Accurate Invariant Checking for Programs Manipulating Lists and Arrays with Infinite Data[⋆]

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

[1] Univ Paris Diderot, Paris Sorbone Cité, LIAFA CNRS UMR 7089, France,
{abou,cenea,sighirea}@liafa.univ-paris-diderot.fr
[2] IST Austria, cezarad@ist.ac.at

Abstract. We propose a logic-based framework for automated reasoning about sequential programs manipulating singly-linked lists and arrays with unbounded data. We introduce the logic SLAD, which allows combining shape constraints, written in a fragment of Separation Logic, with data and size constraints. We address the problem of checking the entailment between SLAD formulas, which is crucial in performing pre-post condition reasoning. Although this problem is undecidable in general for SLAD, we propose a sound and powerful procedure that is able to solve this problem for a large class of formulas, beyond the capabilities of existing techniques and tools. We prove that this procedure is complete, i.e., it is actually a decision procedure for this problem, for an important fragment of SLAD including known decidable logics. We implemented this procedure and shown its preciseness and its efficiency on a significant benchmark of formulas.

1 Introduction

Programs can manipulate dynamic data structures carrying data over infinite domains. Reasoning about the behaviors of such programs is a challenging problem due to the difficulty of representing (potentially infinite) sets of configurations, and of manipulating these representations for the analysis of the execution of program statements. For instance, pre/post-condition reasoning (checking the validity of Hoare triples) requires being able, given pre- and post-conditions ϕ and ψ, and a program statement τ, (1) to compute the strongest post-condition of executing τ starting from ϕ, denoted $\mathtt{post}(\tau,\phi)$, and (2) to check that it entails ψ. Moreover, showing that τ is executable starting from ϕ amounts in checking that $\mathtt{post}(\tau,\phi)$ is satisfiable (i.e., corresponds to a nonempty set of configurations). Therefore, an important issue is to investigate logic-based formalisms where pre/post conditions are expressible for the class of programs under interest, and for which it is possible to compute effectively (strongest) post-conditions, and to check satisfiability and entailment. (Notice that both of these problems have to be considered since it is not required that the logic is closed under negation.)

In this paper, we propose such a framework for the case of programs manipulating singly-linked lists and arrays with data. Several works have addressed this issue, proposing various decidable logics for reasoning about programs with data structures, e.g. [4, 8, 9, 13–15, 17]. Some of these logics [1, 3, 2, 9, 17] focus mainly on shape constraints assuming a bounded data domain, e.g. the Separation logic fragment in [3, 9] for which the entailment problem has a polynomial time complexity. The works in [4, 8, 13–15]

⋆ This work has been partially supported by the French ANR project Veridyc.

focus on reasoning about programs manipulating data structures with unbounded data. They introduce decidability results concerning the satisfiability problem for logics that describe the shape and the data of heap configurations. The formulas in these logics have a quantifier prefix of the form $\exists^*\forall^*$. They are different w.r.t. the type of the variables which are quantified or the basic predicates which are allowed. The validity of an entailment $\phi_1 \Rightarrow \phi_2$ is reduced to the unsatisfiability of $\phi_1 \wedge \neg\phi_2$. It can be performed only between boolean combinations of formulas with quantifier prefixes $\exists^*$ or $\forall^*$.

We define a logic, called SLAD, allowing to express the specifications of all common programs manipulating lists and arrays. This logic combines shape constraints, written in Separation Logic [17], and universally-quantified first-order formulas expressing constraints on the size and the values of the data sequences associated with arrays or sharing-free list segments in the heap. The logic is parametrized by a logic on data; for simplicity, we suppose that data is of integer type and that the logic on data is Presburger arithmetics. The Separation logic formulas are in a fragment that extends the one in [3, 9] with existential quantification and disjunction. Existential quantification is useful to describe for instance lasso-shaped lists, and disjunction is crucial for specifications that involve data. For example, the post-condition of a loop that searches an integer *val* in a list pointed to by *x*, where the variable *xi* is used to traverse the list, states that: either *val* is not in the list, e.g., $xi = \texttt{NULL}$, or *val* is in the list and *xi* points to the corresponding element, e.g., $xi \neq \texttt{NULL}$, *xi* is reachable from *x*, and $xi \rightarrow data = val$.

In general, SLAD formulas have quantifier prefixes of the form $\exists^*\forall^*$ (e.g., the formula in Fig.1 that describes a sorted lasso-shaped list). The validity of entailments between such formulas can not be reduced to the satisfiability of an $\exists^*\forall^*$ formula and thus, it can not be decided using approaches like in [4, 8, 13–15]. Also, in many cases, relevant program assertions are beyond the identified decidable fragments (e.g., relations between the values of the data fields and the size of the allocated list). We define a procedure for checking entailments between SLAD formulas, which exploits their syntax. The entailment between shape constraints is checked using a slightly modified version of the decision procedure for Separation Logic in [9]. If this entailment holds, the procedure reduces the entailment between the data constraints to the validity of a formula in the data logic. The novelty of this procedure is that (1) it does not reduce entailment checking in SLAD to satisfiability checking in the same logic, (2) it is applied to $\exists^*\forall^*$ formulas, and (3) it is sound when applied to *any* SLAD formulas and complete for a relevant fragment of SLAD. Note that the same procedure is also a sound decision procedure for unsatisfiability. The fragment of SLAD, called $\mathsf{SLAD}_{\leq}$, for which the unsatisfiability check is complete includes for instance, the logic APF [8] and the restriction of LISBQ [14] to singly-linked lists. The decision procedure for $\mathsf{SLAD}_{\leq}$ has the same complexity as the decision procedures in [8, 14] (NP-complete, if we fix the number of universal quantifiers). The entailment problems for which our procedure is complete consider $\mathsf{SLAD}_{\leq}$ formulas (satisfying some syntactic restrictions) and are beyond the scope of all existing decision procedures that we are aware of.

Besides decidability results, we show that our approach deals efficiently with a variety of examples, including programs whose specifications are given by SLAD formulas that are not in $\mathsf{SLAD}_{\leq}$ and which can't be handled by existing tools. This is an important feature of our approach: it provides uniform and efficient techniques that are applicable to a large class of formulas, and that are complete for a significant set of formulas.

For the simplicity of the exposition, we begin by defining SLD, a logic on singly-linked lists, and then, in Section 6, we define its extension to arrays, SLAD.

2 Logic SLD, a logic for programs with singly-linked lists

We introduce hereafter the class of programs with singly-linked lists considered in this paper and the syntax of *Singly-linked list with Data Logic* (SLD, for short), a logic to describe sets of program configurations (we relegate the definition of the formal semantics to the long version [5]). Then, we introduce fragments of SLD relevant for the results presented in the following sections. Finally, we give the properties of SLD relevant for program verification. In the following, *PVar* and *DVar* are disjoint sets of pointer resp. integer program variables. `NULL` is a distinguished pointer variable in *PVar*.

2.1 Programs with singly-linked lists

We consider sequential programs manipulating singly-linked lists of the same type, i.e., pointer to a record called `list` formed of one recursive pointer field `next` and one data field `data` of integer type. However, the results presented in this paper work also for *lists with multiple data fields of different types*. As usual, the allocated memory (heap) is represented by a directed labeled graph where nodes represent list cells and edges represent values of the field `next`. The constant `NULL` is represented by a distinguished node $\sharp$ with no output edge. Nodes are labeled with values of the field `data` and pointer variables. For example, Fig. $1(a)$ represents a heap containing a lasso-shaped list pointed to by x. Formally, a program configuration consists of a directed graph representing the heap and a valuation of the integer program variables in *DVar*.

Definition 1 (Program configuration). *A* program configuration *is a tuple* $H = (V, E_n, \ell_P, \ell_D, D)$*, where (1)* V *is a finite set of nodes containing a distinguished node* $\sharp$*, (2)* $E_n : V \rightharpoonup V$ *is a partial function s.t.* $E_n(\sharp)$ *is undefined, (3)* $\ell_P : PVar \rightharpoonup V$ *is a partial function labeling nodes with pointer variables such that* $\ell_P(\mathtt{NULL}) = \sharp$*, (4)* $\ell_D : (V \setminus \{\sharp\}) \to \mathbb{Z}$ *is a function labeling nodes with integers, and (5)* $D : DVar \to \mathbb{Z}$ *is a valuation of the integer variables.* □

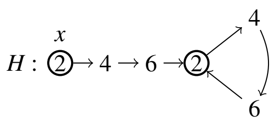
Definition 2 (Simple/Crucial node). *A node labeled with a pointer variable or which has at least 2 predecessors is called* crucial*. Otherwise, it's called a* simple node*.* □

For example, in the program configuration from Fig. 1(a) (*DVar* is empty) the circled nodes are crucial nodes and the other nodes are simple.

2.2 Syntax of SLD

The main features of SLD are introduced through an example. (A detailed presentation of SLD is given [5]). The heap in Fig. $1(a)$ consists of a lasso shaped list whose cyclic part is equal in size and values of the data fields to the non-cyclic part. The data in the cyclic part is strictly sorted. These properties are expressed in SLD as follows:
Shape formulas: The shape of the heap is characterized using the formula φ_S^1 in Fig. $1(b)$, written in a fragment of Separation logic (SL) [17], where (1) n and m are *node variables* interpreted as nodes in the heap, (2) $\mathtt{ls}(n,m)$ denotes a *possibly-empty path* between the nodes denoted by n and m; such a path is called a *list segment*, (3) the *separating conjunction* $*$ expresses the fact that the two list segments are disjoint except for their end nodes, and (4) $x(n)$ says that the pointer variable x labels n.

Let *NVar* be a set of *node variables* interpreted as nodes in program configurations. The syntax of shape formulas is given in Fig. 2. The node $\sharp$ is represented by a constant

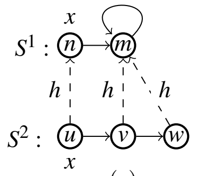


(a)

$$\varphi_S^1 := \texttt{ls}(n,m) * \texttt{ls}(m,m) \wedge x(n)$$

$$\varphi_S^2 := \texttt{ls}(u,v) * \texttt{ls}(v,w) \wedge x(u)$$

(b)



(c)

$$\begin{aligned}\varphi_D^1 := {} & \texttt{dt}(n) = \texttt{dt}(m) \wedge \texttt{len}(n) = \texttt{len}(m) \\ & \wedge \forall y, y'.\ \big(\underbrace{0 < y < \texttt{len}(n) \wedge 0 < y' < \texttt{len}(m) \wedge y = y'}_{G_{m,n}(y,y')}\big) \Rightarrow n[y] = m[y'] \qquad \text{list equality} \\ & \wedge \forall y_1, y_2.\ \underbrace{0 < y_1 < y_2 < \texttt{len}(m)}_{G_m(y_1,y_2)} \Rightarrow \texttt{dt}(n) = \texttt{dt}(m) < m[y_1] < m[y_2] \quad \text{strict sortedness} \\ \varphi_D^2 := {} & \underbrace{\forall y_1, y_2.\ 0 < y_1 < y_2 < \texttt{len}(v) \Rightarrow \texttt{dt}(v) \leq v[y_1] \leq v[y_2]}_{sorted(v)} \qquad \text{sortedness}\end{aligned}$$

(d)

Fig. 1: A program configuration (a) specified by two SLD formulas $\varphi^1 := \exists n, m.\ (\varphi_S^1 \wedge \varphi_D^1)$ and $\varphi^2 := \exists u, v, w.\ (\varphi_S^2 \wedge \varphi_D^2)$ given in (b) and (d) such that $\varphi^1 \vdash \varphi^2$. In (c), S^1 and S^2 are homomorphic SL-graphs representing the SLD graph formulas φ_S^1 resp. φ_S^2.

with the same name in the syntax. For simplicity, we consider the intuitionistic model of Separation logic [17]: if a formula is true on a graph then it remains true for any extension of that graph with more nodes. Our techniques can be adapted to work also for the non-intuitionistic model. Inequalities are important to express properties like list disjointness. For example, the formula $\texttt{ls}(n,u) * \texttt{ls}(m,v) \wedge\ x(n) \wedge z(m)$ has as model a heap with only one list when m, u, v are interpreted in the same node, while $\texttt{ls}(n,u) * \texttt{ls}(m,v) \wedge\ x(n) \wedge z(m) \wedge\ n \neq m \wedge\ u \neq v$ specifies models that contain two disjoint lists.

The restriction `Det` from Fig. 2 and the omission of the "points to" predicate $u \mapsto v$ from Separation logic [9] (which denotes the fact that v is the successor of the node u) are adopted only for simplicity.

$x \in PVar$ program pointer variables	$n, m \in NVar$ node variables $u, v \in NVar \cup \{\sharp\}$ node variables or $\sharp$
$\varphi_E ::= \texttt{ls}(n,u) \mid \varphi_E * \varphi_E$ $\varphi_P ::= x(u) \mid m \neq u \mid \varphi_P \wedge \varphi_P$ $\varphi_S ::= \varphi_E \wedge \varphi_P$, where φ_E satisfies `Det`	`Det` : "φ_E does not contain two predicates $\texttt{ls}(n,u)$ and $\texttt{ls}(n,v)$ where n, u, v are pairwise distinct."

Fig. 2: Syntax of **shape formulas**.

To simplify the presentation, a shape formula is represented by an SL-graph, which is a slight adaptation of the notion introduced in [9]. Each node of an SL-graph corresponds to a node variable, each edge corresponds to an `ls` predicate, and nodes are labeled by pointer variables. For example, the graphs S^1 and S^2 in Fig. $1(c)$ are the SL-graphs associated to the formulas φ_S^1 resp. φ_S^2 in Fig. $1(b)$.

Definition 3 (SL-graphs). *The* SL-graph *associated to a shape formula φ_S is either $\perp$ or a graph $S = (N \cup \{\sharp\}, E_\ell, \ell_P, E_d)$, where N is the set of node variables in φ_S, $E_\ell : N \rightharpoonup N \cup \{\sharp\}$ defines a set of directed edges by $E_\ell(n) = m$ iff $\texttt{ls}(n,m)$ appears in φ_S, $\ell_P : PVar \rightharpoonup N \cup \{\sharp\}$ is defined by $\ell_P(x) = u$ iff $x(u)$ appears in φ_S, and $E_d \subseteq N \times N$*

is a disequality relation which defines a set of undirected edges such that $(n,u) \in E_d$ *iff* $n \neq u$ *appears in* φ_S. *A* path *in an SL-graph is formed only of directed edges.*

The SL-graph $\bot$ represents unsatisfiable shape formulas for which an SL-graph can not be built. For an SL-graph $S \neq \bot$, we use superscripts to denote their components, e.g., E^S_ℓ, and we note $N^S_\sharp$ for $N^S \cup \{\sharp\}$. Also, $Vars^S(n)$ denotes the set of pointer variables labeling n, $\{x \in PVar \mid \ell^S_P(x) = n\}$, and $Vars(S)$ denotes the set of pointer variables in S, $\mathtt{dom}(\ell_P)$. The notions of simple and crucial node are defined similarly for SL-graphs.
Sequence formulas: Consider again the formula φ^1_S in Fig. $1(b)$. The size and the data values of the list segments specified by φ^1_S are characterized by φ^1_D in Fig. $1(d)$, which is a first-order formula over integer sequences. The equality between the list segments corresponding to $\mathtt{ls}(n,m)$ and $\mathtt{ls}(m,m)$ is stated using (1) $\mathtt{len}(n) = \mathtt{len}(m)$, where $\mathtt{len}(n)$ ($\mathtt{len}(m)$) denotes the length, i.e., the number of edges, of the list segment associated to $\mathtt{ls}(n,m)$ ($\mathtt{ls}(m,m)$), (2) $\mathtt{dt}(n) = \mathtt{dt}(m)$, where $\mathtt{dt}(n)$ represents the integer labeling the node n, and (3) a universally quantified formula of the form $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$, where the variables in the set $\mathbf{y}$, called *position variables*, are interpreted as integers and $n[y]$ is interpreted as the integer labeling the node at distance y from n.

For every predicate $\mathtt{ls}(n,m)$ we call *integer sequence associated with* n, for short *sequence of* n, the element of $\mathbb{Z}^*$ obtained by concatenating the integers labeling all the nodes except the last one (i.e., the one represented by m) in the list segment corresponding to $\mathtt{ls}(n,m)$. A term $n[y]$ appears in $U(\mathbf{y})$ only if the guard $G(\mathbf{y})$ contains the constraint $0 < y < \mathtt{len}(n)$. This restriction is used to avoid undefined terms. (For instance, if the length of the list segment starting in n equals 2 then the term $n[y]$ with y interpreted as 3 is undefined.) The strict sortedness is specified using a universal formula of the same form. Intuitively, $U(\mathbf{y})$ constrains the integers labeling a set of nodes determined by the guard $G(\mathbf{y})$. A formula $\rho = \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ is called a *guarded formula* and $G(\mathbf{y})$ is the *guard* of ρ. The syntax of sequence formulas is given in Fig. 3.

$N \cup \{n, n_y\} \subseteq NVar$ node variables $\qquad d \in DVar$ integer variable
$\mathbf{y} \cup \{y\} \subseteq Pos$ position variables $\qquad k \in \mathbb{Z}$ integer constant

Position terms: $p ::= k \mid y \mid \mathtt{len}(n) \mid p + p$
E-terms: $e ::= k \mid d \mid \mathtt{dt}(n) \mid \mathtt{len}(n) \mid e + e$
U-terms: $t ::= e \mid y \mid n[y] \mid t + t$

Existential constraints: $E ::= e \leq e' \mid \neg E \mid E \wedge E \mid \exists d.\ E$, where e and e' are E-terms
Constraints on positions: $C ::= p \leq p' \mid \neg C \mid C \wedge C$ where p and p' are position terms
Guards: $G(\mathbf{y}) ::= C \wedge \bigwedge_{y \in \mathbf{y}} 0 < y < \mathtt{len}(n_y)$,
Data properties: $U(\mathbf{y}) ::= t \leq t' \mid \neg U \mid U \wedge U \mid \exists d.\ U$, where t and t' are U-terms containing position variables from $\mathbf{y}$
Sequence formulas: $\varphi_D ::= E \mid \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y}) \mid \varphi_D \wedge \varphi_D$

Fig. 3: Syntax of **sequence formulas**.

SLD formulas: A formula in SLD is a disjunction of formulas of the form $\exists N_1.\ (\varphi^1_S \wedge \varphi^1_D) * \cdots * \exists N_k.\ (\varphi^k_S \wedge \varphi^k_D)$, where each N_i is the set of all node variables in φ^i_S (which include all the node variables in φ^i_D). Note that such a formula is equivalent to $\exists N_1 \cup \cdots \cup N_k.\ (\varphi^1_S * \cdots * \varphi^k_S \wedge \varphi^1_D \wedge \cdots \wedge \varphi^k_D)$. W.l.o.g, in the following, we will consider only SLD formulas which are disjunctions of formulas of the form $\exists N.\ \varphi_S \wedge \varphi_D$.

The set of program configurations which are models of a formula ψ is denoted $[\psi]$.

2.3 Fragments of SLD

Succinct SLD formulas: An SLD formula ψ is *succinct* if every SL-graph associated to a disjunct of ψ has no simple nodes. For example, $\varphi^1 := \exists n,m.\ \varphi^1_S \wedge \varphi^1_D$ in Fig. 1 is succinct, but the formula φ^5 whose SL-graph is given in the top of Fig. 4 is not succinct.
SLD$_\leq$ formulas: A guard $G(\mathbf{y})$ is called a $\leq$-*guard* if it has the following syntax:

$$\bigwedge_{1\leq j\leq q} p_j \leq p'_j \wedge \bigwedge_{y\in\mathbf{y}} 0 < y < \mathtt{len}(n_y), \tag{i}$$

where p_j and p'_j are either position variables or position terms that do not contain position variables. That is, a $\leq$-guard contains only inequalities of the form $y_1 \leq y_2$, $y_1 \leq p$, $p \leq y_1$, or $p \leq p'$, where $y_1, y_2 \in Pos$ and p, p' are position terms without position variables. Thus, a $\leq$-guard can define only ordering or equality constraints between positions variables in one or several sequences; it can not define, e.g., the successor relation between positions variables. The fragment **SLD$_\leq$** is the set of all SLD formulas ψ such that for any sub-formula $\forall\mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ of φ, (1) $G(\mathbf{y})$ is a $\leq$-guard and (2) any occurrence of a position variable y in $U(\mathbf{y})$ belongs to a term $n[y]$ with $n \in NVar$.

2.4 Closure under post image computation

For any program statement St and any set of program configurations $\mathbb{H}$, $\mathtt{post}(St,\mathbb{H})$ denotes the postcondition operator. The closure of SLD under the computation of the strongest postcondition w.r.t. basic statements (which don't contain "while" loops and deallocation statements) is stated in the following theorem (the proof is given in [5]).

Theorem 1. *Let St be a basic statement and ψ an SLD formula. Then, $\mathtt{post}(St,[\psi])$ is SLD-definable and it can be computed in linear time. Moreover, if ψ is an SLD$_\leq$ formula then $\mathtt{post}(St,[\psi])$ is SLD$_\leq$-definable and it can be computed in linear time.*

3 Checking entailments between SLD formulas

For any ψ_1 and ψ_2 two SLD formulas, ψ_1 *semantically entails* ψ_2 (denoted $\psi_1 \vdash \psi_2$) iff $[\psi_1] \subseteq [\psi_2]$. The following result states that checking the semantic entailment between SLD formulas is undecidable. It is implied by the fact that even the satisfiability problem for SLD is undecidable (by a reduction to the halting problem of 2-counter machines).

Theorem 2. *The satisfiability problem for SLD is undecidable. The problem of checking the semantic entailment between (succinct) SLD formulas is also undecidable.*

We present a procedure for checking entailments between SLD formulas, called *simple syntactic entailment* and denoted $\sqsubseteq_S$, which in general is only sound.
Checking entailments of shape formulas: For SLD formulas without data constraints (i.e., disjunctions of shape formulas), the simple syntactic entailment is a slight extension to disjunctions and existential quantification of the decision procedure for Separation logic introduced in [9]. Thus, given two shape formulas φ and φ', $\varphi \sqsubseteq_S \varphi'$ iff the SL-graph of φ is $\perp$ or there exists an homomorphism from the SL-graph of φ' to the SL-graph of φ. This homomorphism preserves the labeling with program variables, the edges that denote inequalities, and it maps edges of φ' to (possibly empty paths) of φ such that any two disjoint edges of φ' are mapped to two disjoint paths of φ. For example, the dotted edges Fig. $1(c)$ represent the homomorphism h which proves that $\exists n,m.\ \varphi^1_S \vdash \exists u,v,w.\ \varphi^2_S$. This holds because v is not required to be different from w. We have that $\varphi \vdash \varphi'$ iff $\varphi \sqsubseteq_S \varphi'$. Formally, the homomorphism between SL-graphs is defined as follows:

Definition 4 (Homomorphic shape formulas). *Given two SL-graphs S_1 and S_2, S_1 is* homomorphic *to S_2, denoted by $S_1 \mapsto_h S_2$, if $S_1 = S_2 = \bot$ or there exists a function $h : N_+^{S_1} \to N_+^{S_2}$, called homomorphism, such that: (1) $h(\sharp) = \sharp$; (2) for any $n \in N_+^{S_1}$, $Vars^{S_1}(n) \subseteq Vars^{S_2}(h(n))$; (3) for any $e = (n,u) \in E_\ell^{S_1}$, there is a (possibly empty) path π_e in S_2 starting in $h(n)$ and ending in $h(u)$; (4) for any two distinct edges $e_1 = (n,u) \in E_\ell^{S_1}$ and $e_2 = (m,v) \in E_\ell^{S_1}(m,v)$, the corresponding paths π_{e_1} and π_{e_2} associated by h in S_2 don't share any edge; (5) for any $e = (n,u) \in E_d^{S_1}$, $(h(n),h(u)) \in E_d^{S_2}$.* □

For any two SLD formulas ψ and ψ', which are disjunctions of shape formulas, $\psi \sqsubseteq_S \psi'$ iff for any disjunct φ of ψ there exists a disjunct φ' of ψ' such that $\varphi \sqsubseteq_S \varphi'$. One can prove that $\sqsubseteq_S$ is sound, i.e., for any ψ and ψ', if $\psi \sqsubseteq_S \psi'$ then $\psi \vdash \psi'$.

Adding data constraints: For SLD formulas with data constraints, the definition of the simple syntactic entailment is guided by the syntax of SLD. We illustrate it on the formulas from Fig. $1(b),(d)$. First, the procedure checks if the simple syntactic entailment holds between the SL-formulas φ_S^1 and φ_S^2. Then, because the homomorphism in Fig. $1(c)$ maps every edge in S^2 to an edge in S^1, it checks that φ_D^1 entails $\varphi_D^2[h]$, where $\varphi_D^2[h]$ is obtained from φ_D^2 by applying the substitution $[u \mapsto n, v \mapsto m, w \mapsto m]$ defined by the homomorphism h (if the homomorphism h does not satisfy this condition then the simple syntactic entailment does not hold). The entailment between two sequence formulas φ_D and φ_D' is reduced to the entailment in the logic on data by checking that for any guarded formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U'(\mathbf{y})$ in φ_D' there exists a guarded formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ in φ_D such that $U(\mathbf{y}) \Rightarrow U'(\mathbf{y})$. *This entailment check between sequence formulas is also denoted by $\sqsubseteq_S$.* In Fig. $1(d)$, this test is satisfied for the guarded formula in $\varphi_D^2[h]$ by the last guarded formula in φ_D^1 (i.e., strict sortedness implies sortedness). This procedure is efficient because it requires no transformation on the input formulas and the decision procedure on data is applied on small instances and a number of times which is linear in the size of the input formulas.

The simple syntactic entailment for disjunctions of formulas of the form $\exists N.\ \varphi_S \wedge \varphi_D$ is defined as in the case of disjunctions of shape formulas. Clearly, $\sqsubseteq_S$ is only sound. In the following, we will introduce a more precise procedure for checking entailments between SLD formulas, denoted $\sqsubseteq$ and called *syntactic entailment*. The presentation is done in two steps depending on the class of homomorphisms discovered while proving the entailment between shape formulas. First, we will consider *edge homomorphisms*, that map edges of an SL-graph to edges of another SL-graph (i.e., for any edge (u,v), $(h(u),h(v))$ is an edge) and then, we will consider the case of arbitrary homomorphisms.

4 Syntactic entailment w.r.t. edge homomorphisms

The simple syntactic entailment fails to prove some relevant entailments encountered in practice, e.g, the equality of two lists pointed to by x and z, resp., and the fact that the list pointed to by z is sorted implies that the list pointed to by x is also sorted:

$$\exists n,m.\ (\mathtt{ls}(n,\sharp) * \mathtt{ls}(m,\sharp) \wedge x(n) \wedge z(m) \wedge \varphi_D^1) \vdash \exists u.\ \big(\mathtt{ls}(u,\sharp) \wedge x(u) \wedge sorted(u)\big). \quad \text{(ii)}$$

Above, φ_D^1 is the formula in Fig. $1(c)$ and the entailment between the shape formulas is proven by the homomorphism h defined by $h(u) = n$. Checking the entailment between φ_D^1 and $sorted(u)[h]$ fails because the sets of guards in the two formulas are different.

More precisely, φ_D^1 does not contain a guarded formula of the form $\forall y_1, y_2.\ G_n(y_1, y_2) \Rightarrow U$, where $G_n(y_1, y_2) := 0 < y_1 < y_2 < \mathtt{len}(n)$.

Saturation procedure: The problem is that SLD does not have a normal form in the sense that the same property can be expressed using SLD formulas over different sets of guards. In our example, one can add to φ_D^1 the guarded formula $\rho := \forall y_1, y_2.\ 0 < y_1 < y_2 < \mathtt{len}(n) \Rightarrow \mathtt{dt}(n) < n[y_1] < n[y_2]$ while preserving the same set of models. Adding this guarded formula makes explicit the constraints on the integer values in the list segment starting in n, which are otherwise implicit in φ_D^1. If all constraints were explicit then, the simple syntactic entailment would succeed in proving the entailment.

Based on these remarks, we extend the simple syntactic entailment such that before applying the syntactic check between two sequence formulas φ_D and φ'_D presented above, we apply a saturation procedure to the SLD formula in the left hand side of the entailment, called `saturate`. This procedure makes explicit in φ_D all the properties expressed with guards that appear in φ'_D. For example, by applying this procedure the formula ρ is added to φ_1. More precisely, we add to φ_D a trivial formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow \mathit{true}$, for every guard in φ'_D, and then, we call `saturate` which strengthens every guarded formula in φ_D. Roughly, the strengthening of $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U_G(\mathbf{y})$ relies on the following principle: to find a formula U such that $G \Rightarrow U$ is implied by $E \wedge (G_1 \Rightarrow U_1) \wedge \ldots \wedge (G_k \Rightarrow U_k)$, one has to find a (negation-free) boolean combination $\mathbb{C}[G_1, \ldots, G_k]$ of $G_1, \ldots, G_k$ such that $(E \wedge G) \Rightarrow \mathbb{C}[G_1, \ldots, G_k]$, and then set U to $\mathbb{C}[U_1, \ldots, U_k]$. This principle is extended to boolean combinations of guards where some position variables are existentially-quantified (see [5] for more details). Going back to the example in (ii), we add to φ_D^1 the formula $\rho_0 := \forall y_1, y_2.\ G_n(y_1, y_2) \Rightarrow \mathit{true}$. Then, following the principle described above, we have that

$$\big(\mathtt{len}(n) = \mathtt{len}(m) \wedge G_n(y_1, y_2)\big) \Rightarrow \exists y'_1, y'_2.\ \big(G_m(y'_1, y'_2) \wedge G_{m,n}(y_1, y'_1) \wedge G_{m,n}(y_2, y'_2)\big),$$

where G_m and $G_{m,n}$ are the guards from φ_D^1 given in Fig. 1(d). Therefore, the right part of the implication in ρ_0 can be replaced by

$$\exists y'_1, y'_2.\ \big(\mathtt{dt}(n) = \mathtt{dt}(m) < m[y'_1] < m[y'_2] \wedge m[y'_1] = n[y_1] \wedge m[y'_2] = n[y_2]\big),$$

which is equivalent to the right hand side of ρ, $\mathtt{dt}(n) < n[y_1] < n[y_2]$.

Correctness and precision results: The next result shows that the saturation procedure returns a formula equivalent to the input one and that, for the fragment $\mathsf{SLD}_{\leq}$ of SLD, `saturate` computes the strongest guarded formulas which are implied by the input formula. The precision result holds because for the class of $\leq$-guards, it suffices to reason only with representatives for the set of tuples of positions satisfying some guard.

Theorem 3. *Let $\varphi = \exists N.\ \varphi_S \wedge \varphi_D$ be a disjunction-free SLD formula. Then,* `saturate`(φ) *is equivalent to φ and* `saturate`$(\varphi) \sqsubseteq_S \varphi$. *Moreover, for any $\mathsf{SLD}_{\leq}$ formula φ,* `saturate`$(\varphi) = \exists N.\ \varphi_S \wedge \varphi'_D$ *such that the following hold:*

- *the existential constraint of φ'_D, E', is the strongest existential constraint such that $\varphi \vdash (\exists N.\ \varphi_S \wedge E')$, and*
- *for any guard $G(\mathbf{y})$ in φ_D, φ'_D contains the strongest universal formula of the form $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ such that $\varphi \vdash (\exists N.\ \varphi_S \wedge \forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y}))$.* □

This procedure will be used to define a sound and complete decision procedure for the satisfiability of $\mathsf{SLD}_{\leq}$ and a sound and complete decision procedure for checking entailments between formulas in a fragment of $\mathsf{SLD}_{\leq}$ (see Th. 6 for more details).

5 Syntactic entailment w.r.t arbitrary homomorphisms

Suppose that we want to check the entailment between two SLD formulas $\varphi = \exists N.\ \varphi_S \land \varphi_D$ and $\varphi' = \exists N'.\ \varphi'_S \land \varphi'_D$ and that h is an homomorphism that is a witness for the fact that $\varphi_S \vdash \varphi'_S$. If h is not an edge homomorphism then, when proving the entailment between φ_D and $\varphi'_D[h]$, one encounters two difficulties: (1) edges of φ'_S mapped to nodes of φ_S (i.e., edges (u,v) such that $h(u) = h(v)$) and (2) edges of φ'_S mapped to paths in φ_S containing at least two edges (i.e., edges (u,v) such that the nodes $h(u)$ and $h(v)$ are connected by a path of length at least 2).

Procedure `split`**:** In the first case, the edges of φ'_S mapped to nodes of φ_S pose the following problem: the sequence formula $\varphi'_D[h]$ may contain guarded formulas that describe list segments that don't have a correspondent in φ_D. For example, let

$$\begin{aligned}
\varphi^3 &:= \exists n.\ x(n) \quad \land \qquad\qquad \mathtt{dt}(n) = 3 \text{ and}\\
\varphi^4 &:= \exists u,v.\ \mathtt{ls}(u,v) \land x(u) \land\ \mathtt{dt}(u) \geq 2 \land \forall y.\ 0 < y < \mathtt{len}(u) \Rightarrow u[y] \geq 1
\end{aligned}$$

Note that $\varphi^3 \vdash \varphi^4$ and that there exists an homomorphism h between the shape formula of φ^4 and the shape formula of φ^3 given by $h(u) = h(v) = n$. In order to be able to use the same approach as before (i.e., applying `saturate` on φ^3 and then checking the entailment between guarded formulas with similar guards), we define a procedure `split` that transforms the formula φ^3 such that the homomorphism h becomes injective. That is, `split` transforms φ^3 into:

$$\overline{\varphi^3} := \exists n, nn.\ x(n) \land \mathtt{ls}(n, nn) \land \mathtt{dt}(n) = 3 \land \mathtt{len}(n) = 0,$$

where the new node variable nn is added such that $h'(u) = n$ and $h'(v) = nn$ is an injective homomorphism. Note that the two formulas φ^3 and $\overline{\varphi^3}$ are equivalent. Now, as described in the previous section, we can add the trivial formula $\forall y.\ 0 < y < \mathtt{len}(n) \Rightarrow$ *true* to $\overline{\varphi^3}$ and then apply `saturate`, which strengthens it into $\forall y.\ 0 < y < \mathtt{len}(n) \Rightarrow$ *false* because the list segment starting in n is empty. Now, $\mathtt{dt}(n) = 3 \Rightarrow dt(n) \geq 2$ and *false* $\Rightarrow n[y] \geq 1$ which is enough to prove that $\varphi^3 \vdash \varphi^4$.

If the SL-graph of φ_S and φ'_S do not contain cycles then `split` returns a triple $(\overline{\varphi}, h', \varphi')$. The formula $\overline{\varphi}$ is obtained by (1) adding to φ_S a new node variable denoted nn for every two node variables n and m in φ'_S such that $h(n) = h(m)$, (2) placing nn between $h(n)$ and its successor in φ_S (i.e., $\mathtt{ls}(h(n), x)$ is replaced by $\mathtt{ls}(h(n), nn) * \mathtt{ls}(nn, x)$) (3) substituting $h(n)$ with nn in the sequence formula φ_D (in this way, all constraints on $h(n)$ are transferred to the node nn) (4) adding the constraint that the length of the list segment starting in $h(n)$ is 0. The homomorphism h' is injective and it is defined like h except for n and m where $h'(n) = h(n)$ and $h'(m) = nn$. In the general case, $\mathtt{split}(\varphi, h, \varphi')$ returns a set of triples $(\overline{\varphi}, h', \overline{\varphi'})$ with h' an injective homomorphism between the shape formula of $\overline{\varphi'}$ and the shape formula of $\overline{\varphi}$ ($\overline{\varphi}$ is an over-approximation of φ and $\overline{\varphi'}$ is an under-approximation of φ'). We have that $\varphi \vdash \varphi'$ iff $\overline{\varphi} \vdash \overline{\varphi'}$, for some triple $(\overline{\varphi}, h, \overline{\varphi'}) \in \mathtt{split}(\varphi, h, \varphi')$ (see [5] for a complete definition of `split`).

Procedure `fold`**:** The second case is illustrated on the entailment $\varphi^5 \vdash \varphi^6$, where

$$\begin{aligned}
\varphi^5 &:= \exists n, m, p.\ \left(\varphi^5_S \land sorted(n) \land sorted(m) \land \forall y.\ 0 < y < \mathtt{len}(n) \Rightarrow n[y] \leq \mathtt{dt}(m)\right)\\
\varphi^6 &:= \exists u, v.\ \left(\varphi^6_S \land sorted(u)\right)
\end{aligned}$$

and the graph formulas φ^5_S and φ^6_S are given by the SL-graphs S^5 and S^6 in Fig. 4.

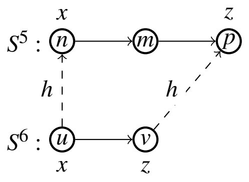

$$\begin{tikzcd} S^5: \overset{x}{n} \arrow[r] & m \arrow[r] & \overset{z}{p} \\ S^6: \underset{x}{u} \arrow[u, dashed, "h"] \arrow[r] & \underset{z}{v} \arrow[ur, dashed, "h"] & \end{tikzcd}$$

Fig. 4

The homomorphism h from S^6 to S^5 defined by $h(u) = n$ and $h(v) = p$ maps the edge (u,v) to the path $(n,m),(m,p)$. Intuitively, the entailment $\varphi^5 \vdash \varphi^6$ holds because, in any model of φ^5, the concatenation between the sequence of integers in the list segment from n to m and the sequence of integers in the list segment from m to p is sorted (i.e., it satisfies the property of the list segment from u to v in φ^6.)

Let $\varphi = \exists N.\ \varphi_S \wedge \varphi_D$ and $\varphi' = \exists N'.\ \varphi'_S \wedge \varphi'_D$ be two SLD formulas and h an homomorphism from φ'_S to φ_S that maps edges of φ'_S to non-empty paths of φ_S. We denote by $\varphi'_D[\![h]\!]$ the sequence formula obtained from φ'_D by (1) substituting u by $h(u)$ in all terms except for $\texttt{len}(u)$ and (2) substituting $\texttt{len}(u)$ by $\sum_{(n,m)\in\overrightarrow{h(u),h(v)}} \texttt{len}(n)$ with v being the successor of u in φ'_S and $\overrightarrow{(h(u),h(v))}$ the path between $h(u)$ and $h(v)$ in the SL-graph of φ_S. For example, $sorted(u)[\![h]\!]$ is the formula:

$$sorted(n+m) := \forall y_1,y_2.\ \underbrace{0 < y_1 < y_2 < \texttt{len}(n) + \texttt{len}(m)}_{G_{n+m}(y_1,y_2)} \Rightarrow \texttt{dt}(n) \leq n[y_1] \leq n[y_2].$$

Remark that the substitution of $\texttt{len}(u)$ by $\texttt{len}(n) + \texttt{len}(m)$ makes the formula $sorted(u)[\![h]\!]$ contain properties of concatenations of list segments.

Note that the entailment $\varphi \vdash \varphi'$ holds if φ_D entails $\varphi'_D[\![h]\!]$. For example, $\varphi^5 \vdash \varphi^6$ holds because the sequence formula of φ^5, denoted φ^5_D, entails $sorted(u)[\![h]\!]$.

The difficulty in proving the entailment between φ^5_D and $sorted(n+m)$ is that φ^5_D does not contain a guarded formula having as guard $0 < y_1 < y_2 < \texttt{len}(n) + \texttt{len}(m)$. In the following, we describe a procedure called `fold` which computes properties of such concatenations of list segments. In this particular case, we add to φ^5_D the trivial formula $\forall y_1,y_2.\ 0 < y_1 < y_2 < \texttt{len}(n) + \texttt{len}(m) \Rightarrow true$ which is strengthened by `fold` into a formula equivalent to $sorted(n+m)$.

For every $G(\mathbf{y})$ as above, `fold` begins by computing a set of auxiliary guards, one for every way of placing positions that satisfy $G(\mathbf{y})$ on the list segments that are concatenated. Then, for every such satisfiable guard $G'(\mathbf{y}')$, it calls the saturation procedure `saturate` to compute a guarded formula of the form $\forall \mathbf{y}'.\ G'(\mathbf{y}') \Rightarrow U'(\mathbf{y}')$ implied by φ. Finally, it defines $U(\mathbf{y})$ as the disjunction of all formulas which are in the right hand side of a guarded formula computed in the previous step (see [5] for more details).

We exemplify the procedure `fold` on the formula $\varphi^5_D \wedge \forall y_1,y_2.\ 0 < y_1 < y_2 < \texttt{len}(n) + \texttt{len}(m) \Rightarrow true$ involved in proving the entailment $\varphi^5 \vdash \varphi^6$ above. A value for y_1 or y_2 that satisfies the constraints in $G_{n+m}(y_1,y_2)$, i.e. it is a position between 1 and $\texttt{len}(n) + \texttt{len}(m) - 1$ on the list starting in n, can either correspond to (1) a position on the sequence associated with the list segment $\texttt{ls}(n,m)$ (when it is less than $\texttt{len}(n)$) or to (2) the first element of the sequence associated with the list segment $\texttt{ls}(m,p)$ (when it equals $\texttt{len}(n)$) or to (3) a position on the tail of the sequence associated with the list segment $\texttt{ls}(m,p)$ (when it is greater than $\texttt{len}(n)$). In each case, an auxiliary guard is computed by adding some constraints to $G_{n+m}(y_1,y_2)$ and by substituting the variables y_1 and y_2 as follows. If y_i is considered to be a position on the tail of some list segment α then the constraint $0 < y_i < \texttt{len}(\alpha)$ is added to $G_{n+m}(y_1,y_2)$ and y_i is substituted by $y_i + \sum_j \texttt{len}(n_j)$, where n_j are all the list segments from n to the predecessor of α. Concretely, if y_i corresponds to a position on the tail of the list segment $\texttt{ls}(n,m)$ then $0 < y_i < \texttt{len}(n)$ is added to $G_{n+m}(y_1,y_2)$ and y_i remains unchanged. If y_i corresponds to a position on the tail of the list segment $\texttt{ls}(m,p)$ then $0 < y_i < \texttt{len}(m)$

is added to $G_{n+m}(y_1,y_2)$ and y_i is substituted by $y_i+\mathtt{len}(n)$. If y_i is considered to be the first element of the list segment $\mathtt{ls}(m,p)$ then it is substituted by the exact value of this position, i.e. $\mathtt{len}(n)$. Below, we consider three cases and give the auxiliary guard computed in each case:

"y_1 is the first element of $\pi_{m,p}$", "y_2 is a position on the tail of $\pi_{m,p}$"

$$0<\mathtt{len}(n)<y_2+\mathtt{len}(n)<\mathtt{len}(n)+\mathtt{len}(m)\wedge 0<y_2<\mathtt{len}(m)$$
$$\equiv 0<y_2<\mathtt{len}(m)$$

"y_1 is a position on the tail of $\pi_{n,m}$", "y_2 is a position on the tail of $\pi_{m,p}$"

$$0<y_1<y_2+\mathtt{len}(n)<\mathtt{len}(n)+\mathtt{len}(m)\wedge 0<y_1<\mathtt{len}(n)\wedge 0<y_2<\mathtt{len}(m)$$
$$\equiv 0<y_1<\mathtt{len}(n)\wedge 0<y_2<\mathtt{len}(m)$$

"y_1 is a position on the tail of $\pi_{m,p}$", "y_2 is the first element of $\pi_{m,p}$"

$$0<y_1+\mathtt{len}(n)<\mathtt{len}(n)<\mathtt{len}(n)+\mathtt{len}(m)\wedge 0<y_1<\mathtt{len}(m)$$
$$\equiv false$$

Notice that the third situation is not possible and it corresponds to an unsatisfiable guard which will be ignored in the following. The procedure `saturate` infers from φ_D^5 the following properties: $\gamma_1 := \forall y_2.\ 0<y_2<\mathtt{len}(m)\Rightarrow \mathtt{dt}(n)\le\mathtt{dt}(m)\le m[y_2]$ and $\gamma_2 := \forall y_1,y_2.\ (0<y_1<\mathtt{len}(n)\wedge 0<y_2<\mathtt{len}(m))\Rightarrow \mathtt{dt}(n)\le\mathtt{dt}(n)\le n[y_1]\le m[y_2]$. The other possible cases for the placement of the positions denoted by y_1 and y_2 are handled in a similar manner.

The right parts of all the generated guarded formulas are "normalized" such that they characterize the terms $n[y_1]$ and $n[y_2]$, where y_1 and y_2 satisfy the constraints in $G_{n+m}(y_1,y_2)$. For example, in the right part of γ_1, $\mathtt{dt}(m)$ is substituted by $n[y_1]$ (because y_1 was considered to be the first element of the list segment starting in m) and $m[y_2]$ is substituted by $n[y_2]$ (because y_2 was considered to be a position on the tail of the list segment starting in m) and in the right part of γ_2, $n[y_1]$ remains unchanged and $m[y_2]$ is substituted by $n[y_2]$. The procedure `fold` returns $\forall y_1,y_2.\ G_{n+m}(y_1,y_2)\Rightarrow U(y_1,y_2)$, where $U(y_1,y_2)$ is the disjunction of all the obtained formulas. In this case, $U(y_1,y_2)$ is equivalent to $\mathtt{dt}(n)\le n[y_1]\le n[y_2]$.

Syntactic entailment: Given two SLD formulas ψ and ψ', the syntactic entailment $\psi\sqsubseteq\psi'$ is defined as follows: for any disjunct φ of ψ there exists a disjunct φ' of ψ' such that $\varphi\sqsubseteq\varphi'$ holds, where the relation $\sqsubseteq$ is defined in Fig. 5.

Correctness and precision results: The following theorem gives precision and correctness results for `fold`. The precision result is implied by the precision of `saturate` for SLD$_\le$ formulas.

Theorem 4. *Let $\varphi=\exists N.\ \varphi_S\wedge\varphi_D$ be a disjunction-free SLD formula. Then,* $\mathtt{fold}(\varphi)$ *is equivalent to φ and* $\mathtt{fold}(\varphi)\sqsubseteq_S\varphi$. *Moreover, for any SLD$_\le$ formula φ,* $\mathtt{fold}(\varphi)=\exists N.\ \varphi_S\wedge\varphi'_D$ *such that for any guard $G(y)$ in φ_D, which describes concatenations of list segments, φ'_D contains the strongest universal formula of the form $\forall\mathbf{y}.\ G(\mathbf{y})\Rightarrow U(\mathbf{y})$ such that $\varphi\vdash(\exists N.\ \varphi_S\wedge\forall\mathbf{y}.\ G(\mathbf{y})\Rightarrow U(\mathbf{y}))$.*

The correctness result for `fold` and `saturate` implies that $\sqsubseteq$ is sound. Next, we identify entailment problems $\psi_1\vdash\psi_2$, where ψ_1 and ψ_2 belong to SLD$_\le$, for which the procedure $\sqsubseteq$ is complete. Roughly, we impose restrictions on ψ_1 and ψ_2 such that a disjunction-free SLD formula in ψ_1 may entail at most one disjunct in ψ_2. For example, we require that ψ_2 is unambiguous. An SLD formula ψ is called *unambiguous* if for any disjunct φ of ψ, the SL-graph of φ contains an undirected (inequality) edge between every two nodes.

Theorem 5 (Soundness). *Let ψ_1 and ψ_2 be* SLD *formulas. If $\psi_1 \sqsubseteq \psi_2$ then $\psi_1 \vdash \psi_2$.*

Theorem 6 (Completeness). *Let ψ_1 and ψ_2 be two* $\mathsf{SLD}_{\leq}$ *formulas. If ψ_1 is unambiguous, ψ_2 is succinct, and for every disjunct φ_1 of ψ_1 there exists at most one disjunct φ_2 of ψ_2 homomorphic to φ_1 then $\psi_1 \vdash \psi_2$ implies* $\texttt{saturate}(\psi_1) \sqsubseteq \psi_2$.

ALGORITHM Syntactic entailment $\varphi \sqsubseteq \varphi'$

```
Require: $\varphi := \exists N.\ \varphi_S \wedge \varphi_D,\ \varphi' := \exists N'.\ \varphi'_S \wedge \varphi'_D$
1: choose $h$ an homomorphism from $\varphi'_S$ to $\varphi_S$
2: choose $(\overline{\varphi}, h, \overline{\varphi'})$ in split$(\varphi, h, \varphi')$
3: add to $\overline{\varphi}$ missing guards from $\overline{\varphi'}[\![h]\!]$
4: $\varphi^1 :=$ fold$(\overline{\varphi})$
5: $\varphi^2 :=$ saturate$(\varphi_1)$
6: check $\varphi^2_D \sqsubseteq_S \overline{\varphi'_D}$, where $\varphi^2_D$ and $\overline{\varphi'_D}$ is the
   sequence formula of $\varphi^2$ and $\overline{\varphi'}$, respectively.
```

Fig. 5

The procedure `saturate` can also be used to check satisfiability of SLD formulas. Notice that an SLD formula $\varphi := \exists N.\ \varphi_S \wedge \varphi_D$ is unsatisfiable iff either the SL-graph of φ_S is $\perp$ or the sequence formula φ_D is unsatisfiable. The latter condition means that the strongest existential constraint E s.t. $\varphi \vdash \exists N.\ (\varphi_S \wedge E)$ is equivalent to *false*.

Theorem 7. *An* SLD *formula ψ is unsatisfiable iff for any disjunct φ of ψ either the SL-graph of φ is $\perp$ or the existential constraint E of* $\texttt{saturate}(\varphi)$ *is unsatisfiable.*

We give in [5] an extension of these results to more general SLD formulas that contain guarded formulas describing concatenations of list segments.

6 Logic SLAD, extension of SLD with arrays

The class of programs considered in Section 2 can be extended to manipulate, besides lists, a fixed set of arrays. We consider that the arrays are not overlapping (e.g., like in Java), and that they are manipulated by operations like allocation, read/write an element, and read the length. A configuration of such programs is also represented by a directed graph and a valuation for the integer program variables. For lack of space, we present here the main ideas of this extension, a detailed presentation of is provided in [5].

Let *AVar* be a set of array variables, disjoint from the sets *PVar* and *DVar*. Also, let *IVar* be the set of integer program variables in *DVar* used in order to access array elements. A variable in *IVar* is called an *index variable*. The syntax of *shape formulas* in SLAD is the one given in Fig. 2 for SLD; only sequence formulas are specifying array properties. The syntax of *sequence formulas* in SLAD extends the one given in Fig. 3 by allowing the following new terms and guards:

Position terms:	E-terms:	U-terms:	Guards:
$p\ ::=\ \ldots \mid i$	$e\ ::=\ \ldots \mid a[p]$	$t\ ::=\ \ldots \mid a[y]$	$G(\mathbf{y})\ ::= C \wedge \bigwedge_{y \in \mathbf{y}} 0 < y < \ell(y)$
$i \in IVar$	$a, a_y \in AVar$		$\ell(y)\ ::= \texttt{len}(n_y) \mid \texttt{len}(a_y)$

The definition of the SLAD guards includes constraints on position variables $y \in \mathbf{y}$ used with arrays. The same condition for U-terms $n[y]$ is applied to terms $a[y]$: they appear in $U(\mathbf{y})$ only if the guard includes the constraint $0 < y < \texttt{len}(a_y)$.

The procedure for checking the syntactic entailment $\varphi_1 \sqsubseteq \varphi_2$ between two disjunction-free formulas $\varphi_1, \varphi_2 \in$ SLAD translates φ_1 and φ_2 into equivalent SLD formulas and then, it applies the syntactic entailment for SLD defined in Fig. 5. Roughly, the translation procedure applied on an SLAD formula φ adds to the shape formula the list segments corresponding to array variables used in the sequence formula, and it soundly translates the terms and guards over arrays into terms and guards over lists. The resulting SLD formula $\overline{\varphi}$ is equivalent to φ and of size polynomial in the size of φ.

The fragment $\mathsf{SLAD}_{\leq}$ of SLAD is defined similarly to the fragment $\mathsf{SLD}_{\leq}$ of SLD (the only difference is that, for any guarded formula $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$, any occurrence of a position variable y in $U(\mathbf{y})$ belongs to a term of the form $n[y]$ or $a[y]$). The following results are straightforward consequences of Th. 6.

Corollary 1. *Let ψ_1, ψ_2 be two formulas in SLAD. If $\psi_1 \sqsubseteq \psi_2$ then $\psi_1 \vdash \psi_2$. Moreover, if ψ_1, ψ_2 are $\mathsf{SLAD}_{\leq}$ formulas satisfying the restrictions in Th. 6, then $\psi_1 \vdash \psi_2$ implies* `saturate`$(\psi_1) \sqsubseteq \psi_2$.

Corollary 2. *Checking the semantic entailment $\psi_1 \vdash \psi_2$, where ψ_1 and ψ_2 are two $\mathsf{SLAD}_{\leq}$ formulas satisfying the restrictions in Th. 6, is decidable. Also, checking the satisfiability of an $\mathsf{SLAD}_{\leq}$ formula is decidable. If we consider $\mathsf{SLAD}_{\leq}$ formulas with a fixed number of universal quantifiers s.t. the logic on data is quantifier-free Presburger arithmetics then the two problems are NP-complete.*

7 Experimental results

We have implemented in CELIA [6] the algorithm $\sqsubseteq$ and the postcondition operator for SLAD, and we have applied the tool to the verification of a significant set of programs manipulating lists and arrays. These programs need invariants and pre/post conditions beyond the decidable fragment $\mathsf{SLAD}_{\leq}$. For example, we have verified a C library implementing sets of integers using strictly sorted lists (procedures/clients of the library are named *setlist-** in Tab. 1). The guarded formulas used by the specifications of this library need guards of the form $y_2 = y_1 + 1$ or $y_1 < y_2$ which are not $\leq$-guards. The verification of the clients is done in a modular way by applying an extension for SLAD of the frame rule from Separation logic.

The verification tool extends the implementation of the abstract domain of universal formulas defined in [6] to which we have added the procedures `saturate`, `split`, `fold`, and the computation of SL-graph homomorphism. The decision procedures $\sqsubseteq$ and `saturate` are also available in an independent tool SLAD whose input are formulas in the SMTLIB2 format. Table 1 provides the characteristics and the experimental results obtained (on a Pentium 4 Xeon at 4 GHz) for an illustrative sample of the verified programs. The full list of verified programs is available at `www.liafa.jussieu.fr/celia/verif/`.

8 Related work and conclusions

Various frameworks have been developed for the verification of programs based on logics for reasoning about data structures, e.g., [1, 4, 6–9, 12–18].

Decidable logics for unbounded data domains: Several works have addressed the issue of reasoning about programs manipulating data structures with unbounded data, e.g. [4, 8, 13–15, 19]. The logics in [8, 13] allow to reason about arrays and they are fragments of SLAD (see [5]). The fragment $\mathsf{SLAD}_{\leq}$, for which the satisfiability problem is decidable, includes the Array Property Fragment [8] when defined over finite arrays but, it is incomparable to the logic LIA [13].

The logics in [4, 14] to reason about composite data structures are more expressive concerning the shape constraints but they are less expressive than SLAD when restricted

Program	pre-cond		inv		`post`(inv)		post-cond		Verif.
	size	logic	size	logic	size	logic	size	logic	time
copyaddV	$1\vee\times 0\forall$	$\mathsf{SLD}_{\leq}$	$3\vee\times 1\forall$	$\mathsf{SLD}_{\leq}$	$2\vee\times 1\forall$	$\mathsf{SLD}_{\leq}$	$1\vee\times 1\forall$	$\mathsf{SLD}_{\leq}$	< 1s
initSeq	$1\vee\times 0\forall$	$\mathsf{SLD}_{\leq}$	$3\vee\times 1\forall$	SLD	$2\vee\times 1\forall$	SLD	$1\vee\times 1\forall$	SLD	< 1s
initFibo	$1\vee\times 0\forall$	$\mathsf{SLD}_{\leq}$	$3\vee\times 2\forall$	SLD	$4\vee\times 2\forall$	SLD	$1\vee\times 1\forall$	$\mathsf{SLD}_{\leq}$	< 1s
setlist-contains	$1\vee\times 2\forall$	SLD	$3\vee\times 4\forall$	SLD	$4\vee\times 4\forall$	SLD	$2\vee\times 3\forall$	SLD	< 1s
setlist-add	$1\vee\times 2\forall$	SLD	$3\vee\times 7\forall$	SLD	$4\vee\times 7\forall$	SLD	$1\vee\times 1\forall$	SLD	< 1s
setlist-union	$1\vee\times 4\forall$	SLD	$4\vee\times 13\forall$	SLD	$5\vee\times 13\forall$	SLD	$1\vee\times 6\forall$	SLD	< 2s
setlist-intersect	$1\vee\times 4\forall$	SLD	$3\vee\times 13\forall$	SLD	$4\vee\times 13\forall$	SLD	$1\vee\times 6\forall$	SLD	< 2s
setlist-client	$0\vee\times 0\forall$	SLD	$2\vee\times 2\forall$	SLD	$3\vee\times 2\forall$	SLD	$1\vee\times 2\forall$	SLD	< 1s
svcomp-list-prop	$1\vee\times 0\forall$	$\mathsf{SLD}_{\leq}$	$3\vee\times 2\forall$	SLD	$4\vee\times 2\forall$	SLD	$1\vee\times 1\forall$	SLD	< 1s
array2list	$1\vee\times 0\forall$	$\mathsf{SLAD}_{\leq}$	$2\vee\times 1\forall$	$\mathsf{SLAD}_{\leq}$	$2\vee\times 1\forall$	$\mathsf{SLAD}_{\leq}$	$1\vee\times 1\forall$	$\mathsf{SLAD}_{\leq}$	< 1s
array-insertsort	$1\vee\times 4\forall$	$\mathsf{SLAD}_{\leq}$	$3\vee\times 5\forall$	$\mathsf{SLAD}_{\leq}$	$2\vee\times 5\forall$	$\mathsf{SLAD}_{\leq}$	$1\vee\times 4\forall$	$\mathsf{SLAD}_{\leq}$	< 3s

Table 1: Experimental results. Size of formulas $n\vee\times m\forall$ means n disjuncts (i.e., shape formulas) with at most m guarded formulas per disjunct. `copyAddV` creates a list from an input list with all data increased by some data parameter. `initSeq` initializes data in a list with consecutive values starting from some data parameter. `initFibo` initializes data in a list with the Fibonacci sequence. `setlist-*` are procedures/clients of a library implementing sets as strictly sorted lists. `svcomp-list-prop` is an example from the SV-COMP competition. `array2list` creates a list from an array.

to heaps formed of singly-linked lists and arrays. The restriction of LISBQ [14] to this class of heaps is included in $\mathsf{SLAD}_{\leq}$ but, the similar restriction of CSL [4] is not included in $\mathsf{SLAD}_{\leq}$; the latter does not permit guards that contain strict inequalities. The decidable fragment of STRAND [15] can describe other recursive data structures than lists but, its restriction to lists is incomparable to SLD: it does not include properties on data expressed using the immediate successor relation between nodes in the lists.

Sound decision procedures: Decision procedures which are sound but, in general, not complete, have been proposed in [11, 16, 10, 18]. The work in [18] targets functional programs and it is not appropriate for imperative programs that mutate the heap.

The framework in [16] considers recursive programs on trees and it defines a sound decision procedure for proving Hoare triples. The validity of the Hoare triple is reduced through some abstraction mechanism to the validity of a formula in a decidable logic. In this paper, we describe a sound procedure for checking entailments between formulas in SLAD, which is independent of the fact that these entailments are obtained from some Hoare triples. Moreover, SLAD is incomparable to the logic used in [16].

Current state of the art SMT solvers do not include a theory for lists having the same expressiveness as SLD. For arrays, most of the SMT solvers deal with formulas in the Array Property Fragment [8]. However, they may prove entailments between array properties in SLAD but not in $\mathsf{SLAD}_{\leq}$ by using heuristics for quantifier instantiation, see e.g. Z3 [11, 10]. Our entailment procedure, which is based on the saturation procedure `saturate`, is more powerful because it is independent of the type of constraints that appear in the right hand side of the guarded formulas. The heuristics used in Z3 work well when the entailment can be proved using some boolean abstraction of the formulas or when the right hand side of the guarded formulas contains only equalities.

In our previous work [6, 7], we introduced a logic on lists called `SL3`, which is included in SLAD. In `SL3`, data properties are also described by universal implications $\forall \mathbf{y}.\ G(\mathbf{y}) \Rightarrow U(\mathbf{y})$ but the guard $G(\mathbf{y})$ is not as expressive as in SLAD. Any two node variables in an `SL3` formula denote distinct vertices in the heap. This can lead to an exponential blow-up for the size of the formulas which implies a blow-up in the complexity of the decision procedure. Checking an entailment between SL3 formulas is reduced to the abstract analysis of a program that traverses the allocated lists and thus, it is impossible to characterize its preciseness using completeness results.

Conclusions: We have defined an approach for checking entailment and satisfiability of formulas on lists and arrays with data. Our approach deals with complex assertions that are beyond the reach of existing techniques and decision procedures. Although we have considered only programs with singly-linked lists and arrays, our techniques can be extended to other classes of data structures (doubly-linked lists, trees) using appropriate embeddings of heap graphs into finite abstract graphs.

References

1. I. Balaban, A. Pnueli, and L.D. Zuck. Shape analysis of single-parent heaps. In *VMCAI*, volume 4349 of *LNCS*, pages 91–105. Springer, 2007.
2. M. Benedikt, T.W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *ESOP*, volume 1576 of *LNCS*, pages 2–19. Springer, 1999.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.
4. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR*, volume 5710 of *LNCS*, pages 178–195. Springer, 2009.
5. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. Technical report, LIAFA, 2011. At http://www.liafa.univ-paris-diderot.fr/∼cenea/SLAD.pdf.
6. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589. ACM, 2011.
7. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, volume 7148 of *LNCS*, pages 1–22. Springer, 2012.
8. A.R. Bradley, Z. Manna, and H.B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
9. B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901 of *LNCS*, pages 235–249, 2011.
10. L. de Moura and N. Bjørner. Efficient e-matching for smt solvers. In *CADE*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
11. Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.
12. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.
13. P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In *FoSSaCS*, volume 4962 of *LNCS*, pages 474–489. Springer, 2008.
14. S.K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, pages 171–182. ACM, 2008.
15. P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data d. In *POPL*, pages 283–294. ACM, 2011.
16. P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree datastructures. In *POPL*, pages 123–136. ACM, 2012.
17. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
18. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210. ACM, 2010.
19. T. Wies, M. Muñiz, and V. Kuncak. An efficient decision procedure for imperative tree data structures. In *CADE*, volume 6803 of *LNCS*, pages 476–491. Springer, 2011.