbounded data which includes an abstraction on the shape of the heap and which is parametrized by some abstract domain on finite sequences of data (a data words abstract domain, $\mathbb{DW}$-domain for short). The latter is intended to abstract the sequences of data in the lists by capturing relevant aspects such as their sizes, the sums or the multisets of their elements, or some class of constraints on their data at different (linearly ordered or successive) positions.

We instantiate our framework by defining new $\mathbb{DW}$-domains corresponding to the aspects mentioned above. The most complex $\mathbb{DW}$-domain is composed of first-order formulas such that their (quantified) universal part is of the form $\forall \mathbf{y}.\ (P \Rightarrow U)$, where $\mathbf{y}$ is a vector of variables interpreted to 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, and possibly also on the positions when data are of numerical type. Then, we assume that our $\mathbb{DW}$-domain on first-order properties is parametrized by some abstract data domain, and we consider that U is defined as an object in that abstract domain. For the sake of simplicity of the presentation, we consider in the rest of the paper that the data are always of type integer (and therefore it is possible to take as abstract data domains the standard octagons or polyhedra abstract domains for instance). Our approach can in fact be applied to any other data domain. As for the syntax of the constraint P, we assume that we are given a finite set of fixed patterns (or templates) such as, for instance, order constraints or difference constraints.

Then, an object in the domain $\mathcal{A}_{\mathbb{HS}}$ is a finite collection of pairs $(\widetilde{G},\widetilde{W})$ such that (1) $\widetilde{G}$ is a graph (where each node has an out-degree of at most 1) representing the set of all the garbage-free heap graphs that can be obtained by inserting sequences of non-shared nodes (nodes with in-degree 1) between any pair of nodes in $\widetilde{G}$ (thus edges in $\widetilde{G}$ represents list segments without sharing), and (2) $\widetilde{W}$ is an abstract object in the considered $\mathbb{DW}$-domain constraining the sequences of data attached to each edge in $\widetilde{G}$. So, all the shared nodes in the concrete heaps are present in $\widetilde{G}$, but $\widetilde{G}$ may have nodes which are not shared. Non-shared nodes which are not pointed by program variables are called simple nodes. We assume that objects in our abstract domain have graphs with k simple nodes, for some given bound k that is also a parameter of the domain. This assumption implies that the number of such graphs is finite (since for a given program with lists it is well known that the number of shared nodes is bounded).

We define sound abstract transformers for the statements in the class of programs we consider. Due to the bound on the number of simple nodes, and since heap transformations may add simple nodes, we use a normalization operation that shrinks paths of simple nodes into a single edge. This operation is accompanied with an operation that generalizes the known relations on the data attached to the eliminated simple nodes in order to produce a constraint (in the $\mathbb{DW}$-domain) on the data word associated with the edge resulting from the normalization. This step is actually quite delicate and special care has to be taken in order to keep preciseness. In particular, this is the crucial step that allows to generate universally quantified properties from a number of relations between a finite (bounded) number of nodes. We have defined sufficient conditions on the sets of allowed patterns under which we obtain best abstract transformers.

We have implemented (in C) a prototype tool CINV based on our approach, and we have carried out several experiments (more than 30 examples) on list manipulating programs (including for instance sorting algorithms such as insertion sort, and the two examples in Figure 1).

2 Modeling and reasoning about programs with singly-linked lists

We consider a class of strongly typed imperative programs manipulating dynamic singly linked lists. We suppose that all manipulated lists have the same type, i.e., reference to a record called `list` including one reference field `next` and one data field `data` of integer type. While the generalization to records with several data fields is straightforward, the presence of a single reference field is important for this work. The programs we consider do not contain procedure calls or concurrency constructs.

Program syntax: Programs are defined on a set of data variables *DVar* of type $\mathbb{Z}$ and a set of pointer variables *PVar* of type `list` (which includes the constant `null`). Data variables can be used in *data terms* built using operations over $\mathbb{Z}$ and in boolean conditions on data built using predicates over $\mathbb{Z}$. Pointers can be used in data terms (`p->data` and in assignments corresponding to heap manipulation like memory allocation/deallocation (`new`/`free`), selector field updates (`p->next=`..., `p->data=`...), and pointer assignments (`p=`...). Boolean conditions on pointers are built using predicates (`p==q` and `p==null`) testing for equality and definedness of pointer variables. No arithmetics is allowed on pointers. We allow sequential composition (`;`), conditionals (`if-then-else`), and iterations (`while`). The full syntax is given in [2].

Program semantics: A program configuration is given by a configuration for the heap and a valuation of data variables. Heaps can be represented naturally by a directed graph. Each object of type `list` is represented by a node. The constant `null` is represented by a distinguished node $\sharp$. The pointer field `next` is represented by the edges of the graph. The nodes are labeled by the values of the data field `data` and by the program pointer variables which are pointing to the corresponding objects. Every node has exactly one successor, except for $\sharp$, the node representing `null`. For example, the graph in Figure 4(a) represents a heap containing two lists $[4,0,5,2,3]$ and $[1,4,3,6,2,3]$ which share their two last cells. Two of the nodes are labeled by the pointer variables x and y.

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

A node which is labeled by a pointer variable or which has at least two predecessors is called a cut point. *Otherwise, it is called a* simple node.

In the following, we consider only heaps without garbage, i.e., all the nodes are reachable from nodes labeled by pointer variables. For simplicity, we suppose that each pointer assignment `p->next=q`, resp. `p=q`, is preceded by `p->next=null`, resp. `p=null`. We define a postcondition operator, denoted $\texttt{post}(St,H)$, for any statement St and any heap H. Figure 2 illustrates a part of its definition that contains all the important graph transformations; the full definition is provided in [2]. A collecting semantics can be defined as usual by extending `post` to sets of heaps. The heap H_{err} is a special value denoting the sink heap configuration obtained when `null` dereferences are done.

The formal definition of operators used in this semantics is given on Figure 3. To access the components of a heap H, we use the dotted notation, e.g., $H.N$ denotes the set of nodes of H. For components which are functions, e.g., S, we use currified operators `get` to apply these components to any heap. In the conclusion of the rule a-ptr2, we abuse notation by letting $\sharp$ denote the constant function which associates $\sharp$ to each node. For instance, $\texttt{get}S(f_n)(H)$ returns the successor in H of a node denoted

$$\frac{n \notin H.N \quad H.V(p) = \sharp}{\texttt{post}(p\texttt{=new}, H) = \texttt{addNode}(p,n)(H)}\ \text{a-new} \qquad \frac{H' = \texttt{unfold}(p)(H) \quad H'.V(p) = \sharp}{\texttt{post}(p\texttt{->next=null}, H) = H_{err}}\ \text{a-ptr1}$$

$$\frac{H' = \texttt{unfold}(p)(H) \quad H'.V(p) \neq \sharp}{\texttt{post}(p\texttt{->next=null}, H) = \texttt{delGarbage}(\texttt{upd}S(\texttt{get}V(p), \sharp)(H'))}\ \text{a-ptr2}$$

$$\frac{H.V(p) \neq \sharp \quad \texttt{eval}(dt)(H) \neq \bot}{\texttt{post}(p\texttt{->data=}dt, H) = \texttt{upd}L(\texttt{get}V(p), dt)(H)}\ \text{a-d1} \qquad \frac{H.V(p) = \sharp}{\texttt{post}(p\texttt{->data=}dt, H) = H_{err}}\ \text{a-d2}$$

Fig. 2. A fragment of the definition of $\texttt{post}(St, H)$.

by $f_n(H)$. Similarly, we use the upd operators to alter the components of heaps. The operator $\texttt{addNode}(p,n)(H)$ adds a fresh node n (not in $H.N$) to H s.t. it is pointed by p and its data is arbitrary. The eval operator evaluates data terms in a concrete heap to integer values or, when null is dereferenced, to $\bot$. The operator $\texttt{unfold}(p)(H)$ is introduced to obtain similar definitions for the concrete and abstract program semantics; in the concrete semantics, it is the identity. The operator $\texttt{delGarbage}(H)$ removes from the heap all the garbage nodes using two operators: (1) $\texttt{getGarbage}(H)$ returns the complete set of garbage nodes (computed, e.g., by a graph traversal algorithm starting from the nodes pointed by program variables); (2) $\texttt{proj}(N)(H)$ removes from H a set of nodes $N \subset H.N$ ($f \uparrow N$ denotes a function obtained from f by removing from its domain the set N).

$$\begin{aligned}
\texttt{get}V(p)(H) &\overset{\text{def}}{=} H.V(p) \quad \texttt{get}S(f_n)(H) \overset{\text{def}}{=} H.S(f_n(H)) \qquad\qquad \texttt{unfold}(p)(H) \overset{\text{def}}{=} H \\
\texttt{addNode}(p,n)(H) &\overset{\text{def}}{=} (H.N, H.S[n \mapsto \sharp], H.V[p \mapsto n], H.L[n \mapsto v], H.D) \quad \text{for some } v \in \mathbf{Z} \\
\texttt{delGarbage}(H) &\overset{\text{def}}{=} \texttt{proj}(\texttt{getGarbage}(H))(H) \\
\texttt{proj}(N)(H) &\overset{\text{def}}{=} (H.N \setminus N, (H.S \uparrow N)[\sharp/n]_{n \in N}, (H.V)[\sharp/n]_{n \in N}, H.L \uparrow N, H.D) \\
\texttt{upd}S(f_n, f_m)(H) &\overset{\text{def}}{=} (H.N, H.S[f_n(H) \mapsto f_m(H)], H.V, H.L, H.D) \\
\texttt{upd}L(f_n, dt)(H) &\overset{\text{def}}{=} (H.N, H.S, H.V, H.L[f_n(H) \mapsto \texttt{eval}(dt)(H)], H.D)
\end{aligned}$$

Fig. 3. Operators used in $\texttt{post}(St, H)$.

3 Abstract domain for program configurations

Starting from a heap H, we can define a *precise abstraction* by (1) a graph G containing as nodes at least all the cut points in H such that two nodes in G are connected by an edge if there exists a path between them in H, and (2) a constraint $\widetilde{W}$ that associates to each node n in G, let m be its successor, a word over $\mathbb{Z}$ which represents the data values of the path $nn_1 \ldots n_k$ from H, where n_k is a predecessor of m. For example, Figures 4(b–c) give precise abstractions for the heap in Figure 4(a). Coarser abstractions can

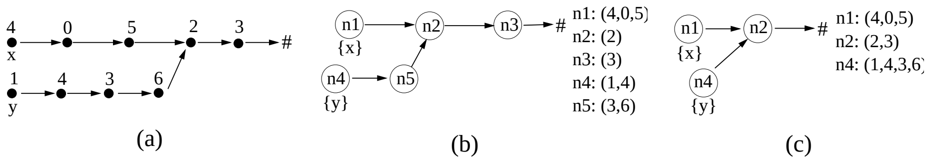
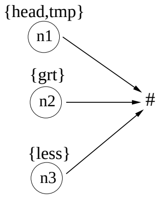


Fig. 4. Concrete and abstract representations for program heaps.

be obtained by replacing $\widetilde{W}$ with less precise constraints characterizing the data words attached to the nodes of the graph and the values of the program data variables. For that, abstract domains on words are used which capture various aspects such as constraints on the sizes, the multisets of their elements, or the data at different positions of the words. For example, Figures 5(a–b) give a precise abstraction of the heap configuration at line 2 of the procedure `Dispatch3` from Figure 1(b). Another abstraction of this configuration is defined using the same graph together with the constraints from Figure 5(c) which characterize the data at any position of the words attached to the nodes of the graph. These constraints are expressed by (universally quantified) first-order formulas where $\mathtt{hd}(n2)$ denotes the first symbol of the word denoted by $n2$, y is a variable interpreted as a position in some word, $y \in \mathtt{tl}(n2)$ means that y belongs to the tail of $n2$, and $n2[y]$ is a term interpreted as the data at the position y of $n2$.



(a)

Precise constraints:

$n1 : (3,4,5)$
$n2 : (5,6)$
$n3 : (1,2)$

(b)

Universally quantified formulas:

$\mathtt{hd}(n2) \geq 3 \wedge \mathtt{hd}(n3) < 3$
$\forall y.\, y \in \mathtt{tl}(n2) \Rightarrow n2[y] \geq 3$
$\forall y.\, y \in \mathtt{tl}(n3) \Rightarrow n3[y] < 3$

(c)

Fig. 5. Different abstractions for some configuration of `Dispatch3`

3.1 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 $\mathcal{C}$ [5] if there exists a Galois connection between $\mathcal{C}$ and $\mathcal{A}$, that is, a pair of monotone functions $(\alpha : C \to A, \gamma : A \to C)$ such that for any $c \in C$ and $a \in A$, $\alpha(c) \sqsubseteq a$ iff $c \subseteq \gamma(a)$. Also, $\mathcal{C}$ is called the *concrete domain* for $\mathcal{A}$. 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 $a \in A$. $f^{\#}$ is a *best abstraction* if $\alpha(f(\gamma(a))) = f^{\#}(a)$ and it is an *exact abstraction* if $f(\gamma(a)) = \gamma(f^{\#}(a))$, for any abstract value $a \in A$.

3.2 Data words (abstract) domains

To represent constraints on words associated to the nodes of the graph as in Figure 5(b)-(c) and the values of the program data variables we use elements from a *data words abstract domain*. Let *DWVar* be a set of variables called data word variables and let $\mathbb{Z}^{+}$ denote the set of non-empty sequences over $\mathbb{Z}$. Also, let $\mathtt{hd}(w)$ (and $\mathtt{tl}(w)$) denote the first element (resp. the tail) of the word w, $[]$ (and $[e]$) the empty word (resp. the word with one element e), and @ the concatenation operator.

Definition 2. *The* data words domain over *DWVar* and *DVar*, *denoted by* $\mathcal{C}_{\mathbb{W}}(\mathit{DWVar}, \mathit{DVar})$, *is the lattice of sets of pairs* (L, D) *with* $L : \mathit{DWVar} \to \mathbb{Z}^{+}$ *and* $D : \mathit{DVar} \to \mathbb{Z}$.

For any data words domain, we define a set of transformers, denoted by $\mathcal{F}_{\mathcal{C}_{\mathbb{W}}}$, as follows ($w, w' \in DWVar$, $d \in DVar$, $W \in \mathcal{C}_{\mathbb{W}}$):

– $\texttt{addSglt}(w,W)$ adds to each pair (L,D) of W a new word w s.t. $\texttt{tl}(L(w)) = []$, i.e., w has only one element,

– $\texttt{selectSglt}(w,W)$ (resp. $\texttt{selectNonSglt}(w,W)$) selects from W the pairs (L,D) for which $\texttt{tl}(L(w)) = []$ (resp. $\texttt{tl}(L(w)) \neq []$), i.e., pairs where the word w has one element (resp. at least two elements),

– $\texttt{split}(w,w',W)$ changes the L component of each pair in W by adding a new word w' and then assigning $[\texttt{hd}(L(w))]$ to $L(w)$ and $\texttt{tl}(L(w))$ to $L(w')$,

– $\texttt{updFst}(w,dt,W)$ changes the L component of each pair (L,D) of W s.t. $\texttt{hd}(L(w))$ takes the value of the arithmetic expression dt in which the basic terms are integer constants, data variables, and terms of the form $\texttt{hd}(w')$ with $w' \in DWVar$,

– $\texttt{proj}(U,W)$ removes from the domain of L the variables in U, for each $(L,D) \in W$,

– $\texttt{concat}(V,W)$, where V is a vector of data word variables, changes the L component of each pair (L,D) of W by assigning to $V[0]$ the concatenation of the words represented by the variables in V, i.e., $L(V[0])@\cdots@L(V[|V|-1])$ and by projecting out the variables in V except the first one. Then, $\texttt{concat}(V_1,\ldots,V_t,W)$ is the component-wise extension of $\texttt{concat}(V,W)$ to t vectors of data word variables, for any $1 \leq t$.

Definition 3. *$\mathcal{A}_{\mathbb{W}} = \left(A^{\mathbb{W}}, \sqsubseteq^{\mathbb{W}}, \sqcap^{\mathbb{W}}, \sqcup^{\mathbb{W}}, \top^{\mathbb{W}}, \bot^{\mathbb{W}}\right)$ is a* $\mathbb{DW}$-domain over *DWVar* and *DVar if it is an abstract domain for $\mathcal{C}_{\mathbb{W}}(DWVar, DVar)$. Let $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{W}}}$ denote the set of abstract transformers corresponding to $\mathcal{F}_{\mathcal{C}_{\mathbb{W}}}$.*

We define in Section 4 a $\mathbb{DW}$-domain which formalizes the abstraction from Figure 5(c). Moreover, we define in [2] the $\mathbb{DW}$-domain $\mathcal{A}_{\Sigma}$ (resp. $\mathcal{A}_{\mathbb{M}}$) representing constraints over the sum (resp. the multiset) of data in a word.

3.3 The domain of abstract heap sets

In the following, we assume that for each node of a heap there exists a data word variable with the same name.

Definition 4. *An* abstract heap over *PVar*, *DVar*, and a $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{W}}$ *is a tuple $\widetilde{H} = \langle N, S, V, \widetilde{W}\rangle$ where N, S, V are as in the definition of heaps, and $\widetilde{W}$ is an abstract value in $\mathcal{A}_{\mathbb{W}}$ over the data word variables $N \setminus \{\sharp\}$ and the data variables DVar. A k-*abstract heap *is an abstract heap with at most k simple nodes.*

An example of an abstract heap is given in Figure 5(a) and (c). 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 we consider (see Section 2) contain a bounded number of cut points [15].

Finally, we define $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}}) = \left(A^{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}}), \sqsubseteq^{\mathbb{HS}}, \sqcap^{\mathbb{HS}}, \sqcup^{\mathbb{HS}}, \top^{\mathbb{HS}}, \bot^{\mathbb{HS}}\right)$ as a finite powerset domain over $\mathcal{A}_{\mathbb{H}}(k, \mathcal{A}_{\mathbb{W}})$. Its elements are called *k-abstract heap sets*. Obviously, $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}})$ is an abstract domain for $\mathcal{C}_{\mathbb{H}}$.

Definition 5. *A k-*abstract heap set over *PVar*, *DVar*, and a $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{W}}$ *is a finite set of non-isomorphic k-abstract heaps over PVar, DVar, and* $\mathcal{A}_{\mathbb{W}}$.

The operators associated to $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{W}})$ and its widening operator are obtained from those of $\mathcal{A}_{\mathbb{H}}(k,\mathcal{A}_{\mathbb{W}})$ as usual [6]. 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 heap graphs.

3.4 Abstract postcondition transfomer

The abstract postcondition transformer $\texttt{post}^{\#}$ on abstract heap sets is obtained by replacing in the definition of $\texttt{post}$ (see Figure 2) the heap H by an abstract heap set A_H and every concrete operator defined in Figure 3 by its abstract version given in Figure 6.

Next, $\texttt{post}^{\#}$ is extended to obtain a postcondition transformer on k-abstract heap sets, denoted $\texttt{post}^{\#}_k$, by

$$\texttt{post}^{\#}_k(St,A_H) \stackrel{\text{def}}{=} \bigsqcup^{\mathbb{HS}}_{\widetilde{H}\in\texttt{post}^{\#}(St,A_H)} \texttt{Normalize}^{\#}_k(\widetilde{H}),$$

where $\texttt{Normalize}^{\#}_k$ takes as input an abstract heap graph $\widetilde{H}$ and, if $\widetilde{H}$ is not a k-abstract heap graph then it returns a 0-abstract heap graph (an abstract heap graph with no simple nodes). Suppose that $V_1,\ldots,V_t$ are all the (disjoint) paths in $\widetilde{H}$ of the form $nn_1\ldots n_k$, where $k\geq 1$, n is a cut point, n_i is a simple node, for any $1\leq i\leq k$, and the successor of n_k is a cut point. $\texttt{Normalize}^{\#}_k$ calls the transformer $\texttt{concat}^{\#}(V_1,\ldots,V_t,\widetilde{W})$, then it replaces the paths of simple nodes starting from $V_i[0]$ by one edge, and finally it removes from the graph the simple nodes of each V_i.

Remark 1. By definition 4, for any call to $\texttt{concat}^{\#}(V_1,\ldots,V_t,\widetilde{W})$ made by $\texttt{post}^{\#}_k$, the sum $|V_1|+\ldots+|V_t|$ is bounded by $2k$. Consequently, we can define transformers $\texttt{concat}^{\#}$ s.t. $\texttt{concat}^{\#}(V_1,\ldots,V_t,\widetilde{W})$ with $\widetilde{W}\in\mathcal{A}_{\mathbb{W}}$ is undefined if $|V_1|+\ldots+|V_t|>2k$.

$$F(A_H) \stackrel{\text{def}}{=} \bigsqcup^{\mathbb{HS}}_{\widetilde{H}\in A_H} F(\widetilde{H}),\ \text{for any } F\in\{\texttt{get}^{\#}V(p),\texttt{get}^{\#}S(f_n),\texttt{unfold}^{\#}(n),\texttt{addNode}^{\#}(p,n),$$
$$\texttt{delGarbage}^{\#},\texttt{proj}^{\#}(N),\texttt{upd}S^{\#}(f_n,f_m),\texttt{upd}L^{\#}(f_n,dt)\}$$

$$\begin{aligned}
\texttt{get}^{\#}V(p)(\widetilde{H}) &\stackrel{\text{def}}{=} \widetilde{H}.V(p) \quad \texttt{get}^{\#}S(f_n)(\widetilde{H}) \stackrel{\text{def}}{=} \widetilde{H}.S(f_n(\widetilde{H}))\\
\texttt{unfold}^{\#}(n)(\widetilde{H}) &\stackrel{\text{def}}{=} \langle \widetilde{H}.N,\widetilde{H}.S,\widetilde{H}.V,\texttt{selectSglt}^{\#}(n,\widetilde{H}.\widetilde{W})\rangle\\
&\sqcup^{\mathbb{HS}} \langle \widetilde{H}.N\cup\{m\},\widetilde{H}.S[n\mapsto m,m\mapsto\widetilde{H}.S(n)],\widetilde{H}.V,\\
&\qquad \texttt{selectSglt}^{\#}(n,\texttt{split}^{\#}(n,m,\texttt{selectNonSglt}^{\#}(n,\widetilde{H}.\widetilde{W})))\rangle\\
\texttt{addNode}^{\#}(p,n)(\widetilde{H}) &\stackrel{\text{def}}{=} \langle \widetilde{H}.N,\widetilde{H}.S[n\mapsto\sharp],\widetilde{H}.V[p\mapsto n],\texttt{addSglt}^{\#}(n,\widetilde{H}.\widetilde{W})\rangle\\
\texttt{delGarbage}^{\#}(\widetilde{H}) &\stackrel{\text{def}}{=} \texttt{proj}^{\#}(\texttt{getGarbage}(\widetilde{H}))(\widetilde{H})\\
\texttt{proj}^{\#}(N)(\widetilde{H}) &\stackrel{\text{def}}{=} \langle \widetilde{H}.N\setminus N,(\widetilde{H}.S\uparrow N)[\sharp/n]_{n\in N},(\widetilde{H}.V)[\sharp/n]_{n\in N},\texttt{proj}^{\#}(N,\widetilde{H}.\widetilde{W})\rangle\\
\texttt{upd}S^{\#}(f_n,f_m)(\widetilde{H}) &\stackrel{\text{def}}{=} \langle \widetilde{H}.N,\widetilde{H}.S[f_n(\widetilde{H})\mapsto f_m(\widetilde{H})],\widetilde{H}.V,\widetilde{H}.\widetilde{W}\rangle\\
\texttt{upd}L^{\#}(f_n,dt)(\widetilde{H}) &\stackrel{\text{def}}{=} \langle \widetilde{H}.N,\widetilde{H}.S,\widetilde{H}.V,\texttt{updFst}^{\#}(f_n(\widetilde{H}),dt[\texttt{hd}(\widetilde{H}.V(p))/p\texttt{->data}]_{p\in PVars},\widetilde{H}.\widetilde{W})\rangle
\end{aligned}$$

Fig. 6. Operators used in $\texttt{post}^{\#}(St,A_H)$.

Theorem 1. *For any k-abstract heap set A_H in $\mathcal{A}_{\mathbb{HS}}(k,\mathcal{A}_{\mathbb{W}})$, the following hold:*
– (soundness) $\texttt{post}(St,\gamma^{\mathbb{HS}}(A_H))\sqsubseteq^{\mathbb{HS}}\gamma^{\mathbb{HS}}(\texttt{post}^{\#}_k(St,A_H))$;
– (precision) if all the abstract transformers in $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{W}}}$ of the domain $\mathcal{A}_{\mathbb{W}}$ are best (exact, resp.) abstractions then $\texttt{post}^{\#}_k$ is also a best (exact, resp.) abstraction.

We define in the next section a $\mathbb{DW}$-domain for which $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{W}}}$ contains sound and best abstract transformers. [2] presents other sound transformers for $\mathbb{DW}$-domains representing sum and multiset constraints.

4 A $\mathbb{DW}$-domain with universally quantified formulas

We define the $\mathbb{DW}$-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 $\mathit{DWVar} \cup \mathit{DVar}$.

4.1 Abstract domain definition

The formulas in $A^{\mathbb{U}}$ contain a quantifier-free part and a conjunction of universally quantified formulas of the form $\forall \mathbf{y}.\ (P \Rightarrow U)$, where $\mathbf{y}$ is a vector of (integer) variables representing positions in the words, the *guard* P is a constraint over $\mathbf{y}$, and U is a quantifier-free formula over *DVar*, $\mathbf{y}$, and *DWVar*. The formula P is defined using a finite set $\mathcal{P}$ of *guard patterns* considered as a parameter of $\mathcal{A}_{\mathbb{U}}$.

Syntax of guard patterns: Let $\mathbb{O} \subseteq \mathit{DWVar}$ be a set of distinguished data word variables and $\omega_1, \ldots, \omega_n \in \mathbb{O}$. Let $\mathbf{y}_1, \ldots, \mathbf{y}_n$ be non-empty vectors of *position variables* interpreted as positions in the words denoted by $\omega_1, \ldots, \omega_n$ (these variables are universally quantified in the elements of $A^{\mathbb{U}}$). We assume that these vectors are pairwise disjoint and that $\omega_i \neq \omega_j$, for any $i \neq j$. We denote by y_i^j the jth element of the vector $\mathbf{y}_i$, $1 \leq j \leq |\mathbf{y}_i|$. Let $\Omega \subseteq \mathbb{O}$ be a set of variables not necessarily distinct from $\omega_1, \ldots, \omega_n$.

The guard patterns are conjunctions of (1) a formula that associates vectors of position variables with data word variables, (2) an arithmetical constraint on the values of some position variables, and (3) order constraints between the position variables associated with the same data word variable. Formally,

$$P(\mathbf{y}_1, \ldots, \mathbf{y}_n, \omega_1, \ldots, \omega_n, \Omega) ::= \bigwedge_{1 \leq i \leq n} \mathbf{y}_i \in \mathtt{tl}(\omega_i) \wedge P_L(y_1^1, \ldots, y_n^1, \Omega) \wedge \bigwedge_{1 \leq i \leq n} P_R^i(\mathbf{y}_i)$$

$$P_R(y^1 y^2 \ldots y^m) ::= y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m$$

where (1) for each vector $\mathbf{y}_i$, $\mathbf{y}_i \in \mathtt{tl}(\omega_i)$ states that the positions denoted by $\mathbf{y}_i$ belong to the tail of the word denoted by ω_i and that $\mathtt{len}(\omega_i) \geq |\mathbf{y_i}| + 1$; the terms $\mathtt{len}(\omega_i)$ and $\mathtt{tl}(\omega_i)$ denote the length and the tail of the word represented by ω_i, (2) P_L is a boolean combination of linear constraints over y_i^1 with $1 \leq i \leq n$; these constraints may use the terms $\mathtt{len}(\omega)$ with $\omega \in \Omega$ but we assume that P_L is not a constraint for the lengths of the words in Ω, i.e., $\wedge_{\omega \in \Omega} \mathtt{len}(\omega) > 0$ implies $\exists y_1^1, \ldots, y_n^1.\ P_L$ (in Presburger arithmetic), (3) for each vector $\mathbf{y}_i$, the formula $P_R(\mathbf{y}_i)$ is an order constraint over $\mathbf{y}_i$, where $\prec_i \in \{\leq, <, <_1\}$ with $x <_1 y$ iff $y = x + 1$.

The elements of $A^{\mathbb{U}}$: Let $\mathcal{V} \subseteq \mathit{DWVar}$ and let $\mathcal{P}$ be a set of guard patterns. We define $\mathcal{P}(\mathcal{V})$ to be the set of all formulas obtained from some $P(\mathbf{y}_1, \ldots, \mathbf{y}_n, \omega_1, \ldots, \omega_n, \Omega) \in \mathcal{P}$ by substituting each ω_i with some $w_i \in \mathit{DWVar}$, for any $1 \leq i \leq n$, and Ω with some $\mathbf{W} \subseteq \mathit{DWVar}$. We assume that $w_i \neq w_j$, for any $i \neq j$. Then, an element of $A^{\mathbb{U}}$ has the following syntax:

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

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 of the form $P(\mathbf{y}_1,\ldots,\mathbf{y}_n,w_1,\ldots,w_n,\mathbf{W})$ with $\mathbf{y} = \mathbf{y}_1 \cup \ldots \cup \mathbf{y}_n$, and U_g is a quantifier-free arithmetical formula over the terms in $E(\mathcal{V})$ together with $w[y]$ and y, for any $w \in DWVar$ and $y \in \mathbf{y}$. The terms $\mathtt{hd}(w)$, resp. $w[y]$, denote the data at the first position, resp. the position denoted by y, of the word represented by w. We assume that E and U_P are also 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 $\mathcal{A}_{\mathbb{U}}$.

Examples: The following formula is an element of $A^{\mathbb{U}}$ parametrized by $\mathcal{P} = \{y_1 \in \mathtt{tl}(\omega_1) \wedge y_2 \in \mathtt{tl}(\omega_2) \wedge y_1 = y_2\}$ and the Polyhedra domain [7]. It expresses the fact that the word denoted by w_1 is a copy of the word denoted by w_2:

$$\begin{gathered}\mathtt{len}(w_1) = \mathtt{len}(w_2) \wedge \mathtt{hd}(w_1) = \mathtt{hd}(w_2) \\ \wedge \forall y_1, y_2.\ \big((y_1 \in \mathtt{tl}(w_1) \wedge y_2 \in \mathtt{tl}(w_2) \wedge y_1 = y_2) \Rightarrow w_1[y_1] = w_2[y_2]\big)\end{gathered} \tag{H}$$

The following element of $A^{\mathbb{U}}$ over $\mathcal{P} = \{(y_1,y_2,y_3) \in \mathtt{tl}(\omega) \wedge y_1 <_1 y_2 <_1 y_3\}$ and the Polyhedra domain represents words w whose data are in the Fibonacci sequence:

$$\mathtt{hd}(w) = 1 \wedge \forall y_1,y_2,y_3.\ \big((y_1,y_2,y_3) \in \mathtt{tl}(w) \wedge y_1 <_1 y_2 <_1 y_3) \Rightarrow w[y_3] = w[y_1] + w[y_2]\big).$$

Lattice operators: The concretization function for elements in $A^{\mathbb{U}}$ is defined according to the classical semantics of these formulas. 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

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

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 $g(\mathbf{y}) = P(\mathbf{y}_1,\ldots,\mathbf{y}_n,w_1,\ldots,w_n,\mathbf{W}) \in \mathcal{P}(\mathcal{V}_1) \cup \mathcal{P}(\mathcal{V}_2)$, if for all $i \in [1,n]$, $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(w_i) \geq |\mathbf{y}_i| + 1$ then $E \sqcap^{\mathbb{Z}} U_g \sqsubseteq^{\mathbb{Z}} U'_g$. Also, $\widetilde{W} \sqcup^{\mathbb{U}} \widetilde{W}'$ is defined by $E \sqcup^{\mathbb{Z}} E' \wedge \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1) \cup \mathcal{P}(\mathcal{V}_2)} \forall \mathbf{y}.\ (g(\mathbf{y}) \Rightarrow U_g \sqcup^{\mathbb{Z}} U'_g)$. The operators $\sqcap^{\mathbb{U}}$ and $\nabla^{\mathbb{U}}$ are defined in a similar way.

In the following, we present the two most interesting abstract transformers in $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{U}}}$, $\mathtt{split}^{\#}$ and $\mathtt{concat}^{\#}$. For the sake of readability, we present these transformers only for guard patterns of the form $P(\mathbf{y}_1, \omega_1, \Omega)$, i.e., patterns with positions belonging to only one word. The general case is given in [2]. At the end of this section, we give sufficient conditions to obtain soundness and precision for all transformers in $\mathcal{F}^{\#}_{A_{\mathbb{U}}}$.

4.2 Abstract transformer $\mathtt{split}^{\#}$

Let $\widetilde{W}(\mathcal{V}) = E(\mathcal{V}) \wedge \phi(\mathcal{V}) \in A^{\mathbb{U}}$ as in (G). The transformer $\mathtt{split}^{\#}(u,v,\widetilde{W})$ splits u into its head and its tail; the head is assigned to u and the tail to v. It produces a formula $E'(\mathcal{V} \cup \{v\}) \wedge \phi'(\mathcal{V} \cup \{v\})$, where $\phi'(\mathcal{V} \cup \{v\}) = \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V} \cup \{v\})} \forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U'_g$ and:

1. E' is obtained from E by: (1) adding constraints on $\mathtt{hd}(v)$ obtained from $\phi(\mathcal{V})$, (2) substituting $\mathtt{len}(u)$ with $\mathtt{len}(v) + 1$ and assigning 1 to $\mathtt{len}(u)$,
2. ϕ' is obtained from ϕ by: (1) adding constraints on $\mathtt{tl}(v)$ computed from the constraints on $\mathtt{tl}(u)$ in ϕ, and then, (2) applying the second step from the computation of E' to the right hand side of each implication.

In the following, we detail only the important steps.

Constraints on $\mathtt{hd}(v)$**:** Let $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U_g$ be a conjunct of ϕ with $\mathbf{y} = y^1 \dots y^m$ and $g(\mathbf{y}) = \mathbf{y} \in \mathtt{tl}(u) \wedge y^1 \prec_1 y^2 \prec_2 \dots \prec_m y^m \wedge P_L(y^1, \mathbf{W})$. A constraint on $\mathtt{hd}(v)$ in $\widetilde{W}'$ is deduced from U_g when y^1 coincides with the first position in v. Formally, if the Presburger formula $y^1 \prec_1 y^2 \prec_2 \dots \prec_m y^m \wedge P_L(y^1, \mathbf{W}) \wedge E \wedge y^1 = 1$ is satisfiable then

$$E'_g = E \sqcap^{\mathbb{Z}} \left(U_g \uparrow^{\mathbb{Z}} \left(\mathbf{y} \setminus y^1\right)\right) \left[1/y^1, \mathtt{hd}(v)/u[y^1]\right]$$

is a constraint on $\mathtt{hd}(v)$. Then, E' is the meet ($\sqcap^{\mathbb{Z}}$) between all abstract values E'_g, computed as above, for every conjunct $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U_g$ of ϕ with $g(\mathbf{y}) \in \mathcal{P}(\{u\})$.

Constraints on $\mathtt{tl}(v)$**:** The formulas that constrain the tail of v are of the form $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U'_g$, where $g(\mathbf{y}) \in \mathcal{P}(\{v\})$. We compute simultaneously all U'_g with $g(\mathbf{y}) \in \mathcal{P}(\{v\})$ as follows: (1) we start with $U'_g = \top^{\mathbb{Z}}$, for every $g(\mathbf{y}) \in \mathcal{P}(\{v\})$, and (2) for every $g(\mathbf{y}) = \mathbf{y} \in \mathtt{tl}(u) \wedge y^1 \prec_1 y^2 \prec_2 \dots \prec_m y^m \wedge P_L(y^1, \mathbf{W})$ in $\mathcal{P}(\mathcal{V})$, we do the following:

- if $y^1 \prec_1 y^2 \prec_2 \dots \prec_m y^m \wedge P_L(y^1, \mathbf{W}) \wedge E \wedge y^1 > 1$ is satisfiable, then let $g' = g[v/u]$ and $U'_{g'} = U'_{g'} \sqcap^{\mathbb{Z}} U_g$;
- if $y^1 \prec_1 y^2 \prec_2 \dots \prec_m y^m \wedge P_L(y^1, \mathbf{W}) \wedge E \wedge y^1 = 1$ is satisfiable and $|\mathbf{y}| > 1$, then we try to generate a universal formula constraining $|\mathbf{y}| - 1$ positions that holds on the tail of v. Let $\mathbf{y}' = y^2 \dots y^m$ and $g'(\mathbf{y}') = \mathbf{y}' \in \mathtt{tl}(v) \wedge y^2 \prec_2 \dots \prec_m y^m$. If $g'(\mathbf{y}') \in \mathcal{P}(\{v\})$ then $U'_{g'} = U'_{g'} \sqcap^{\mathbb{Z}} U_g \uparrow^{\mathbb{Z}} \left(\{y^2, \dots, y^m\}\right)$.

4.3 Abstract transformer $\mathtt{concat}^{\#}$

Let $\widetilde{W}(\mathcal{V}) = E(\mathcal{V}) \wedge \phi(\mathcal{V}) \in A^{\mathbb{U}}$ and $V = v_1 \dots v_n$ be a vector of variables in $\mathcal{V}$. The transformer $\mathtt{concat}^{\#}(V, \widetilde{W})$ assigns to v_1 the concatenation of the words represented by the variables from V in $\widetilde{W}$ and it removes $v_2, \dots, v_n$.

Let $\alpha_1 \beta_1 \dots \alpha_m \beta_m \alpha_{m+1}$ be a decomposition of V, where β_i are maximal sub-vectors of V of length at least 2 such that for any $1 \leq i \leq m$ and $v \in \beta_i$, $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(v) = 1$ (α_1 and α_{m+1} may be empty). We define $\widetilde{W}'(\mathcal{V}') = \mathtt{concat}^{\#}(V, \widetilde{W}(\mathcal{V}))$ in two steps:

1. we concatenate the singleton words of each β_i, for $i = 1$ to n. Let $\widetilde{W}_0(\mathcal{V}_0) = \widetilde{W}(\mathcal{V})$ where $\mathcal{V}_0 = \mathcal{V}$, and for every $1 \leq i \leq m$, let $\widetilde{W}_{i+1}(\mathcal{V}_{i+1}) = \mathtt{concat}^{\#}(\beta_i, \widetilde{W}_i(\mathcal{V}_i))$ where $\mathcal{V}_{i+1} = \mathcal{V}_i \setminus \mathtt{tl}(\beta_i)$. This step generates universally quantified formulas on $\mathtt{hd}(\beta_i)$ by collecting from E the constraints on $\mathtt{hd}(v)$ with $v \in \mathtt{tl}(\beta_i)$;
2. let $\alpha' = \alpha_1 \mathtt{hd}(\beta_1) \dots \alpha_m \mathtt{hd}(\beta_m) \alpha_{m+1}$. Notice that α' does not contain two successive variables denoting singletons. We define $\widetilde{W}'(\mathcal{V}') = \mathtt{concat}^{\#}(V', \widetilde{W}_m(\mathcal{V}_m))$, where $\mathcal{V}' = \mathcal{V}_m \setminus \mathtt{tl}(\alpha')$.

The result of $\mathtt{concat}^{\#}(V, \widetilde{W}(\mathcal{V}))$ is a formula of the form $E'(\mathcal{V}') \wedge \phi'(\mathcal{V}')$, where $\phi'(\mathcal{V}') = \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}')} \forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U'_g$ and:

- E' is obtained from E by: (1) updating the length constraints, i.e. $\mathtt{len}(v_1) = \mathtt{len}(v_1) + \dots + \mathtt{len}(v_n)$, and (2) projecting out $\mathtt{hd}(v)$ and $\mathtt{len}(v)$ with $v \in V \setminus \{v_1\}$,
- ϕ' is obtained from ϕ and E by replacing each sub-formula $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U_g$ of ϕ with $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U'_g$ such that:
  1. if $g(\mathbf{y}) \notin \mathcal{P}(V)$ (g does not constrain the words denoted by V), then U'_g is obtained from U_g by applying the same transformations as for E';
  2. if $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$ then U'_g is the strongest possible constraint that we can compute from $\widetilde{W}$ which characterizes the data from the tail of v_1, knowing that v_1 represents in $\widetilde{W}'$ the concatenation of the words denoted by $v_1, \dots, v_n$ in $\widetilde{W}$.

We give hereafter the computation of the sub-formulas of ϕ' over v_1 when V is a sequence of singletons, and when V doesn't contain more than two successive singletons.
Concatenating words of length one: Let $V = v_1 \dots v_n$ be a vector of data word variables in $\mathcal{V}$ such that $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(v_i) = 1$, for all $1 \leq i \leq n$.

Let $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$ be a guard with $|\mathbf{y}| \leq n$. For every $\sigma_g : \mathbf{y} \rightarrow [2..n]$, if $g(\mathbf{y})\ \sigma_g \wedge E$ is a satisfiable Presburger formula, then let E_{σ_g} be the numerical abstract value computed from E by (1) substituting, for any i in the co-domain of σ_g, each occurrence of $\mathtt{hd}(v_i)$ with $v_1[\sigma_g^{-1}(i)]$, (2) substituting each $\mathtt{len}(v_i)$ with 1, and (3) projecting out all the terms $\mathtt{hd}(v_i)$ with i not in the co-domain of σ_g. We define U'_g as the join of all abstract values E_{σ_g} computed as above.

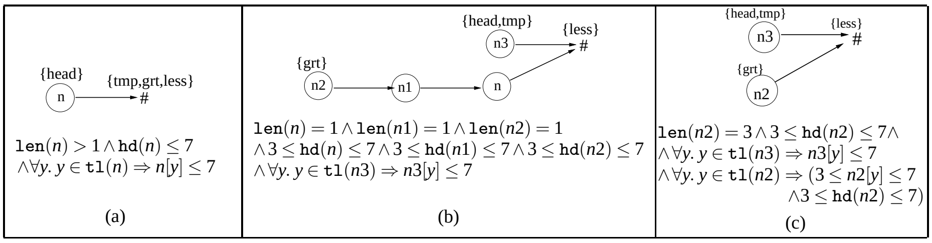

{head}
n
{tmp,grt,less}
#
$\mathtt{len}(n) > 1 \wedge \mathtt{hd}(n) \leq 7$
$\wedge \forall y.\, y \in \mathtt{tl}(n) \Rightarrow n[y] \leq 7$
(a)
{head,tmp}
n3
{less}
#
{grt}
n2
n1
n
$\mathtt{len}(n) = 1 \wedge \mathtt{len}(n1) = 1 \wedge \mathtt{len}(n2) = 1$
$\wedge 3 \leq \mathtt{hd}(n) \leq 7 \wedge 3 \leq \mathtt{hd}(n1) \leq 7 \wedge 3 \leq \mathtt{hd}(n2) \leq 7$
$\wedge \forall y.\, y \in \mathtt{tl}(n3) \Rightarrow n3[y] \leq 7$
(b)
{head,tmp}
n3
{less}
#
{grt}
n2
$\mathtt{len}(n2) = 3 \wedge 3 \leq \mathtt{hd}(n2) \leq 7 \wedge$
$\wedge \forall y.\, y \in \mathtt{tl}(n3) \Rightarrow n3[y] \leq 7$
$\wedge \forall y.\, y \in \mathtt{tl}(n2) \Rightarrow (3 \leq n2[y] \leq 7$
$\wedge 3 \leq \mathtt{hd}(n2) \leq 7)$
(c)

Fig. 7. Abstract heaps for the procedure `Dispatch3`

Example 1. Suppose that we analyse the procedure `Dispatch3` from Figure 1 using $\mathcal{A}_{\mathbb{HS}}(1, \mathcal{A}_{\mathbb{U}})$ where $\mathcal{A}_{\mathbb{U}}$ is parametrized by $\mathcal{P} = \{y \in \mathtt{tl}(\omega_1)\}$ and the Polyhedra domain. Also, suppose that the initial abstract configuration is the one from Figure 7(a). After several iterations of the loop, one of the obtained abstract heaps is pictured in Figure 7(c). It is obtained by applying $\mathtt{Normalize}_k^\#$ on the abstract heap in Figure 7(b), which calls $\mathtt{concat}^\#(n_2, n_1, n, \widetilde{W})$ where $\widetilde{W}$ is the formula in Figure 7(b). To compute U such that $\forall y.\, y \in \mathtt{tl}(n2) \Rightarrow U$ is a sub-formula of $\widetilde{W}'$ we use $\sigma_1(y) = 1$ and $\sigma_2(y) = 2$ and we define $E_{\sigma_i} = 3 \leq n2[y] \leq 7 \wedge 3 \leq \mathtt{hd}(n2) \leq 7$, for all $1 \leq i \leq 2$. Then, $U = E_{\sigma_1} \sqcup^{\mathbb{Z}} E_{\sigma_2}$.
Concatenating words of length greater than one: Let $V = v_1, \dots, v_n$ be a vector of variables from $\mathcal{V}$ s.t. there exists no $1 \leq i < n$ with $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(v_i) = 1 \wedge \mathtt{len}(v_{i+1}) = 1$.

Our aim is to generate, for every $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$, a formula of the form $\forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U'_g$ on the concatenation of $v_1,\dots,v_n$ from formulas describing in $\widetilde{W}$ properties of each of the v_is. In order to obtain non-trivial properties (i.e., U'_g is not simply $\top^{\mathbb{Z}}$) we need that the set of guard patterns $\mathcal{P}$ contains "enough" patterns to capture relations on elements within the input words $v_1,\dots,v_n$. This is ensured by considering a set of patterns denoted by $\mathtt{Closure}(P(\mathbf{y}, \omega, \Omega), \omega_1 \dots, \omega_n)$, where $P(\mathbf{y}, \omega, \Omega)$ is the pattern used to define the guard $g(\mathbf{y})$.

We give here a brief description of $\mathtt{Closure}(P(\mathbf{y}, \omega, \Omega), \omega_1 \dots, \omega_n)$ (the full definition is given in [2]). Assume that ω represents the concatenation of n words denoted by $\omega_1,\dots,\omega_n \in \mathbb{O}$. Let $\mathbf{p} : \mathbf{y} \rightarrow \mathbb{Z}$ be a valuation for the variables $\mathbf{y}$ that satisfies P and $p = \mathbf{p}(y)$, for some $y \in \mathbf{y}$. If $p < \mathtt{len}(\omega_1)$ then p is also a position of the word denoted by ω_1, if $\mathtt{len}(\omega_1) \leq p < \mathtt{len}(\omega_1) + \mathtt{len}(\omega_2)$ then $p - \mathtt{len}(\omega_1)$ is a position of ω_2, etc. In general, with any such p we can associate a position on one of the words $\omega_1,\dots,\omega_n$. Therefore, for any valuation $\mathbf{p}$ as above we define:

1. a mapping $\sigma : \mathbf{y} \to \{\mathtt{hd}(\omega_1), \mathtt{tl}(\omega_1) \ldots, \mathtt{hd}(\omega_n), \mathtt{tl}(\omega_n)\}$ s.t for any $y \in \mathbf{y}$, $\sigma(y) = \mathtt{hd}(\omega_i)$ iff $\mathbf{p}(y)$ corresponds to the first position of ω_i and $\sigma(y) = \mathtt{tl}(\omega_i)$ iff $\mathbf{p}(y)$ corresponds to a position in the tail of ω_i,
2. n valuations $\mathbf{p}_i : \sigma^{-1}(\mathtt{tl}(\omega_i)) \to \mathbb{Z}$, for any $1 \leq i \leq n$, where $\sigma^{-1}(\mathtt{tl}(\omega_i))$ is the set of variables from $\mathbf{y}$ which are mapped to $\mathtt{tl}(\omega_i)$ by σ and $\mathbf{p}_i(y)$ is the position in the tail of ω_i which corresponds to $\mathbf{p}(y)$.

Let Σ_P be the set of all mappings σ as above. Then, for every $\sigma \in \Sigma_P$, we define a set of patterns $\mathcal{T}_\sigma$ of the form $P'(\sigma^{-1}(\mathtt{tl}(\omega_i)), \omega_i, \Omega)$ with $1 \leq i \leq n$. The pattern $P'(\sigma^{-1}(\mathtt{tl}(\omega_i)), \omega_i, \Omega)$ characterizes the valuations $\mathbf{p}_i$, for any $\mathbf{p}$ a valuation for $\mathbf{y}$ corresponding to σ that satisfies P. Finally, we define $\mathtt{Closure}(P(\mathbf{y}, \omega, \Omega), \omega_1 \ldots, \omega_n)$ as the union of all the $\mathcal{T}_\sigma$ s.t. $\sigma \in \Sigma_P$. Let $\mathtt{Closure}(P, k)$ denote the union of all $\mathtt{Closure}(P, \omega_1, \ldots, \omega_n)$ with $n \leq 2k$. By Remark 1, $\mathtt{Closure}(P, k)$ is sufficient to handle any call to $\mathtt{concat}^{\#}$ made by the domain of k-abstract heap sets.

We now have all the ingredients to define the value U'_g with $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$: (1) assume that $g(\mathbf{y})$ is obtained from $P(\mathbf{y}, \omega, \Omega) \in \mathcal{P}$ by some substitution $\gamma : \mathbb{O} \to \mathit{DWVar}$, (2) let $\sigma \in \Sigma_P$, $\mathcal{T}_\sigma \in \mathtt{Closure}(P(\mathbf{y}, \omega, \Omega), \omega_1 \ldots, \omega_n)$, and $\overline{\mathcal{T}}_\sigma$ be the guards obtained from $\mathcal{T}_\sigma$ by applying γ and by substituting every ω_i with v_i, for any $1 \leq i \leq n$; we define

$$U_{\mathcal{T}_\sigma} = \textstyle\bigsqcap^{\mathbb{Z}}_{g' \in \overline{\mathcal{T}}_\sigma} U_{g'} \sqcap^{\mathbb{Z}} (E\ \theta)\!\uparrow\!\{\mathtt{hd}(v), \mathtt{len}(v) | v \in \{v_2, \ldots, v_n\}\},$$

where $U_{g'}$ is implied by g' in ϕ and θ substitutes every $\mathtt{hd}(v_i)$ for which $\sigma(y) = \mathtt{hd}(\omega_i)$, for some $y \in \mathbf{y}$, with $v_1[y]$, and (3) U'_P is the join of all $U'_{\mathcal{T}_\sigma}$ with $\sigma \in \Sigma_P$.

Example 2. Suppose that we analyse the procedure `Fibonacci` from Section 1 using the domain of 3-abstract heap sets parametrized by $\mathcal{A}_{\mathbb{U}}$ over $\mathcal{P} = \mathtt{Closure}(P, 3)$, where $P((y_1, y_2, y_3), \omega) = (y_1, y_2, y_3) \in \mathtt{tl}(\omega) \wedge y_1 <_1 y_2 <_1 y_3$, and the Polyhedra domain.

The analysis starts from an initial state in which `head` points to a non-empty list. After some iterations of the loop, we obtain an abstract heap having 7 nodes in a row, n_i, $1 \leq i \leq 6$, and # such that n_1 and n_6 are pointed by the program variables `head` and `x`, resp. We apply $\mathtt{Normalize}^{\#}_k$ which calls $\mathtt{concat}^{\#}((n_1, n_2, n_3, n_4, n_5), \widetilde{W}_0)$, where $\widetilde{W}_0$ is the formula in $\mathcal{A}_{\mathbb{U}}$ associated to this abstract heap. $\widetilde{W}_0$ is a conjunction between

$$\begin{gathered} E ::= \mathtt{len}(n_1) = 5 \wedge \mathtt{hd}(n_1) = 1 \wedge \mathtt{hd}(n_2) = 8 \wedge \mathtt{hd}(n_3) = 13 \wedge \mathtt{hd}(n_4) = 21 \\ \wedge\, \mathtt{hd}(n_5) = 34 \wedge m1 = 13 \wedge m2 = 21 \wedge \textstyle\bigwedge_{2 \leq i \leq 5} \mathtt{len}(n_i) = 1 \end{gathered}$$

and some universally-quantified formulas, including

$$\begin{gathered} \psi ::= \forall y_1, y_2, y_3.\ ((y_1, y_2, y_3) \in \mathtt{tl}(n_1) \wedge y_1 <_1 y_2 <_1 y_3) \Rightarrow (n_1[y_3] = n_1[y_1] + n_1[y_2]) \\ \wedge \forall y_1.\ ((y_1) \in \mathtt{tl}(n_1) \wedge y_1 = \mathtt{len}(n_1) - 1) \Rightarrow (n_1[y_1] = 3)\,, \end{gathered}$$

We identify $\alpha_1 = n_1$ and $\beta_1 = (n_2, n_3, n_4, n_5)$ s.t. β_1 represents only singletons and we compute $\widetilde{W}_1$ by applying $\mathtt{concat}^{\#}(\beta_1, \widetilde{W}_0)$. In $\widetilde{W}_1$, the constraints on n_1 are the same as in $\widetilde{W}_0$, the data word variables n_3, n_4, and n_5 are removed, and the universally quantified formulas over n_2 are transformed such that n_2 represents the concatenation of the singletons denoted by n_2, n_3, n_4, and n_5 in $\widetilde{W}_0$. For example, we deduce that $\psi' := \forall y_3.\ ((y_3) \in \mathtt{tl}(n_2) \wedge y_3 = 1) \Rightarrow (n_2[y_3] = 8)$.

Now we apply $\mathtt{concat}^{\#}(n_1 n_2, \widetilde{W})$. We use the fact that $\mathtt{Closure}(P, \omega_1, \omega_2)\,[n_1/\omega_1, n2/\omega_2]$ is the union of $\mathcal{T}_i$ with $1 \leq i \leq 5$ where

$$\begin{gathered} \mathcal{T}_1 = \{g_1 ::= (y_1) \in \mathtt{tl}(n_1) \wedge y_1 = \mathtt{len}(n_1) - 1,\ g_2 ::= (y_3) \in \mathtt{tl}(n_2) \wedge y_3 = 1\} \\ \mathcal{T}_2 = \{g_3 ::= (y_1, y_2) \in \mathtt{tl}(n_1) \wedge y_1 <_1 y_2 \wedge y_1 = \mathtt{len}(n_1) - 2\},\ \mathcal{T}_3 = \{g_4 ::= P\,[n_1/\omega]\}, \\ \mathcal{T}_4 = \{g_5 ::= (y_2, y_3) \in \mathtt{tl}(n_2) \wedge y_2 <_1 y_3 \wedge y_2 = 1\}, \mathcal{T}_5 = \{g_6 ::= P\,[n_2/\omega]\}. \end{gathered}$$

The procedure $\texttt{concat}^{\#}(\widetilde{W}, n_1, n_2)$ computes the value implied by the guard $P((y_1,y_2,y_3),n_1)$ in $\widetilde{W}'$ as $U'_P = \bigsqcup^{\mathbb{Z}}_{1\leq i\leq 5} U_{\mathcal{T}_i}$, where E_1 is the quantifier-free part of $\widetilde{W}_1$, $U_{\mathcal{T}_1} = U_{g_1} \sqcap^{\mathbb{Z}} U_{g_2} \sqcap^{\mathbb{Z}} E_1\,[n_1[y_2]/\texttt{hd}(n_2)]$, $U_{\mathcal{T}_2} = U_{g_3} \sqcap^{\mathbb{Z}} E_1\,[n_1[y_3]/\texttt{hd}(n_2)]$, $U_{\mathcal{T}_3} = U_{g_4}$, $U_{\mathcal{T}_4} = U_{g_5} \sqcap^{\mathbb{Z}} E_1\,[n_1[y_1]/\texttt{hd}(n_2)]$, and $U_{\mathcal{T}_5} = U_{g_6}$.

For example, from E_1, the second conjunct of ψ, and ψ', we obtain that $U_{\mathcal{T}_1} = n_1[y_1] = 3 \wedge n_1[y_2] = 5 \wedge n_1[y_3] = 8$, which describes a sub-sequence of the Fibonacci number series. Actually, every $U_{\mathcal{T}_i}$ describes a sub-sequence of this series which implies that the data from the tail of n_1 is also such a sub-sequence.

4.4 Soundness and precision

An abstract value $\widetilde{W} \in A^{\mathbb{U}}$ such that $\alpha^{\mathbb{U}}(\gamma^{\mathbb{U}}(\widetilde{W})) = \widetilde{W}$ is called *a closed abstract value*. A set of guard patterns $\mathcal{P}$ is *closed* if it equals $\texttt{Closure}(\mathcal{P}', k)$, for some $\mathcal{P}'$. A guard pattern $P(\mathbf{y_1}, \omega_1 \ldots, \mathbf{y_q}, \omega_q)$ with $\mathbf{y_i} = y_i^1 \ldots y_i^{p_i}$, for any $1 \leq i \leq q$, is called *simple* if it is of the form: $\bigwedge_{1\leq i\leq q} \mathbf{y_i} \in \texttt{tl}(\omega_i) \wedge y_i^1 \leq y_i^2 \leq \ldots \leq y_i^{p_i}$. Based on these definitions, the soundness and the precision of the abstract transformers are given by the following theorem. The precision of all the transformers except $\texttt{updFst}^{\#}$ is obtained, for example, using the Octahedra [4] or the Polyhedra domain [7].

Theorem 2. *Let $\mathcal{A}_{\mathbb{U}}$ be an abstract domain as above parametrized by a set of guard patterns $\mathcal{P}$ and by a numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$ which contains a sound projection operator and an exact meet operator (i.e., $\gamma^{\mathbb{Z}}(E \sqcap^{\mathbb{Z}} E') = \gamma^{\mathbb{Z}}(E) \cap \gamma^{\mathbb{Z}}(E')$). Then,*

- *All the abstract transformers of $\mathcal{A}_{\mathbb{U}}$ are sound.*
- *If (1) $\mathcal{P}$ is closed and it contains only simple patterns (2) the projection operator from $\mathcal{A}_{\mathbb{Z}}$ is exact, and (3) the abstract transformers in $\mathcal{A}_{\mathbb{Z}}$ corresponding to assignments $x = z_1 + \ldots + z_t$, $x = z_1 - 1$, and $x = 1$, where $x, z_1, \ldots, z_t$ are integer variables, are exact, then $\alpha^{\mathbb{U}}(F(Param, \gamma^{\mathbb{U}}(\widetilde{W}))) = F^{\#}(Param, \widetilde{W})$, for any F except* updFst*, for any set of parameters Param, and for any closed $\widetilde{W}$. Moreover, if the abstract transformer in $\mathcal{A}_{\mathbb{Z}}$ is exact for data expressions of the form dt, then* $\texttt{updFst}(x, dt, \widetilde{W})$ *is a best abstraction for any closed $\widetilde{W}$.*

The full version [2] contains a procedure that computes for any abstract value $\widetilde{W} \in A^{\mathbb{U}}$, a closed abstract value $\widetilde{W}'$ s.t. $\gamma^{\mathbb{U}}(\widetilde{W}) = \gamma^{\mathbb{U}}(\widetilde{W}')$. Moreover, we show that for simple patterns, all the abstract transformers preserve the closure property, that is, they output closed values when applied to closed values.

5 Experimental results

We have implemented the abstract reachability analysis using the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ domain in a tool called CINV[3]. Our implementation is generic in three dimensions. First, $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ is interfaced with the APRON platform [13], in order to use its fix-point computation engines; we use INTERPROC. Second, the implementations of the $\mathbb{DW}$-domains can be plugged in the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ domain. We have implemented the $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{U}}$ for the set of patterns $y \in \texttt{tl}(w)$, $(y_1, y_2) \in \texttt{tl}(w) \wedge y_1 < y_2$, and $y_1 \in \texttt{tl}(w_1) \wedge y_2 \in \texttt{tl}(w_2)$. In addition, we have implemented $\mathcal{A}_{\Sigma}$ and $\mathcal{A}_{\mathbb{M}}$ for reasoning about sums and multisets. Third, the implemented $\mathbb{DW}$-domains are generic on the numerical domain $\mathcal{A}_{\mathbb{Z}}$ used

[3] A detailed presentation is available at `http://www.liafa.jussieu.fr/cinv/`.

to represent data and length constraints. For this, we use again the APRON interface to access domains like octagons or polyhedra.

We have carried out experiments on a wide spectrum of programs including programs performing list traversal to search or to update data, programs with destructive updates and changes in the shape (e.g., list dispatch or reversal, sorting algorithms such as insertion sort), and programs computing complex arithmetical relations. Our tool was able to synthesize ordering constraints, data preservation constraints like those in (B) and (C) from Section 1, relations between data and lengths of lists, e.g. (D), and complex arithmetical relations, e.g. (F). Besides constraints which affect only one list, CINV was able to synthesize relations on data from different lists. For example, the program that creates a copy of a list, generates the post-condition given in (H) from Section 4.

Performances: Each example has been carried out in less than 1 second using 4KB to 63MB on an Intel 686 with 2GHz and 1 Go of RAM. The most expensive example is the insertion sort (with destructive updates) which takes 0.99s and 62.2MB. Traversal algorithms such as search and local update algorithms, require only few hundredths of a second, e.g., 0.02s for the maximum calculation. Properties of programs such as `Fibonacci` are generated in few tenths of seconds, e.g., 0.42s for (F).

6 Conclusion and related work

We have defined powerful invariant synthesis techniques for a significant class of programs manipulating dynamic lists with unbounded data. Future work includes (1) extending the framework to handle a wider class of data structures, e.g. doubly-linked lists, composed data structures, (2) developing heuristic techniques for automatic synthesis of the patterns used in $\mathcal{A}_{\mathbb{U}}$, and (3) defining other abstract domains for data sequences, in particular, domains based on different classes of universally quantified formulas.

Related Work: Invariant synthesis for programs with dynamic data structures has been addressed using different approaches including abstract interpretation [12, 8–11, 16–18], Craig interpolants [14], and automata-theoretic techniques [1, 3]. The contributions of our paper are (1) a generic framework for combining an abstraction for the heap with various abstraction for data sequences, (2) new abstract domains on data sequences to reason about aspects beyond the reach of the existing methods such as the sum or the multiset of all elements in a sequence, as well as a new domain for generating an expressive class of first order universal formulas, and (3) precision results of the abstract transformers for a significant class of programs. Several works [12, 8, 16] consider invariant synthesis for programs with uni-dimensional arrays of integers. These programs can be straightforwardly encoded in our framework. In [11], a synthesis technique for universally quantified formulas is presented. Our technique differs from this one by the type of user guiding information. Indeed, the quantified formulas considered in [11] are of the form $\forall \mathbf{y}.\ F_1 \Rightarrow F_2$, where F_2 must be given by the user. In contrast, our approach fixes the formulas in left hand side of the implication and synthesizes the right hand side. Therefore, the two approaches are in principle incomparable. The techniques in [12, 8] are applicable to programs with arrays. The class of invariants they can generate is included in the one handled by our approach using $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$. These techniques are based on an automatically generated finite partitioning of the array indices. We consider a larger class of programs for which these techniques can not be applied. The analysis introduced in [16] for programs with arrays can synthesize invariants on multisets of the elements in array fragments. This technique differs from ours based on the domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ by the fact that it can not be applied directly to programs with dynamic lists. Finally, the analysis in [10] 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, assuming that data have been abstracted away. Our approach is less general concerning shape properties but it is more expressive concerning properties on data.

References

1. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006.
2. A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, and M. Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. Research report 00473754, HAL, 2010.
3. M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In *Proc. of CAV*, volume 5643 of *LNCS*, pages 157–172, 2009.
4. R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proc. of SAS*, volume 3148 of *LNCS*, pages 312–327. Springer, 2004.
5. 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, 1977.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of POPL*, pages 269–282, 1979.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL*, pages 84–96, 1978.
8. D. Gopan, T.W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL*, pages 338–350, 2005.
9. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Proc. of SAS*, volume 4134 of *LNCS*, pages 240–260. Springer, 2006.
10. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *Proc. of POPL*, pages 239–251, 2009.
11. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proc. of POPL*, pages 235–246, 2008.
12. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proc. of PLDI*, pages 339–348, 2008.
13. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.
14. R. Jhala and K.L. McMillan. Array abstractions from proofs. In W. Damm and H. Hermanns, editors, *Proc. of CAV*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.
15. 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, 2005.
16. V. Perrelle and N. Halbwachs. An analysis of permutations in arrays. In *Proc. of VMCAI*, volume 5944 of *LNCS*, pages 279–294. Springer, 2010.
17. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
18. Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proc. of VMCAI*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.