Logical Methods in Computer Science
Volume 17, Issue 2, 2021, pp. 15:1–15:33
https://lmcs.episciences.org/

Submitted Feb. 19, 2018
Published May 11, 2021

DECIDABILITY FOR ENTAILMENTS OF SYMBOLIC HEAPS WITH ARRAYS

DAISUKE KIMURA [a] AND MAKOTO TATSUTA [b]

[a] Toho University, Chiba, Japan
e-mail address: kmr@is.sci.toho-u.ac.jp

[b] National Institute of Informatics, Tokyo, Japan
e-mail address: tatsuta@nii.ac.jp

Abstract. This paper presents two decidability results on the validity checking problem for entailments of symbolic heaps in separation logic with Presburger arithmetic and arrays. The first result is for a system with arrays and existential quantifiers. The correctness of the decision procedure is proved under the condition that sizes of arrays in the succedent are not existentially quantified. This condition is different from that proposed by Brotherston et al. in 2017 and one of them does not imply the other. The main idea is a novel translation from an entailment of symbolic heaps into a formula in Presburger arithmetic. The second result is the decidability for a system with both arrays and lists. The key idea is to extend the unroll collapse technique proposed by Berdine et al. in 2005 to arrays and arithmetic as well as double-linked lists.

1. Introduction

Separation logic [20] has been successfully used to verify/analyze heap-manipulating imperative programs with pointers [3, 7, 8, 9, 12], and in particular it is successful for verify/analyze memory safety. The advantage of separation logic is modularity brought by the frame rule, with which we can independently verify/analyze each function that may manipulate heaps [9].

In order to develop an automated analyzer/verifier of pointer programs based on separation logic, symbolic-heap systems, which are fragments of separation logic, are often considered [2, 3, 9]. Despite of its simple and restricted form, symbolic heaps have enough expressive power, for example, Infer [10] and HIP/SLEEK are based on symbolic-heap systems. Symbolic heaps are used as assertions A and B in Hoare-triple $\{A\}P\{B\}$. For program analysis/verification, the validity of entailments need to be checked automatically.

Inductive definitions in a symbolic-heap system is important, since they can describe recursive data structures such as lists and trees. Symbolic-heap systems with inductive predicates have been studied intensively [2, 3, 1, 6, 11, 13, 14, 15, 16, 21]. Berdine et al. [2, 3] introduced a symbolic-heap system with hard-coded list and tree predicates, and showed the decidability of its entailment problem. Iosif et al. [15, 16] considered a system with general

Key words and phrases: Separation Logic, Program Analysis/Verification, Decidability, Arrays, Lists.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-17(2:15)2021

inductive predicates, and showed its decidability under the bounded tree-width condition. Tatsuta et al. [21] introduced a system with general monadic inductive predicates.

Arrays are one of the most important primitive data structures of programs. It is also important to verify that there is no buffer overflow in programs with arrays. In order to verify/analyze pointer programs with arrays, this paper introduces two symbolic-heap systems with the array predicate.

The first symbolic-heap system, called **SLA** (Separation Logic with Arrays), contains the points-to and the array predicates as well as existential quantifiers in spatial parts. The entailments of this system also have disjunction in the succedents. Our first main theorem is the decidability of the entailment problem. For this theorem we need the condition: the sizes of arrays in the succedent of an entailment do not contain any existential variables. It means that the size of arrays in the succedent is completely determined by the antecedent.

The basic idea of our decision procedure for **SLA** is a novel translation of a given entailment into an equivalent formula in Presburger arithmetic. We use "*sorted*" symbolic heaps as a key idea for defining the translation. A heap represented by a sorted symbolic heap has addresses sorted in the order of the spatial part. If both sides of a given entailment are sorted, the validity of the entailment is decided by comparing spatial parts on both sides starting from left to right.

The second system, called **SLAL** (Separation Logic with Arrays and Lists), contains the points-to predicate, the array predicate, the singly-linked list predicate, and the doubly-linked list predicate. They also have disjunction in the succedents. The entailments of this system are restricted to quantifier-free symbolic heaps. This restriction comes from a technical reason, namely, our key idea (the unroll collapse technique) does not work in the presence of existential quantifiers. Although the restriction reduces the expressive power of entailments, the quantifier-free entailments are useful for verify/analyze memory safety [2, 3].

The second main theorem of this paper is the decidability of the entailment problem for **SLAL**. Our decision procedure is split into the two stages (a) and (b): (a) the first stage eliminates the list predicates from the antecedent of a given entailment by applying *the unroll collapse technique*. Originally the unroll collapse for acyclic singly-linked list segments is invented by Berdine et al. [2]. We extend the original one in two ways; the unroll collapse in **SLAL** is extended to arithmetic and arrays as well as doubly-linked list segments. (b) the second stage eliminates list predicates from the succedent of the entailment by the proof search technique. To do this, we introduce a sound and complete proof system that has valid quantifier-free entailments in **SLA** as axioms, which are checked by the first decision procedure for **SLA**.

As related work, as far as we know there are two papers about symbolic-heap systems that have arrays as primitive. Calcagno et al. [8] studied program analysis based on a symbolic-heap system in the presence of pointer arithmetic. They assumed a decision procedure for entailments with arrays and did not propose it. Brotherston et al. [7] considered the same system as **SLA**, and investigated several problems about it.

In [7], they proposed a decision procedure for the entailment problem (of **SLA**) by giving an equivalent condition to the existence of a counter-model for a given entailment, then checking a Presburger formula that expresses the condition. In order to do this, they imposed the restriction that the second argument of the points-to predicate in the succedent of an entailment is not existentially quantified. Their result decides a different class of entailments from the class decided by ours, and our class neither contains their class nor is contained by it.

When we extend separation logic with arrays, it may be different from previous array logics in the points that (1) it is specialized for memory safety, and (2) it can scale up by modularity. Bradley et al. [5], Bouajjani et al. [4], and Lahiri et al. [18] discussed logics for arrays but their systems are totally different from separation logic. So we cannot apply their techniques to our case. Piskac et al. [19] proposed a separation logic system with list segments, and it can be combined with various SMT solvers, including array logics. However, when we combine it with array logics, the arrays are external and the resulting system does not describe the arrays by spatial formulas with separating conjunction.

The first result of this paper is based on [17] but we give detailed proofs of the key lemmas (stated in Lemma 4.1) for the result.

The paper is organized as follows. Section 2 introduces the first system **SLA**. Section 3 defines and discusses the decision procedure of the entailment problem for **SLA**. In Section 4, we show the first main theorem, namely, the decidability result of **SLA**. In Section 5, we introduce the second system **SLAL**. Section 6 shows the unroll collapse property. Section 7 gives a decision procedure for **SLAL**. Section 8 shows the second main theorem, namely, the decidability result of **SLAL**. We conclude in Section 9.

2. Separation Logic with Arrays

This section defines the syntax and semantics of our separation logic with arrays. We first give the separation logic **G** with arrays in the ordinary style. Then we define the symbolic-heap system **SLA** as a fragment of **G**.

2.1. Syntax of System G of Separation Logic with Arrays.

We use the following notations in this paper. Let $(p_j)_{j\in J}$ be a sequence indexed by a finite set J. We write $\{p_j \mid j \in J\}$ for this sequence. This sequence will sometimes be abbreviated by $\overrightarrow{p}$. We write $q \in \overrightarrow{p}$ when q is an element of $\overrightarrow{p}$.

We have first-order variables $x, y, z, \ldots \in \mathrm{Vars}$ and constants $0, 1, 2, \ldots$. The syntax of **G** is defined as follows:

$$\begin{array}{rl} \text{Terms} & t ::= x \mid 0 \mid 1 \mid 2 \mid \cdots \mid t + t. \\ \text{Formulas} & F ::= t = t \mid F \wedge F \mid \neg F \mid \exists x F \mid \mathrm{Emp} \mid t \mapsto (t, \ldots, t) \mid \mathrm{Arr}(t, t) \mid F * F. \end{array}$$

Atomic formulas $t \mapsto (u_1, \ldots, u_{pt})$ and $\mathrm{Arr}(t, u)$ are called a points-to atomic formula and an array atomic formula, respectively. The points-to predicate $\mapsto$ is $(pt+1)$-ary predicate (the number pt is fixed beforehand). We sometimes write $t \mapsto u$ instead of $t \mapsto (u)$ when $\mapsto$ is a binary predicate. We use the symbol - to denote an unspecified term.

Each formula is interpreted by a variable assignment and a heap: Emp is true when the heap is empty; $t \mapsto (\overrightarrow{u})$ is true when the heap has only a single memory cell of address t that contains the value $\overrightarrow{u}$; $\mathrm{Arr}(t, u)$ is true when the heap has only an array of index from t to u; a separating conjunction $F_1 * F_2$ is true when the heap is split into some two disjoint sub-heaps, F_1 is true for one, and F_2 is true for the other. The formal definition of these interpretations is given in the next subsection.

A term t that appears in either $t \mapsto (\text{-})$, $\mathrm{Arr}(t, \text{-})$ or $\mathrm{Arr}(\text{-}, t)$ of F is called an address term of F. The set of free variables (denoted by $\mathrm{FV}(F)$) of F is defined as usual. We also define $\mathrm{FV}(\overrightarrow{F})$ as the union of $\mathrm{FV}(F)$, where $F \in \overrightarrow{F}$.

We use abbreviations $F_1 \vee F_2$, $F_1 \to F_2$, and $\forall x F$ defined in the usual way. We also write $t \neq u$, $t \leq u$, $t < u$, True, and False for $\neg(t = u)$, $\exists x(u = t + x)$, $t + 1 \leq u$, $0 = 0$, and $0 \neq 0$, respectively.

A formula is said to be *pure* if it is a formula of Presburger arithmetic.

We implicitly use the associative and commutative laws for the connectives $*$ and $\wedge$, the fact that Emp is the unit of $*$ and True is the unit of $\wedge$, and permutation of the existential quantifiers.

For I a finite set and $\{F_i\}_{i\in I}$ a set of formulas, we write $\bigwedge_{i\in I} F_i$ for the conjunction of the elements of $\{F_i\}_{i\in I}$. If I is empty, it is defined as True.

2.2. **Semantics of System G of Separation Logic with Arrays.** Let N be the set of natural numbers. We define the following semantic domains:

$$\mathrm{Val} \stackrel{\mathrm{def}}{=} N, \quad \mathrm{Loc} \stackrel{\mathrm{def}}{=} N \setminus \{0\}, \quad \mathrm{Store} \stackrel{\mathrm{def}}{=} \mathrm{Vars} \to \mathrm{Val}, \quad \mathrm{Heap} \stackrel{\mathrm{def}}{=} \mathrm{Loc} \to_{\mathrm{fin}} \mathrm{Val}^{pt}.$$

Loc means addresses of heaps. 0 means Null. An element s in Store is called a *store* that means a valuation of variables. The update $s[x_1 := a_1, \ldots, x_k := a_k]$ of s is defined by $s[x_1 := a_1, \ldots, x_k := a_k](z) = a_i$ if $z = x_i$, otherwise $s[x_1 := a_1, \ldots, x_k := a_k](z) = s(z)$. An element h in Heap is called a *heap*. The domain of h (denoted by $\mathrm{Dom}(h)$) means the memory addresses which are currently used. $h(n)$ means the value at the address n if it is defined. We sometimes use notation $h_1 + h_2$ for the disjoint union of h_1 and h_2, that is, it is defined when $\mathrm{Dom}(h_1)$ and $\mathrm{Dom}(h_2)$ are disjoint sets, and $(h_1 + h_2)(n)$ is $h_i(n)$ if $n \in \mathrm{Dom}(h_i)$ for $i = 1, 2$. The restriction $h|_X$ of h is defined by $\mathrm{Dom}(h|_X) = X \cap \mathrm{Dom}(h)$ and $h|_X(m) = h(m)$ for any $m \in \mathrm{Dom}(h|_X)$. A pair (s, h) is called a *heap model*.

The interpretation $s(t)$ of a term t by s is defined by extending the definition of s by $s(n) = n$ for each constant n, and $s(t + u) = s(t) + s(u)$. We also use the notation $s(\overrightarrow{u})$ defined by $s(u_1, \ldots, u_k) = (s(u_1), \ldots, s(u_k))$.

The interpretation $s, h \models F$ of F under the heap model (s, h) is defined inductively as follows:

$s, h \models t = u$ iff $s(t) = s(u)$,
$s, h \models F_1 \wedge F_2$ iff $s, h \models F_1$ and $s, h \models F_2$,
$s, h \models \neg F$ iff $s, h \not\models F$,
$s, h \models \exists x F$ iff $s[x := a], h \models F$ for some $a \in \mathrm{Val}$,
$s, h \models \mathrm{Emp}$ iff $\mathrm{Dom}(h) = \emptyset$,
$s, h \models t \mapsto (\overrightarrow{u})$ iff $\mathrm{Dom}(h) = \{s(t)\}$ and $h(s(t)) = s(\overrightarrow{u})$,
$s, h \models \mathrm{Arr}(t, u)$ iff $s(t) \leq s(u)$ and $\mathrm{Dom}(h) = \{x \in N \mid s(t) \leq x \leq s(u)\}$,
$s, h \models F_1 * F_2$ iff $s, h_1 \models F_1$, $s, h_2 \models F_2$, and $h = h_1 + h_2$ for some h_1, h_2.

We sometimes write $s \models F$ if $s, h \models F$ holds for any h. This notation is mainly used for pure formulas, since their interpretation does not depend on the heap-part of heap models. We also write $\models F$ if $s, h \models F$ holds for any s and h.

The notation $F_1 \models F_2$ is an abbreviation of $\models F_1 \to F_2$, that is, $s, h \models F_1$ implies $s, h \models F_2$ for any s and h.

2.3. **Symbolic-Heap System with Arrays.** The symbolic-heap system **SLA** is defined as a fragment of **G**. The syntax of **SLA** is given as follows. Terms of **SLA** are the same as those of **G**. Formulas of **SLA** (called *symbolic heaps*) have the following form:

$$\phi ::= \exists \overrightarrow{x}(\Pi \wedge \Sigma)$$

where Π is a pure formula of **G** and Σ is the spatial part defined by

$$\Sigma ::= \mathrm{Emp} \mid t \mapsto (t, \ldots, t) \mid \mathrm{Arr}(t, t) \mid \Sigma * \Sigma.$$

In this paper, we use the following notations for symbolic heaps. The symbol σ is used to denote an atomic formula of Σ. We write $x \mapsto \_$ for $\exists z(x \mapsto z)$ where z is fresh. We also write $\exists \overrightarrow{x}(\Pi \wedge \Sigma) \wedge \Pi'$ for $\exists \overrightarrow{x}(\Pi \wedge \Pi' \wedge \Sigma)$, write $\exists \overrightarrow{x}(\Pi \wedge \Sigma) * \Sigma'$ for $\exists \overrightarrow{x}(\Pi \wedge \Sigma * \Sigma')$, and write $\exists \overrightarrow{x}(\Pi \wedge \Sigma) * \exists \overrightarrow{x'}(\Pi' \wedge \Sigma')$ for $\exists \overrightarrow{x}\, \overrightarrow{x'}(\Pi \wedge \Pi' \wedge \Sigma * \Sigma')$.

In this paper, we consider *entailments* of **SLA** that have the form:

$$\phi \vdash \{\phi_i \mid i \in I\} \qquad (I \text{ is a finite set}).$$

The left-hand side of the symbol $\vdash$ is called the antecedent. The right-hand side of the symbol $\vdash$ is called the succedent. The right-hand side $\{\phi_i \mid i \in I\}$ of an entailment means the disjunction of the symbolic heaps ϕ_i $(i \in I)$.

An entailment $\phi \vdash \{\phi_i \mid i \in I\}$ is said to be *valid* if $\phi \models \bigvee\{\phi_i \mid i \in I\}$ holds.

A formula of the form $\Pi \wedge \Sigma$ is called a *QF symbolic heap* (denoted by φ). Note that existential quantifiers may appear in the pure part of a QF symbolic heap. We can easily see that $\exists \overrightarrow{x}\varphi \models \overrightarrow{\phi}$ is equivalent to $\varphi \models \overrightarrow{\phi}$. So we often assume that the left-hand sides of entailments are QF symbolic heaps.

We call entailments of the form $\varphi \vdash \{\varphi_i \mid i \in I\}$ *QF entailments*.

2.4. **Analysis/Verification of Memory Safety.** We intend to use our entailment checker as a part of our analysis/verification system for memory safety. We briefly explain it for motivating our entailment checker.

The target programming language is essentially the same as that in [20] except we extend the allocation command `malloc`, which returns the first address of the allocated memory block or returns nil if it fails to allocate. We define our programming language in programming language C style.

$$\begin{aligned}
\text{Expressions} \quad & e ::= x \mid 0 \mid 1 \mid 2 \ldots \mid e + e. \\
\text{Boolean expressions} \quad & b ::= e == e \mid e < e \mid b \&\& b \mid b \| b \mid !b. \\
\text{Programs} \quad & P ::= x = e; \mid \texttt{if}\ (b)\{P\}\ \texttt{else}\ \{P\}; \mid \texttt{while}\ (b)\{P\}; \mid P\ P \\
& \qquad \mid x = \texttt{malloc}(y); \mid x = *y; \mid *x = y; \mid \texttt{free}(x);.
\end{aligned}$$

$x = \texttt{malloc}(y)$; allocates y cells and set x to the pointer to the first cell. Note that this operation may fail if there is not enough free memory.

Our assertion language is a disjunction of symbolic heaps, namely,

$$\text{Assertions} \quad A ::= \phi_1 \vee \cdots \vee \phi_n.$$

We write $\phi * (\phi_1 \vee \ldots \vee \phi_n)$ for $(\phi * \phi_1 \vee \ldots \vee \phi * \phi_n)$, and write $\exists x(\phi_1 \vee \ldots \vee \phi_n)$ for $(\exists x\phi_1 \vee \ldots \vee \exists x\phi_n)$. The notation $\Pi \wedge (\phi_1 \vee \ldots \vee \phi_n)$ is defined similarly.

In the same way as [20], we use a triple $\{A\}\ P\ \{B\}$ that means that if the assertion A holds at the initial state and the program P is executed, then (1) if P terminates then the assertion B holds at the resulting state, and (2) P does not cause any memory errors.

As inference rules for triples, we have ordinary inference rules for Hoare triples including the consequence rule, as well as the following rules (axioms) for memory operations. We

write $\mathrm{Arr2}(x, y)$ for $\exists z(\mathrm{Arr}(x, z) \wedge x + y = z + 1)$. $\mathrm{Arr2}(x, y)$ denotes the memory block at address x of size y.

$\{A\}\ x = \mathtt{malloc}(y); \{\exists x'(A[x := x'] \wedge x = \mathrm{nil} \vee A[x := x'] * \mathrm{Arr2}(x, y[x := x']))\},$
$\{A * y \mapsto t\}\ x = *y; \{\exists x'(A[x := x'] * y \mapsto t[x := x'] \wedge x = t[x := x'])\},$
$\{\exists x'(A * x \mapsto x')\}\ *x = y; \{A * x \mapsto y\},$
$\{\exists x'(A * x \mapsto x')\}\ \mathtt{free}(x); \{A\},$ where x' is fresh.

In order to prove memory safety of a program P under a precondition A, it is sufficient to show that $\{A\}P\{\mathrm{True}\}$ is provable.

By separation logic with arrays, we can show a triple $\{A\}\ x = \mathtt{malloc}(y); \{\exists x'(A[x := x'] \wedge x = \mathrm{nil} \vee A[x := x'] * \mathrm{Arr2}(x, y[x := x']))\}$, but it is impossible without arrays since y in $\mathtt{malloc}(y)$ is a variable. With separation logic with arrays, we can also show $\{\mathrm{Arr}(p, p + 3)\}\ *p = 5; \{p \mapsto 5 * \mathrm{Arr}(p + 1, p + 3)\}$.

For the consequence rule

$$\frac{\{A'\}\ P\ \{B'\}}{\{A\}\ P\ \{B\}} \qquad (\text{if } A \to A' \text{ and } B' \to B)$$

we have to check the side conditions $A \to A'$ and $B' \to B$. Let A be $\phi_1 \vee \ldots \vee \phi_n$ and A' be $\phi'_1 \vee \ldots \vee \phi'_m$. Then we will use our entailment checker to decide $\phi_i \vdash \phi'_1, \ldots, \phi'_m$ for all $1 \leq i \leq n$.

3. Decision Procedure for **SLA**

This section gives our decision procedure of the entailment problem for **SLA** by introducing the key idea, namely sorted symbolic heaps, for the decision procedure, and defining the translation from sorted symbolic heaps into formulas in Presburger arithmetic. We finally state the decidability result for **SLA**, which is the first main theorem in this paper.

3.1. **Sorted Entailments.**

This subsection describes *sorted* symbolic heaps. In this and the next sections, for simplicity, we assume that the number pt is 1, that is, the points-to predicate is a binary one. The decidability result and the decision procedure in these sections can be straightforwardly extended to arbitrary pt. In these two sections, we will not implicitly use the commutative law for $*$, or the unit law for Emp in order to define the following notations.

We give a pure formula $t < \Sigma$, which means the first address expressed by Σ is greater than t. It is inductively defined as follows:

$$t < \mathrm{Emp} \stackrel{\mathrm{def}}{=} \mathrm{True}, \qquad t < t_1 \mapsto (\text{-}) \stackrel{\mathrm{def}}{=} t < t_1,$$
$$t < \mathrm{Arr}(t_1, \text{-}) \stackrel{\mathrm{def}}{=} t < t_1, \qquad t < (\mathrm{Emp} * \Sigma_1) \stackrel{\mathrm{def}}{=} t < \Sigma_1,$$
$$t < (t_1 \mapsto (\text{-}) * \Sigma_1) \stackrel{\mathrm{def}}{=} t < t_1, \qquad t < (\mathrm{Arr}(t_1, \text{-}) * \Sigma_1) \stackrel{\mathrm{def}}{=} t < t_1.$$

Then we inductively define a pure formula $\mathrm{Sorted}(\Sigma)$, which means the address terms are sorted in Σ as follows.

$$\mathrm{Sorted}'(\mathrm{Emp}) \stackrel{\mathrm{def}}{=} \mathrm{True},$$

$$\mathrm{Sorted}'(t \mapsto u) \stackrel{\mathrm{def}}{=} \mathrm{True},$$

$$\mathrm{Sorted}'(\mathrm{Arr}(t, u)) \stackrel{\mathrm{def}}{=} t \leq u,$$

$$\mathrm{Sorted}'(\mathrm{Emp} * \Sigma_1) \stackrel{\mathrm{def}}{=} \mathrm{Sorted}'(\Sigma_1),$$

$$\mathrm{Sorted}'(t \mapsto u * \Sigma_1) \stackrel{\mathrm{def}}{=} t < \Sigma_1 \wedge \mathrm{Sorted}'(\Sigma_1),$$

$$\mathrm{Sorted}'(\mathrm{Arr}(t,u) * \Sigma_1) \stackrel{\mathrm{def}}{=} t \leq u \wedge u < \Sigma_1 \wedge \mathrm{Sorted}'(\Sigma_1),$$

$$\mathrm{Sorted}(\Sigma) \stackrel{\mathrm{def}}{=} 0 < \Sigma \wedge \mathrm{Sorted}'(\Sigma).$$

We define $\Sigma^{\sim}$ as $\mathrm{Sorted}(\Sigma) \wedge \Sigma$. Let ϕ be $\exists \vec{x}(\Pi \wedge \Sigma)$. We write $\tilde{\phi}$ or $\phi^{\sim}$ for $\exists \vec{x}(\Pi \wedge \Sigma^{\sim})$. We call $\tilde{\phi}$ a *sorted symbolic heap*.

We define $\mathrm{Perm}(\Sigma)$ as the set of permutations of Σ with respect to $*$. A symbolic heap ϕ' is called a permutation of ϕ if $\phi = \exists \overrightarrow{x}(\Pi \wedge \Sigma)$, $\phi' = \exists \overrightarrow{x}(\Pi \wedge \Sigma')$ and Σ' is a permutation of Σ. We write $\mathrm{Perm}(\phi)$ for the set of permutations of ϕ. Note that $s, h \models \tilde{\phi}'$ for some $\phi' \in \mathrm{Perm}(\phi)$ if and only if $s, h \models \phi$. An entailment is said to be *sorted* if all symbolic heaps in its antecedent and succedent are sorted.

The next lemma claims that checking the validity of entailments can be reduced to checking the validity of sorted entailments.

Lemma 3.1. *$s \models \tilde{\varphi}' \rightarrow \bigvee\{\tilde{\phi}' \mid i \in I, \phi' \in \mathrm{Perm}(\phi_i)\}$ for all $\varphi' \in \mathrm{Perm}(\varphi)$ if and only if $s \models \varphi \rightarrow \bigvee_{i \in I} \phi_i$.*

Proof. We first show the left-to-right part. Assume the left-hand side of the claim. Fix $\varphi' \in \mathrm{Perm}(\varphi)$ and suppose $s, h \models \tilde{\varphi}'$. Then we have $s, h \models \varphi$. By the assumption, $s, h \models \phi_i$ for some $i \in I$. Hence we have $s, h \models \bigvee\{\tilde{\phi}' \mid i \in I, \phi' \in \mathrm{Perm}(\phi_i)\}$. Next we show the right-to-left part. Assume the right-hand side and $s, h \models \varphi$. We have $s, h \models \tilde{\varphi}'$ for some $\varphi' \in \mathrm{Perm}(\varphi)$. By the assumption, $s, h \models \tilde{\phi}'$ for some $\phi' \in \mathrm{Perm}(\phi_i)$. Thus we have $s, h \models \phi_i$ for some $i \in I$. □

The basic idea of our decision procedure is as follows: (1) A given entailment is decomposed into sorted entailments according to Lemma 3.1; (2) the decomposed sorted entailments are translated into Presburger formulas by the translation P given in the next subsection; (3) the translated formulas are decided by the decision procedure of Presburger arithmetic.

3.2. **Translation P.** We define the translation P from QF entailments into Presburger formulas. We note that the resulting formula may contain new fresh variables (denoted by z). In the definition of P, we fix a linear order on an index set I to take an element of the minimum index. For saving space, we use some auxiliary notations. Let $\{t_j\}_{j \in J}$ be a set of terms indexed by a finite set J. We write $u = t_J$ for $\bigwedge_{j \in J} u = t_j$. We also write $u < t_J$ for $\bigwedge_{j \in J} u < t_j$. We note that both $u = t_\emptyset$ and $u < t_\emptyset$ are True, since $\bigwedge_{j \in \emptyset} u = t_j$ and $\bigwedge_{j \in \emptyset} u < t_j$ are defined by True.

The definition of $P(\Pi, \Sigma, S)$ is given as listed in Fig. 1, where S is a finite set $\{(\Pi_i, \Sigma_i)\}_{i \in I}$. We assume that pattern-matching is done from the top to the bottom.

In order to describe the procedure P, we temporarily extend terms to include $u - t$ where u, t are terms. In the result of P, which is a Presburger arithmetic formula, we eliminate these extended terms by replacing $t' + (u - t) = t''$ and $t' + (u - t) < t''$ by $t' + u = t'' + t$ and $t' + u < t'' + t$, respectively.

Let $\sharp_*(\Sigma)$ be the number of $*$ in Σ. Let $\sharp_*(\{(\Pi_i, \Sigma_i)\}_{i \in I})$ be $\sum_{i \in I} \sharp_*(\Sigma_i)$. We define $\mathrm{FirstRemove}_{\Sigma}(\Sigma')$ by Σ'_0 if Σ has the form $t \mapsto u * \Sigma_0$ and Σ' has the form $t' \mapsto u' * \Sigma'_0$, or

$$
\begin{array}{lll}
P(\Pi, \mathrm{Emp} * \Sigma, S) & \stackrel{\mathrm{def}}{=} & P(\Pi, \Sigma, S) \qquad \textbf{(EmpL)} \\
P(\Pi, \Sigma, \{(\Pi', \mathrm{Emp} * \Sigma')\} \cup S) & \stackrel{\mathrm{def}}{=} & P(\Pi, \Sigma, \{(\Pi', \Sigma')\} \cup S) \qquad \textbf{(EmpR)} \\
P(\Pi, \mathrm{Emp}, \{(\Pi', \Sigma')\} \cup S) & \stackrel{\mathrm{def}}{=} & P(\Pi, \mathrm{Emp}, S), \quad \text{where } \Sigma' \not\equiv \mathrm{Emp} \qquad \textbf{(EmpNEmp)} \\
P(\Pi, \mathrm{Emp}, \{(\Pi_i, \mathrm{Emp})\}_{i \in I}) & \stackrel{\mathrm{def}}{=} & \Pi \to \bigvee_{i \in I} \Pi_i \qquad \textbf{(EmpEmp)} \\
P(\Pi, \Sigma, \{(\Pi', \mathrm{Emp})\} \cup S) & \stackrel{\mathrm{def}}{=} & P(\Pi, \Sigma, S), \quad \text{where } \Sigma \not\equiv \mathrm{Emp} \qquad \textbf{(NEmpEmp)} \\
P(\Pi, \Sigma, \emptyset) & \stackrel{\mathrm{def}}{=} & \neg(\Pi \wedge \mathrm{Sorted}(\Sigma)) \qquad \textbf{(empty)}
\end{array}
$$

$$
\begin{array}{l}
P(\Pi, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u_i * \Sigma_i)\}_{i \in I}) \qquad (\mapsto\mapsto) \\
\qquad \stackrel{\mathrm{def}}{=} P(\Pi \wedge t < \Sigma, \Sigma, \{(\Pi_i \wedge t = t_i \wedge u = u_i \wedge t_i < \Sigma_i, \Sigma_i)\}_{i \in I})
\end{array}
$$

$$
\begin{array}{l}
P(\Pi, t \mapsto u * \Sigma, \{(\Pi_i, \mathrm{Arr}(t_i, t_i') * \Sigma_i)\} \cup S) \qquad \textbf{(}\mapsto\textbf{Arr)} \\
\qquad \stackrel{\mathrm{def}}{=} P(\Pi \wedge t_i' = t_i, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u * \mathrm{Arr}(t_i + 1, t_i') * \Sigma_i)\} \cup S) \\
\qquad\quad \wedge P(\Pi \wedge t_i' > t_i, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u * \mathrm{Arr}(t_i + 1, t_i') * \Sigma_i)\} \cup S) \\
\qquad\quad \wedge P(\Pi \wedge t_i' < t_i, t \mapsto u * \Sigma, S)
\end{array}
$$

$$
\begin{array}{l}
P(\Pi, \mathrm{Arr}(t, t') * \Sigma, S) \qquad \textbf{(Arr}\mapsto\textbf{)} \\
\quad \stackrel{\mathrm{def}}{=} P(\Pi \wedge t' > t, t \mapsto z * \mathrm{Arr}(t + 1, t') * \Sigma, S) \\
\qquad \wedge P(\Pi \wedge t' = t, t \mapsto z' * \Sigma, S), \quad \text{where } (\Pi'', t'' \mapsto u'' * \Sigma'') \in S \text{ and } z, z' \text{ are fresh}
\end{array}
$$

$$
\begin{array}{l}
P(\Pi, \mathrm{Arr}(t, t') * \Sigma, \{(\Pi_i, \mathrm{Arr}(t_i, t_i') * \Sigma_i)\}_{i \in I}) \qquad \textbf{(ArrArr)} \\
\stackrel{\mathrm{def}}{=} \bigwedge_{I' \subseteq I} P\left(\begin{array}{l} \Pi \wedge m = m_{I'} \wedge m < m_{I \setminus I'} \wedge t \le t' \wedge t' < \Sigma \wedge 0 < t, \Sigma, \\ \{(\Pi_i \wedge t = t_i \wedge t_i' < \Sigma_i, \Sigma_i)\}_{i \in I'} \cup \{(\Pi_i \wedge t = t_i, \mathrm{Arr}(t_i + m + 1, t_i') * \Sigma_i)\}_{i \in I \setminus I'} \end{array} \right) \\
\wedge \bigwedge_{\emptyset \neq I' \subseteq I} P\left(\begin{array}{l} \Pi \wedge m' < m \wedge m' = m_{I'} \wedge m' < m_{I \setminus I'} \wedge 0 < t, \mathrm{Arr}(t + m' + 1, t') * \Sigma, \\ \{(\Pi_i \wedge t = t_i \wedge t_i' < \Sigma_i, \Sigma_i)\}_{i \in I'} \cup \{(\Pi_i \wedge t = t_i, \mathrm{Arr}(t_i + m' + 1, t_i') * \Sigma_i)\}_{i \in I \setminus I'} \end{array} \right), \\
\qquad \text{where } m, m_i, \text{ and } m' \text{ are abbreviations of } t' - t,\ t_i' - t_i, \text{ and } m_{\min I'}, \text{ respectively.}
\end{array}
$$

Figure 1: The translation P

Σ' otherwise. This is obtained as follows: the points-to predicates at the first positions of Σ and Σ' are removed (if they exist), then the resulting formula made from Σ' is returned. $\mathrm{FirstRemove}_\Sigma(\{(\Pi_i, \Sigma_i)\}_{i \in I})$ is also defined by $\{(\Pi_i, \mathrm{FirstRemove}_\Sigma(\Sigma_i))\}_{i \in I}$.

The $\stackrel{\mathrm{def}}{=}$ steps terminate since the measure $(\sharp_*(\mathrm{FirstRemove}_\Sigma(S)), \sharp_*(\Sigma) + \sharp_*(S), |S|)$ for $P(\Pi, \Sigma, S)$ strictly decreases, where $|S|$ is the number of elements in S.

The formula $P(\Pi, \Sigma, \{(\Pi_i, \Sigma_i)\}_{i \in I})$ means that the QF entailment $\Pi \wedge \tilde{\Sigma} \vdash \{\Pi_i \wedge \tilde{\Sigma}\}_{i \in I}$ is valid. From this intuition, we sometimes call Σ the left spatial formula, and also call $\{\Sigma_i\}_{i \in I}$ the right spatial formulas. We call the left-most position of a spatial formula the head position. The atomic formula appears at the head position is called the head atom.

We will explain the meaning of each clause in Figure 1.

The clauses **(EmpL)** and **(EmpR)** just remove Emp at the head position.

The clause **(EmpNEmp)** handles the case where the left spatial formula is Emp. A pair (Π', Σ') in the third argument of P is removed if Σ' is not Emp, since $\Pi' \wedge \Sigma'$ cannot be satisfied by the empty heap.

The clause **(EmpEmp)** handles the case where the left formula and all the right spatial formulas are Emp. This case P returns a Presburger formula which is equivalent to the corresponding entailment is valid.

The clause **(NEmpEmp)** handles the case where the left spatial formula is not Emp and a pair (Π', Emp) appears in the third argument of P. We remove the pair since $\Pi' \wedge \mathrm{Emp}$ cannot be satisfied by any non-empty heap. For example, $P(\mathrm{True}, x \mapsto 0 * y \mapsto 0, \{(\mathrm{True}, \mathrm{Emp})\})$ becomes $P(\mathrm{True}, x \mapsto 0 * y \mapsto 0, \emptyset)$.

The clause **(empty)** handles the case where the third argument of P is empty. This case P returns a Presburger formula which is equivalent to that the left symbolic heap $\Pi \wedge \Sigma$ is not satisfiable. For example, $P(\mathrm{True}, x \mapsto 0 * y \mapsto 0, \emptyset)$ returns $\neg(x < y)$ using the fact that True is the unit of $\wedge$.

The clause **($\mapsto\mapsto$)** handles the case where all the head atoms of Σ and $\{\Sigma_i\}_{i \in I}$ are the points-to atomic formulas. This case we remove all of them and put equalities on the right pure parts. By this rule the measure is strictly reduced. For example, $P(\mathrm{True}, 3 \mapsto y * 4 \mapsto 11, \{(\mathrm{True}, x \mapsto y' * \mathrm{Arr}(4, 4))\})$ becomes $P(3 < 4, 4 \mapsto 11, \{(3 = x \wedge y = y' \wedge x < 4, \mathrm{Arr}(4, 4))\})$.

The clause **($\mapsto$Arr)** handles the case where the head atom of the left spatial formula is the points-to atomic formula and some right spatial formula Σ_i has the array atomic formula as its head atom. Then we split the array atomic formula into a points-to atomic formula and an array atomic formula for the rest. We have three subcases according to the size of the head array. The first case is when the size of the array is 1: We replace the head array by a points-to atomic formula. The second case is when the size of the head array is greater than 1: We split the head array atomic formula into a points-to atomic formula and an array atomic formula for the rest. The last case is when the size of the head array is less than 1: We just remove (Π_i, Σ_i), since the array atomic formula is false. We note that this rule can be applied repeatedly until all head array atomic formulas of the right spatial formulas are unfolded, since the left spatial formula is unchanged. Then the measure is eventually reduced by applying **($\mapsto\mapsto$)**. For example, $P(\mathrm{True}, x \mapsto 10, \{(\mathrm{True}, \mathrm{Arr}(4, 5))\})$ becomes

$$\begin{aligned} &P(5 = 4, x \mapsto 10, \{(\mathrm{True}, 4 \mapsto 10)\}) \\ &\quad \wedge P(5 > 4, x \mapsto 10, \{(\mathrm{True}, 4 \mapsto 10 * \mathrm{Arr}(5, 5))\}) \\ &\quad \wedge P(5 < 4, x \mapsto 10, \emptyset). \end{aligned}$$

The clause **(Arr$\mapsto$)** handles the case where the head atom of the left spatial formula is an array atomic formula and there is a right spatial formula whose head atom is a points-to atomic formula. We have two subcases according to the size of the head array. The first case is the case where the size of the array is greater than 1: The array atomic formula is split into a points-to atomic formula (with a fresh variable z) and an array atomic formula for the rest. The second case is when the size of the array is 1: The array atomic formula is unfolded and replaced by a points-to atomic formula with a fresh variable z'. We note that the left head atom becomes a points-to atomic formula after applying this rule. Hence the measure is eventually reduced, since **($\mapsto\mapsto$)** or **($\mapsto$Arr)** will be applied next. For example, $P(\mathrm{True}, \mathrm{Arr}(x, 3), \{(\mathrm{True}, y \mapsto 10)\})$ becomes

$$\begin{aligned} &P(x < 3, x \mapsto z * \mathrm{Arr}(x + 1, 3), \{(\mathrm{True}, y \mapsto 10)\}) \\ &\quad \wedge P(x = 3, x \mapsto z', \{(\mathrm{True}, y \mapsto 10)\}). \end{aligned}$$

The last clause **(ArrArr)** handles the case where all the head atoms in the left and right spatial formulas are array atomic formulas. We first find the head arrays of the shortest length among the head arrays. Next we split each longer array into two arrays so that the first part has the same size as the shortest array. Then we remove the first parts. The

shortest arrays are also removed. In this operation we have two subcases: The first case is when the array atomic formula of the left spatial formula has the shortest size and disappears by the operation. The second case is when the array atomic formula of the left spatial formula has a longer size, it is split into two array atomic formulas, and the second part remains. We note that the measure is strictly reduced, since at least one shortest array atomic formula is removed. For example, $P(\mathrm{True}, \mathrm{Arr}(x,5), \{(\mathrm{True}, \mathrm{Arr}(y,2) * \mathrm{Arr}(3,z))\})$ becomes

$$\begin{aligned}
&P(5-x=2-y \wedge x \leq 5 \wedge 0<x, \mathrm{Emp}, \{(x=y \wedge 2<3, \mathrm{Arr}(3,z))\}) \\
&\quad \wedge P(5-x<2-y \wedge x \leq 5 \wedge 0<x, \mathrm{Emp}, \\
&\qquad \{(x=y, \mathrm{Arr}(y+(5-x)+1,2) * \mathrm{Arr}(3,z))\}) \\
&\quad \wedge P(2-y<5-x \wedge 0<x, \mathrm{Arr}(x+(2-y)+1,5), \{(x=y \wedge 2<3, \mathrm{Arr}(3,z))\}).
\end{aligned}$$

This example is a case when I is a singleton set (we write $\{0\}$ for it), Π and Π_0 are True, Σ is Emp, Σ_0 is $\mathrm{Arr}(3,z)$, t is x, t' is 5, t_0 is y, t'_0 is 2 in **(ArrArr)**. The first clause of this result is obtained from the case $I' = I$ of the first conjunct in **(ArrArr)**, namely, $m = m_{I'}$ is $5-x=2-y$ and $m < m_{I \setminus I'}$ is True. The second clause is obtained from the case $I' = \emptyset$ of the first conjunct, namely, $m = m_{I'}$ is True and $m < m_{I \setminus I'}$ is $5-x<2-y$. The last clause is obtained from the case $I' = I$ of the second conjunct in **(ArrArr)**, namely, $m' < m$ is $2-y<5-x$, $m' = m_{I'}$ is $2-y=5-x$, and $m' < m_{I \setminus I'}$ is True.

Example 3.2. The sorted entailment $(x \mapsto 10 * v \mapsto 11)^{\sim} \vdash \mathrm{Arr}(x', v')^{\sim}$ is translated by computing $P(\mathrm{True}, x \mapsto 10 * v \mapsto 11, \{(\mathrm{True}, \mathrm{Arr}(x', v'))\})$. We will see its calculation step by step. It first becomes

$$P(x'=v', x \mapsto 10 * v \mapsto 11, \{(\mathrm{True}, x' \mapsto 10)\}) \tag{a}$$
$$\wedge\, P(x'<v', x \mapsto 10 * v \mapsto 11, \{(\mathrm{True}, x' \mapsto 10 * \mathrm{Arr}(x'+1, v'))\}) \tag{b}$$
$$\wedge\, P(x'>v', x \mapsto 10 * v \mapsto 11, \emptyset) \tag{c}$$

by ($\mapsto$Arr) taking Π to be True, $t \mapsto u$ to be $x \mapsto 10$, Σ to be $v \mapsto 11$, Π_i to be True, $\mathrm{Arr}(t_i, t'_i)$ to be $\mathrm{Arr}(x', v')$, Σ_i to be Emp, and S to be empty. The first conjunct (a) becomes $P(x'=v' \wedge x<v, v \mapsto 11, \{(x'=x \wedge 10=10, \mathrm{Emp})\})$ by ($\mapsto\mapsto$) taking Π to be $x'=v'$, $t \mapsto u$ to be $x \mapsto 10$, Σ to be $v \mapsto 11$, I to be the singleton set $\{0\}$, Π_0 to be True, $t_0 \mapsto u_0$ to be $x' \mapsto 10$, and Σ_0 to be Emp. Then it becomes

$$\neg(x'=v' \wedge x<v) \tag{a'}$$

by (NEmpEmp) and (empty). The third conjunct (c) becomes

$$\neg(x'>v' \wedge x<v) \tag{c'}$$

by (empty) taking Π to be $x'>v'$ and Σ to be $x \mapsto 10 * v \mapsto 11$. The second conjunct (b) becomes $P(x'<v' \wedge x<v, v \mapsto 11, \{(x=x' \wedge 10=10, \mathrm{Arr}(x'+1, v'))\})$ by ($\mapsto\mapsto$) taking Π to be $x'<v'$, $t \mapsto u$ to be $x \mapsto 10$, Σ to be $v \mapsto 11$, I to be the singleton set $\{0\}$, Π_0 to be True, $t_0 \mapsto u_0$ to be $x' \mapsto 10$, and Σ_0 to be $\mathrm{Arr}(x'+1, v')$, then it becomes

$$P(x'<v' \wedge x<v \wedge x'+1=v', v \mapsto 11, \{(x=x' \wedge 10=10, x'+1 \mapsto 11)\}) \tag{b1}$$
$$\wedge\, P(x'<v' \wedge x<v \wedge x'+1<v', v \mapsto 11, \{(x=x' \wedge 10=10, x'+1 \mapsto 11 * \mathrm{Arr}(x'+2, v'))\}) \tag{b2}$$
$$\wedge\, P(x'<v' \wedge x<v \wedge x'+1>v', v \mapsto 11, \emptyset) \tag{b3}$$

by ($\mapsto$Arr) taking Π to be $x' < v' \wedge x < v$, $t \mapsto u$ to be $v \mapsto 11$, Σ to be Emp, Π_i to be $x = x' \wedge 10 = 10$, $\mathrm{Arr}(t_i, t_i')$ to be $\mathrm{Arr}(x'+1, v')$, Σ_i to be Emp and S to be $\emptyset$. Hence we have

$$\begin{aligned} & P(x' < v' \wedge x < v \wedge x' + 1 = v', \mathrm{Emp}, \\ & \qquad \{(x = x' \wedge 10 = 10 \wedge v = x' + 1 \wedge 11 = 11, \mathrm{Emp})\}) && \text{(b1')} \\ & \wedge P(x' < v' \wedge x < v \wedge x' + 1 < v', \mathrm{Emp}, \\ & \qquad \{(x = x' \wedge 10 = 10 \wedge v = x' + 1 \wedge 11 = 11, \mathrm{Arr}(x'+2, v'))\}) && \text{(b2')} \\ & \wedge \neg(x' < v' \wedge x < v \wedge x' + 1 > v') && \text{(b3')} \end{aligned}$$

by ($\mapsto\mapsto$) and (empty). We note that (b2') becomes $P(x' < v' \wedge x < v \wedge x' + 1 < v', \mathrm{Emp}, \emptyset\})$ by (EmpNEmp) taking Π to be $x' < v' \wedge x < v \wedge x' + 1 < v'$, Π' to be $x = x' \wedge 10 = 10 \wedge v = x' + 1 \wedge 11 = 11$, Σ' to be $\mathrm{Arr}(x'+2, v')$, and S to be $\emptyset$. Thus we obtain

$$\begin{aligned} & \big((x' < v' \wedge x < v \wedge x' + 1 = v') \rightarrow (x = x' \wedge 10 = 10 \wedge v = x' + 1 \wedge 11 = 11)\big) && \text{(b21)} \\ & \qquad \wedge \neg(x' < v' \wedge x < v \wedge x' + 1 < v') \wedge \neg(x' < v' \wedge x < v \wedge x' + 1 > v'), && \text{(b22)} \end{aligned}$$

where (b21) is obtained from (b1') by applying (EmpEmp), and (b22) is obtained from (b2') and (b3') by applying (empty). Finally, by combining (a'), (b21), (b22), and (c'), we have

$$\begin{aligned} & \neg(x' = v' \wedge x < v) \\ & \wedge \big((x' < v' \wedge x < v \wedge x' + 1 = v') \rightarrow (x = x' \wedge 10 = 10 \wedge v = x' + 1 \wedge 11 = 11)\big) \\ & \wedge \neg(x' < v' \wedge x < v \wedge x' + 1 < v') \wedge \neg(x' < v' \wedge x < v \wedge x' + 1 > v') \\ & \wedge \neg(x' > v' \wedge x < v) \end{aligned}$$

as the translation result of $P(\mathrm{True}, x \mapsto 10 * v \mapsto 11, \{(\mathrm{True}, \mathrm{Arr}(x', v'))\})$.

3.3. **Decidability Theorem.** The aim of P is to give an equivalent formula of Presburger arithmetic to a given entailment. The correctness property of P is stated as follows.

Proposition 3.3 (Correctness of Translation P)**.** *If any array atomic formula in Σ_i has the form $\mathrm{Arr}(t, t+u)$ such that the term u does not contain $\overrightarrow{y}$, then*

$$\Pi \wedge \tilde{\Sigma} \models \{\exists \overrightarrow{y_i}(\Pi_i \wedge \tilde{\Sigma}_i)\}_{i \in I} \quad \textit{iff} \quad \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, \{(\Pi_i, \Sigma_i)\}_{i \in I})$$

where $\overrightarrow{y}$ is a sequence of $\overrightarrow{y_i}$ ($i \in I$), and $\overrightarrow{z}$ is $FV(P(\Pi, \Sigma, \{(\Pi_i, \Sigma_i)\}_{i \in I})) \setminus FV(\Pi, \Sigma, \{\Pi_i\}_{i \in I}, \{\Sigma_i\}_{i \in I})$.

We note that $\overrightarrow{z}$ are the fresh variables introduced in the unfolding of $P(\Pi, \Sigma, \{(\Pi_i, \Sigma_i)\}_{i \in I})$. The proof of this theorem will be given in the next section.

The correctness property is shown with the condition described in the theorem. This condition avoids a complicated situation for $\overrightarrow{y}$ and $\overrightarrow{z}$, such that some variables in $\overrightarrow{y}$ depend on $\overrightarrow{z}$, and some determine $\overrightarrow{z}$. For example, if we consider $\mathrm{Arr}(1,5) \vdash \exists y_1 y_2(\mathrm{Arr}(1, y_1) * y_1 + 1 \mapsto y_2 * \mathrm{Arr}(y_1 + 2, 5))$, we will have $y_1 + 1 \mapsto z$ during the unfolding of $P(\mathrm{True}, \mathrm{Arr}(1,5), \{(\mathrm{True}, \mathrm{Arr}(1, y_1) * y_1 + 1 \mapsto y_2 * \mathrm{Arr}(y_1 + 2, 5))\})$. Then we have $z = y_2$ after some logical simplification. This fact means that y_2 depends on z, and moreover z is indirectly determined by y_1. The latter case occurs when sizes of array depend on $\overrightarrow{y}$. We need to exclude this situation.

Finally we have the decidability result for the entailment problem of **SLA** under the condition from the above theorem and the property of sorted entailments (stated in Lemma 3.1).

Theorem 3.4 (Decidability of Validity Checking of Entailments in **SLA**). *Validity checking of entailments* $\Pi \wedge \Sigma \vdash \{\exists \overrightarrow{y_i}(\Pi_i \wedge \Sigma_i)\}_{i \in I}$ *in* **SLA** *is decidable, if any array atomic formula in* Σ_i *has the form* $\mathrm{Arr}(t, t+u)$ *such that the term* u *does not contain* $\overrightarrow{y_i}$.

Example 3.5. An example $\mathrm{Arr}(x,x) \vdash x \mapsto 0, \exists y(y > 0 \wedge x \mapsto y)$ satisfies the condition, and its validity is checked in the following way.

- It is decomposed into several sorted entailments: in this case, it produces one sorted entailment $\mathrm{Arr}(x,x)^{\sim} \vdash x \mapsto 0, \exists y(y > 0 \wedge x \mapsto y)$
- Compute $P(\mathrm{True}, \mathrm{Arr}(x,x), S_1)$, where S_1 is $\{(\mathrm{True}, x \mapsto 0), (y > 0, x \mapsto y)\}$. It becomes $P(x < x, x \mapsto z * \mathrm{Arr}(x+1, x), S_1) \wedge P(x = x, x \mapsto z, S_1)$ by (Arr$\mapsto$). Then it becomes

  $$P(x < x \wedge x < x+1, \mathrm{Arr}(x+1, x), S_2) \wedge P(x = x \wedge x < x+1, \mathrm{Emp}, S_2),$$

  where S_2 is $\{(x = x \wedge z = 0, \mathrm{Emp}), (y > 0 \wedge x = x \wedge z = y, \mathrm{Emp})\}$. The former conjunct becomes $P(x < x \wedge x < x+1, \mathrm{Arr}(x+1, x), \emptyset)$ by (NEmpEmp), then, by (empty), it becomes $\neg(x < x \wedge x < x+1 \wedge x+1 \leq x)$. The latter conjunct becomes $x = x \wedge x < x+1 \to (x = x \wedge z = 0) \vee (y > 0 \wedge x = x \wedge z = y)$, which is equivalent to $x < x+1 \to z = 0 \vee (y > 0 \wedge z = y)$,
- Check the validity of the formula $\forall xz \exists y P(\mathrm{True}, \mathrm{Arr}(x,x), S_1)$, which is equivalent to $\forall xz \exists y(\neg(x < x \wedge x < x+1 \wedge x+1 \leq x) \wedge (z = 0 \vee (y > 0 \wedge z = y)))$. Finally the procedure answers "valid", since the produced Presburger formula is valid.

3.4. **Other Systems of Symbolic Heaps with Arrays.** Other known systems of symbolic heaps with arrays are only the system given in Brotherston et al. [7]. They gave a different condition for decidability of the entailment problem for the same symbolic-heap system. Their condition disallows existential variables in u for each points-to atomic formula $t \mapsto u$ in the succedent of an entailment. In order to clarify the difference between our condition and their condition, we consider the following entailments:

(i) $\mathrm{Arr}(x,x) \vdash x \mapsto 0, \exists y(y > 0 \wedge x \mapsto y)$,
(ii) $\mathrm{Arr}(1,5) \vdash \exists y, y'(\mathrm{Arr}(y, y+1) * \mathrm{Arr}(y', y'+2))$,
(iii) $\mathrm{Arr}(1,5) \vdash \exists y(\mathrm{Arr}(1, 1+y) * \mathrm{Arr}(2+y, 5))$,
(iv) $\mathrm{Arr}(1,5) \vdash \exists y, y'(\mathrm{Arr}(1, 1+y) * 2+y \mapsto y' * \mathrm{Arr}(3+y, 5))$.

The entailment (i) can be decided by our decision procedure, but it cannot be decided by their procedure. The entailment (ii) is decided by both theirs and ours. The entailment (iii) is decided by theirs, but it does not satisfy our condition. The entailment (iv) is decided by neither theirs nor ours.

Our system and the system in [7] have the same purpose, namely, analysis/verification of memory safety. Basically their target programming language and assertion language are the same as ours given in this section. These entailment checkers are essentially used for deciding the side condition of the consequence rule. As explained above, ours and theirs have different restrictions for decidability. Hence the class of programs is the same for ours and theirs, but some triples can be proved only by ours and other triples can be proved

only by theirs, according to the shape of assertions. In this sense both our system and their system have advantage and disadvantage.

4. Correctness of Decision Procedure

This section shows correctness of our decision procedure. We first show the basic property of sorted entailments.

4.1. Correctness of Translation.

This subsection shows correctness of the translation P. The main difficulty for showing correctness is how to handle the new variables (denoted by z) that are introduced during the unfolding P. In order to do this, we temporarily extend our language with new terms, denoted by $[t]$. A term $[t]$ means the value at the address t, that is, it is interpreted to $h(s(t))$ under (s,h). We will use this notation instead of z, since z must appear in the form $t \mapsto z$ during unfolding P, and this t is unique for z. Note that both s and h are necessary for interpreting a formula of the extended language even if it is a pure formula.

In this extended language, we temporarily introduce a variant P' of P so that we use $[t]$ instead of z, which is defined in the same way as P except

$$\begin{aligned} P'(\Pi, \mathrm{Arr}(t,t') * \Sigma, S) \stackrel{\mathrm{def}}{=} & P'(\Pi \wedge t' = t, t \mapsto [t] * \Sigma, S) \\ & \wedge P'(\Pi \wedge t' > t, t \mapsto [t] * \mathrm{Arr}(t+1, t') * \Sigma, S), \end{aligned}$$

when $(\Pi'', t'' \mapsto u'' * \Sigma'') \in S$. Note that P' never introduces any new variables.

We will introduce some notations. Let S be $\{(\Pi, \Sigma)\}_{i \in I}$. Then we write $\tilde{S}$ for $\{\Pi_i \wedge \tilde{\Sigma_i}\}_{i \in I}$. We write $\mathrm{Dom}(s, \Sigma)$ for the set of addresses used by Σ under s, that is, it is inductively defined as follows: $\mathrm{Dom}(s, \mathrm{Emp}) = \emptyset$, $\mathrm{Dom}(s, \mathrm{Emp} * \Sigma_1) = \mathrm{Dom}(s, \Sigma_1)$, $\mathrm{Dom}(s, t \mapsto u * \Sigma_1) = \{s(t)\} \cup \mathrm{Dom}(s, \Sigma_1)$, and $\mathrm{Dom}(s, \mathrm{Arr}(t,u) * \Sigma_1) = \{s(t), \ldots, s(u)\} \cup \mathrm{Dom}(s, \Sigma_1)$ if $s(t) \leq s(u)$.

The next lemma clarifies the connections between entailments, P, and P'.

Lemma 4.1. (1) *Assume* $s, h \models \hat{\Pi} \wedge \hat{\Sigma}$. *Suppose* $P'(\Pi, \Sigma, S)$ *appears in the unfolding of* $P'(\hat{\Pi}, \hat{\Sigma}, \hat{S})$. *Then*

$s, h|_{\mathrm{Dom}(s,\Sigma)} \models P'(\Pi, \Sigma, S)$ *iff* $s, h|_{\mathrm{Dom}(s,\Sigma)} \models \Pi \wedge \mathrm{Sorted}(\Sigma) \to \bigvee \tilde{S}$.

(2) $\forall sh(s, h \models \Pi \wedge \tilde{\Sigma} \to s, h \models \exists \overrightarrow{y} P'(\Pi, \Sigma, S))$ *iff* $\Pi \wedge \tilde{\Sigma} \models \exists \overrightarrow{y} \bigvee \tilde{S}$.

(3) $\models \neg(\Pi \wedge \mathrm{Sorted}(\Sigma)) \to P(\Pi, \Sigma, S)$.

(4) *Let* $\overrightarrow{z}$ *be fresh variables introduced during the calculation of* $P(\Pi, \Sigma, S)$. *Then* $\forall sh(s, h \models \Pi \wedge \tilde{\Sigma} \to s, h \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S))$ *iff* $\models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$.

Proof of Lemma 4.1 (1). This is shown by induction on the steps $\stackrel{\mathrm{def}}{=}$. Consider cases according to the definition of P'.

Case 1 ($\mapsto\mapsto$-case):

$$\begin{aligned} & P'(\Pi, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u_i * \Sigma_i)\}_{i \in I}) \\ & \quad \stackrel{\mathrm{def}}{=} P'(\Pi \wedge t < \Sigma, \Sigma, \{(\Pi_i \wedge t = t_i \wedge u = u_i \wedge t_i < \Sigma_i, \Sigma_i)\}_{i \in I}). \end{aligned}$$

Let $h_1 = h|_{\mathrm{Dom}(s, t \mapsto u * \Sigma)}$, $h_2 = h|_{\mathrm{Dom}(s,\Sigma)}$. Then $h_1 = \{(s(t), h(s(t)))\} + h_2$. It is enough to show

$$s, h_1 \models \Pi \wedge \mathrm{Sorted}(t \mapsto u * \Sigma) \to \bigvee_{i \in I} \Pi_i \wedge (t_i \mapsto u_i * \Sigma_i)^{\sim} \qquad \text{(a)}$$

iff

$$s, h_2 \models \Pi \wedge t < \Sigma \wedge \mathrm{Sorted}(\Sigma) \rightarrow \bigvee_{i \in I} \Pi_i \wedge t = t_i \wedge u = u_i \wedge t_i < \Sigma_i \wedge \Sigma_i^{\sim}. \tag{b}$$

The only-if part. Assume (a) and the antecedent of (b). Then the antecedent of (a) holds, since it is equivalent to the antecedent of (b). Then the succedent of (a) is true for s, h_1. Hence the succedent of (b) is true for s, h_2.

The if part. Assume (b) and the antecedent of (a). Then the antecedent of (b) holds, since it is equivalent to the antecedent of (a). Then the succedent of (b) is true for s, h_2. Hence the succedent of (a) is true for s, h_1.

Case 2 (**Arr**$\mapsto$-case):

$$\begin{aligned} P'(\Pi, \mathrm{Arr}(t, t') * \Sigma, S) \stackrel{\mathrm{def}}{=} & P'(\Pi \wedge t' = t \wedge t \leq \Sigma, \Sigma, S) \\ & \wedge P'(\Pi \wedge t' > t, t \mapsto [t] * \mathrm{Arr}(t+1, t') * \Sigma, S). \end{aligned}$$

Let $h_3 = h|_{\mathrm{Dom}(s, \mathrm{Arr}(t,t') * \Sigma)}$, $h_4 = h|_{\mathrm{Dom}(s, t \mapsto [t] * \Sigma)}$, and $h_5 = h|_{\mathrm{Dom}(s, t \mapsto [t] * \mathrm{Arr}(t+1,t') * \Sigma)}$. It is enough to show

$$s, h_3 \models \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma) \rightarrow \bigvee \tilde{S} \tag{c}$$

is equivalent to the conjunction of the following two clauses:

$$s, h_4 \models \Pi \wedge t' = t \wedge t < \Sigma \wedge \mathrm{Sorted}(\Sigma) \rightarrow \bigvee \tilde{S} \tag{d}$$

and

$$s, h_5 \models \Pi \wedge t' > t \wedge \mathrm{Sorted}(t \mapsto [t] * \mathrm{Arr}(t+1, t') * \Sigma) \rightarrow \bigvee \tilde{S}. \tag{e}$$

Case 2.1: the case of $s(t) = s(t')$.

We note that $h_3 = h_4$. The antecedent of (d) is equivalent to the antecedent of (c). (e) is true since $s(t) = s(t')$. Hence (c) and $(d) \wedge (e)$ are equivalent.

Case 2.2: the case of $s(t') > s(t)$.

We note that $h_3 = h_5$. The antecedent of (e) is equivalent to the antecedent of (c). (d) is true since $s(t') > s(t)$. Hence (c) and $(d) \wedge (e)$ are equivalent.

Case 3 ($\mapsto$**Arr**-case):

$$\begin{aligned} & P'(\Pi, t \mapsto u * \Sigma, \{(\Pi_i, \mathrm{Arr}(t_i, t_i') * \Sigma_i)\} \cup S) \\ & \quad \stackrel{\mathrm{def}}{=} P'(\Pi \wedge t_i' = t_i, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u * \Sigma_i)\} \cup S) \\ & \qquad \wedge P'(\Pi \wedge t_i' > t_i, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u * \mathrm{Arr}(t_i + 1, t_i') * \Sigma_i)\} \cup S) \\ & \qquad \wedge P'(\Pi \wedge t_i' < t_i, t \mapsto u * \Sigma, S) \end{aligned}$$

This case is proved by showing the following claim, which is shown similarly to the claim of Case 2. Let $h' = h|_{\mathrm{Dom}(s, t \mapsto u * \Sigma)}$. Let Sorted_L be an abbreviation of $\mathrm{Sorted}(t \mapsto u * \Sigma)$. Then

$$s, h' \models \Pi \wedge \mathrm{Sorted}_L \rightarrow \Pi_i \wedge (\mathrm{Arr}(t_i, t_i') * \Sigma_i)^{\sim} \vee \bigvee \tilde{S}$$

is equivalent to the conjunction of the following three clauses:

$$s, h' \models \Pi \wedge t'_i = t_i \wedge \mathrm{Sorted}_L \to \Pi_i \wedge (t_i \mapsto u * \Sigma_i)^{\sim} \vee \bigvee \tilde{S}$$
$$s, h' \models \Pi \wedge t'_i > t_i \wedge \mathrm{Sorted}_L \to \Pi_i \wedge (t_i \mapsto u * \mathrm{Arr}(t_i + 1, t'_i) * \Sigma_i)^{\sim} \vee \bigvee \tilde{S},$$
$$s, h' \models \Pi \wedge t'_i < t_i \wedge \mathrm{Sorted}_L \to \bigvee \tilde{S}.$$

Case 4 (**(ArrArr)**-case): Consider that $P'(\Pi, \mathrm{Arr}(t, t') * \Sigma, \{(\Pi_i, \mathrm{Arr}(t_i, t'_i) * \Sigma_i)\}_{i \in I})$ is defined by the conjunction of

$$P'\left(\begin{array}{l} \Pi \wedge m = m_{I'} \wedge m < m_{I \setminus I'} \wedge t \leq t' \wedge t' < \Sigma \wedge 0 < t, \Sigma, \\ \{(\Pi_i \wedge t = t_i \wedge t'_i < \Sigma_i, \Sigma_i)\}_{i \in I'} \cup \{(\Pi_i \wedge t = t_i, \mathrm{Arr}(t_i + m + 1, t'_i) * \Sigma_i)\}_{i \in I \setminus I'} \end{array}\right)$$

for all $I' \subseteq I$ and

$$P'\left(\begin{array}{l} \Pi \wedge m' < m \wedge m' = m_{I'} \wedge m' < m_{I \setminus I'} \wedge 0 < t, \mathrm{Arr}(t + m' + 1, t') * \Sigma, \\ \{(\Pi_i \wedge t = t_i \wedge t'_i < \Sigma_i, \Sigma_i)\}_{i \in I'} \cup \{(\Pi_i \wedge t = t_i, \mathrm{Arr}(t_i + m' + 1, t'_i) * \Sigma_i)\}_{i \in I \setminus I'} \end{array}\right)$$

for all $I' \subseteq I$ with $I' \neq \emptyset$, where m, m_i, and m' are abbreviations of $t' - t$, $t'_i - t_i$, and $m_{\min I'}$, respectively.

Let $h_6 = h|_{\mathrm{Dom}(s, \mathrm{Arr}(t,t') * \Sigma)}$, $h_7 = h|_{\mathrm{Dom}(s,\Sigma)}$, and $h_8 = h|_{\mathrm{Dom}(s, \mathrm{Arr}(t+m'+1,t') * \Sigma)}$. It is enough to show

$$s, h_6 \models \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma) \to \bigvee_{i \in I} \Pi_i \wedge (\mathrm{Arr}(t_i, t'_i) * \Sigma_i))^{\sim} \tag{f}$$

is equivalent to the conjunction of the following

$$\begin{aligned} s, h_7 \models{}& \Pi \wedge m = m_{I'} \wedge m < m_{I \setminus I'} \wedge 0 < t \wedge t \leq t' \wedge t' < \Sigma \wedge \mathrm{Sorted}(\Sigma) \\ &\to \bigvee_{i \in I'} \Pi_i \wedge t = t' \wedge t'_i < \Sigma_i \wedge \Sigma_i^{\sim} \\ &\quad \vee \bigvee_{i \in I \setminus I'} \Pi_i \wedge t = t' \wedge (\mathrm{Arr}(t_i + m' + 1, t'_i) * \Sigma_i)^{\sim} \end{aligned} \tag{g}$$

for any $I' \subseteq I$, and

$$\begin{aligned} s, h_8 \models{}& \Pi \wedge m' < m \wedge m' = m_{I'} \wedge m' < m_{I \setminus I'} \wedge 0 < t \wedge \mathrm{Sorted}(\mathrm{Arr}(t + m' + 1, t') * \Sigma) \\ &\to \bigvee_{i \in I'} \Pi_i \wedge t = t' \wedge t'_i < \Sigma_i \wedge \Sigma_i^{\sim} \\ &\quad \vee \bigvee_{i \in I \setminus I'} \Pi_i \wedge t = t' \wedge (\mathrm{Arr}(t_i + m' + 1, t'_i) * \Sigma_i)^{\sim} \end{aligned} \tag{h}$$

for any $I' \subseteq I$ with $I' \neq \emptyset$.

Case 4.1: the case of $s \models m = m_{I'} \wedge m < m_{I \setminus I'}$ for some $I' \subseteq I$.

The antecedent of the conjunct of the form of (g) with I' being this I' satisfies the case condition. The conjunct of the form (g) with I' begin not this I' and the conjuncts of the form (h) are true, because their antecedents are false by the case condition.

Now we show the only-if part and the if part by using $h_6 = h|_{\{s(t), s(t+1), \ldots, s(t')\}} + h_7$.

The only-if part: Suppose (f) is true. By the above observation it is enough to consider the conjunct of the form (g) with this I'. Assume the antecedent of the conjunct is true for s, h_7. Then its succedent is also true for s, h_7 since the succedent of (f) is true for s, h_6. Therefore the form of (g) with I' being this I' holds.

The if part: Suppose that the forms of (g) for any $I' \subseteq I$ hold, and the forms of (h) for any I' such that $\emptyset \neq I' \subseteq I$ hold. Assume that the antecedent of (f) is true for s, h_6. Then the succedent of the form (g) with I' being this I' holds for s, h_7 since its antecedent is true. Hence the succedent of (f) is true for s, h_6. Therefore (f) holds.

Case 4.2: the case of $s \models m' < m \wedge m' = m_{I'} \wedge m' < m_{I \setminus I'}$ for some $I' \subseteq I$. It is similar to Case 4.1 by using $h_6 = h|_{\{s(t),\ldots,s(t+m')\}} + h_8$.

Proof of Lemma 4.1 (2). We first show the only-if part. Assume the left-hand side of the claim and $s, h \models \Pi \wedge \tilde{\Sigma}$. By the left-hand side, we obtain $s, h \models \exists \overrightarrow{y} P'(\Pi, \Sigma, S)$. Hence we have s' such that $s', h \models P'(\Pi, \Sigma, S)$. By (1), $s', h \models \Pi \wedge \mathrm{Sorted}(\Sigma) \rightarrow \bigvee S$. Thus $s', h \models \bigvee S$. Finally we have $s, h \models \exists \overrightarrow{y} \bigvee S$.

Next we show the if part. Fix s, h. Assume the right-hand side of the claim and $s, h \models \Pi \wedge \tilde{\Sigma}$. By the right-hand side, $s, h \models \exists \overrightarrow{y} \bigvee \tilde{S}$ holds. Hence we have s' such that $s', h \models \bigvee S$. Then we obtain $s', h \models \Pi \wedge \mathrm{Sorted}(\Sigma) \rightarrow \bigvee \tilde{S}$. By (1), $s', h \models P'(\Pi, \Sigma, S)$ holds. Finally we have $s, h \models \exists \overrightarrow{y} P'(\Pi, \Sigma, S)$.

Proof of Lemma 4.1 (3). It is shown by induction on the steps of $\stackrel{\mathrm{def}}{=}$. Consider cases according to the definition of P.

Case 1 ((**EmpL**)-case): By the induction hypothesis we have $\models \neg(\Pi \wedge \mathrm{Sorted}(\Sigma)) \rightarrow P(\Pi, \Sigma, S)$. This is equivalent to $\models \neg(\Pi \wedge \mathrm{Sorted}(\mathrm{Emp} * \Sigma)) \rightarrow P(\Pi, \mathrm{Emp} * \Sigma, S)$.

The other cases (**EmpR**), (**EmpNEmp**), (**NEmpEmp**), and ($\mapsto\mapsto$) are shown in a similar way. We will consider the remaining cases.

Case 2 ((**EmpEmp**)-case): This case is easily shown, since $\models \neg(\Pi \wedge \mathrm{Sorted}(\mathrm{Emp})) \rightarrow (\Pi \rightarrow \bigvee_{i \in I} \Pi_i)$ trivially holds.

Case 3 ((**empty**)-case): This case is easily shown, since $\models \neg(\Pi \wedge \mathrm{Sorted}(\mathrm{Emp})) \rightarrow \neg(\Pi \wedge \mathrm{Sorted}(\mathrm{Emp}))$ trivially holds.

Case 4 (($\mapsto$**Arr**)-case): By the induction hypothesis, we have

$$\models \neg(\Pi \wedge t'_i = t_i \wedge \mathrm{Sorted}(t \mapsto u * \Sigma)) \rightarrow P^{(=)},$$

where $P^{(=)}$ is an abbreviation of $P(\Pi \wedge t'_i = t_i, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u * \Sigma_i)\} \cup S)$,

$$\models \neg(\Pi \wedge t'_i > t_i \wedge \mathrm{Sorted}(t \mapsto u * \Sigma)) \rightarrow P^{(>)},$$

where $P^{(>)}$ is an abbreviation of $P(\Pi \wedge t'_i > t_i, t \mapsto u * \Sigma, \{(\Pi_i, t_i \mapsto u * \mathrm{Arr}(t_i + 1, t'_i) * \Sigma_i)\} \cup S)$, and

$$\models \neg(\Pi \wedge t'_i < t_i \wedge \mathrm{Sorted}(t \mapsto u * \Sigma)) \rightarrow P^{(<)},$$

where $P^{(<)}$ is an abbreviation of $P(\Pi \wedge t'_i < t_i, t \mapsto u * \Sigma, S)$. In order to show the current case, it is enough to show $\Pi \wedge t'_i = t_i \wedge \mathrm{Sorted}(t \mapsto u * \Sigma) \rightarrow \Pi \wedge \mathrm{Sorted}(t \mapsto u * \Sigma)$, $\Pi \wedge t'_i > t_i \wedge \mathrm{Sorted}(t \mapsto u * \Sigma) \rightarrow \Pi \wedge \mathrm{Sorted}(t \mapsto u * \Sigma)$, and $\Pi \wedge t'_i < t_i \wedge \mathrm{Sorted}(t \mapsto u * \Sigma) \rightarrow \Pi \wedge \mathrm{Sorted}(t \mapsto u * \Sigma)$. They trivially hold by comparing conjuncts.

Case 5 (($\mathbf{Arr}\mapsto$)-case): By the induction hypothesis, we have

$$\models \neg(\Pi \wedge t' > t \wedge \mathrm{Sorted}(t \mapsto z * \mathrm{Arr}(t + 1, t') * \Sigma)) \rightarrow P^{(>)},$$

where $P^{(>)}$ is an abbreviation of $P(\Pi \wedge t' > t, t \mapsto z * \mathrm{Arr}(t + 1, t') * \Sigma, S)$, and

$$\models \neg(\Pi \wedge t' = t \wedge \mathrm{Sorted}(t \mapsto z' * \Sigma)) \rightarrow P^{(=)},$$

where $P^{(=)}$ is an abbreviation of $P(\Pi \wedge t' = t, t \mapsto z' * \Sigma, S)$. In order to show the current case, it is enough to show $\Pi \wedge t' > t \wedge \mathrm{Sorted}(t \mapsto z * \mathrm{Arr}(t+1, t') * \Sigma) \to \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma)$ and $\Pi \wedge t' = t \wedge \mathrm{Sorted}(t \mapsto z' * \Sigma) \to \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma)$. They are immediately shown by the definition of Sorted.

Case 6 ((**ArrArr**)-case): For $I' \subseteq I$, we abbreviate $m = m_{I'} \wedge m < m_{I \setminus I'}$ and $m' < m \wedge m' = m_{I'} \wedge m' < m_{I \setminus I'}$ by $\Pi_{I'}^{(=)}$ and $\Pi_{I'}^{(<)}$, respectively. We write $S_{I'}^{(=)}$ and $S_{I'}^{(<)}$ for the third arguments of the first and the second P for I' in the right-hand side of (ArrArr), respectively. We also write Arr for $\mathrm{Arr}(t + m' + 1, t')$. By the induction hypothesis, we obtain the forms of

$$\models \neg(\Pi \wedge \Pi_{I'}^{(=)} \wedge t \leq t' \wedge t' < \Sigma \wedge 0 < t \wedge \mathrm{Sorted}(\Sigma)) \\ \to P(\Pi \wedge \Pi_{I'}^{(=)} \wedge t \leq t' \wedge t' < \Sigma \wedge 0 < t, \Sigma, S_{I'}^{(=)})$$

for each $I' \subseteq I$, and

$$\models \neg(\Pi \wedge \Pi_{I'}^{(<)} \wedge 0 < t \wedge \mathrm{Sorted}(\mathrm{Arr} * \Sigma)) \to P(\Pi \wedge \Pi_{I'}^{(<)} \wedge 0 < t, \mathrm{Arr} * \Sigma, S_{I'}^{(<)})$$

for each $I' \subseteq I$ with $I' \neq \emptyset$. In order to show the current case, it is enough to show $\Pi \wedge \Pi_{I'}^{(=)} \wedge t \leq t' \wedge t' < \Sigma \wedge 0 < t \wedge \mathrm{Sorted}(\Sigma) \to \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma)$ for each $I' \subseteq I$, and $\Pi \wedge \Pi_{I'}^{(<)} \wedge 0 < t \wedge \mathrm{Sorted}(\mathrm{Arr} * \Sigma) \to \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma)$ for each $I' \subseteq I$ with $I' \neq \emptyset$. They are immediately shown by the definition of Sorted.

Proof of Lemma 4.1 (4). We note that $\forall sh(s, h \models \Pi \wedge \tilde{\Sigma} \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S))$ is equivalent to $\forall s(\exists h(s, h \models \Pi \wedge \tilde{\Sigma}) \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S))$. Moreover it is equivalent to $\forall s(s \models \Pi \wedge \mathrm{Sorted}(\Sigma) \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S))$. By (3), we have $\models \neg(\Pi \wedge \mathrm{Sorted}(\Sigma)) \to \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$, since $\Pi \wedge \mathrm{Sorted}(\Sigma)$ does not contain $\overrightarrow{z}$, which are fresh variables introduced during the calculation of $P(\Pi, \Sigma, S)$. Hence we obtain $\models (\Pi \wedge \mathrm{Sorted}(\Sigma)) \vee \neg(\Pi \wedge \mathrm{Sorted}(\Sigma)) \to \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$. This is equivalent to $\models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$. □

4.2. **Decidability Proof.** This subsection proves the correctness of the decision procedure.

Proof of Proposition 3.3 Let S be $\{(\Pi_i, \Sigma_i)\}_{i \in I}$. Then the left-hand side is equivalent to $\Pi \wedge \tilde{\Sigma} \models \exists \overrightarrow{y} \bigvee \tilde{S}$. Moreover, by Lemma 4.1 (2), it is equivalent to

$$\forall sh(s, h \models \Pi \wedge \tilde{\Sigma} \to s, h \models \exists \overrightarrow{y} P'(\Pi, \Sigma, S)). \tag{i}$$

By Lemma 4.1 (4), the right-hand side is equivalent to

$$\forall sh(s, h \models \Pi \wedge \tilde{\Sigma} \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)). \tag{j}$$

Now we will show the equivalence of (i) and (j). Here we assume $[t_1], \ldots, [t_n]$ appear in $P'(\Pi, \Sigma, S)$ and $s \models t_1 < \ldots < t_n$, we let $\overrightarrow{z} = z_1, \ldots, z_n$.

Recall that our condition requires that sizes of arrays in the succedent do not depend on existential variables. We note that, under the condition of Theorem 3.4, each t of $t \mapsto u$ or $\mathrm{Arr}(t, t')$ in the second argument of P' during the unfolding of P' does not contain any existential variables. By this fact, we can see that each term $[t]$ does not contain existential variables, since it first appears as $t \mapsto [t]$ in the second argument of P' during the unfolding of P'. So we can obtain $P'(\Pi, \Sigma, S) = P(\Pi, \Sigma, S)[\overrightarrow{z} := [\overrightarrow{t}]]$. Hence (i) is obtained from (j) by taking z_i to be $[t_i]$ for $1 \leq i \leq n$.

We show the inverse direction. Assume (i). Fix s and h such that $s, h \models \Pi \wedge \tilde{\Sigma}$. We will show $s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$. Take $\overrightarrow{a}$ for $\overrightarrow{z}$. Let s' be $s[\overrightarrow{z} := \overrightarrow{a}]$. We claim that

$s(t_j) \in \mathrm{Dom}(h)$ $(j = 1, \ldots, n)$, since each t_j appears as an address of an array atomic formula in Σ. Define h' by $\mathrm{Dom}(h') = \mathrm{Dom}(h)$ and $h'(m) = a_j$ if $m = s(t_j)$ $(j = 1, \ldots, n)$, otherwise $h'(m) = h(m)$. Then we have $s, h' \models \Pi \wedge \tilde{\Sigma}$. By (i), we obtain $s, h' \models \exists \overrightarrow{y} P'(\Pi, \Sigma, S)$. Hence $s' \models \exists \overrightarrow{y} P(\Pi, \Sigma, S)$ holds, since $s', h' \models [\overrightarrow{t}] = \overrightarrow{z}$. Therefore we have $s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$. Thus we obtain (j). □

Hence we obtained the decidability result of **SLA** stated in Theorem 3.4.

5. Separation Logic with Arrays and Lists

From this section, we will show the second result of this paper: the decidability of validity checking for QF entailments of symbolic heap system with array and list predicates. The decidability result of the previous section will be used in that of the second result.

We start from the separation logic $\mathbf{G}^+$, which is obtained from **G** by assuming the point-to predicate is ternary and adding the singly-linked list predicate ls(-,-) and the doubly-linked list predicate dll(-,-,-,-). Then we define the symbolic heap system **SLAL** which is a fragment of $\mathbf{G}^+$.

5.1. Syntax of $\mathbf{G}^+$ and SLAL.

The terms of $\mathbf{G}^+$ are same as those of **G**. The formulas (denoted by F) of $\mathbf{G}^+$ are defined as follows:

$$F ::= t = t \mid F \wedge F \mid \neg F \mid \exists x F \mid \mathrm{Emp} \mid F * F \mid t \mapsto (t, t) \mid \mathrm{Arr}(t, t) \mid \mathrm{ls}(t, t) \mid \mathrm{dll}(t, t, t, t).$$

The notations for **G** mentioned in Section 2 are also used for $\mathbf{G}^+$.

We call the singly-linked and doubly-linked list predicates *list predicates*. We also call a formula *list-free* if it does not contain any list predicates.

The list predicates are inductively defined predicates and they are introduced with the following definitions clauses:

$$\mathrm{ls}(x, y) ::= (x = y \wedge \mathrm{Emp}) \vee \exists z w (x \mapsto (z, w) * \mathrm{ls}(z, y)),$$
$$\mathrm{dll}(x_1, y_1, x_2, y_2) ::= (x_1 = y_1 \wedge x_2 = y_2 \wedge \mathrm{Emp}) \vee \exists x (x_1 \mapsto (x, y_2) * \mathrm{dll}(x, y_1, x_2, x_1)).$$

We define the k-times unfolding of the list predicates.

Definition 5.1. For $k \geq 0$, $\mathrm{ls}^k(t, u)$ and $\mathrm{dll}^k(t, u, v, w)$ are inductively defined by:

$\mathrm{ls}^0(t, u) \stackrel{\mathrm{def}}{=} 0 \neq 0 \wedge \mathrm{Emp}$,

$\mathrm{ls}^{k+1}(t, u) \stackrel{\mathrm{def}}{=} (t = u \wedge \mathrm{Emp}) \vee \exists z w (t \mapsto (z, w) * \mathrm{ls}^k(z, u))$,

$\mathrm{dll}^0(t, u, v, w) \stackrel{\mathrm{def}}{=} 0 \neq 0 \wedge \mathrm{Emp}$, and

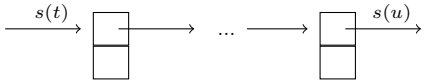
$\mathrm{dll}^{k+1}(t, u, v, w) \stackrel{\mathrm{def}}{=} (t = u \wedge v = w \wedge \mathrm{Emp}) \vee \exists z (t \mapsto (z, w) * \mathrm{dll}^k(z, u, v, t))$.

For each formula F of $\mathbf{G}^+$ and heap model (s, h), we define $s, h \models F$ extending that of **G** with

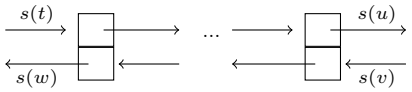
$s, h \models \mathrm{ls}(t, u) \stackrel{\mathrm{def}}{\iff} s, h \models \mathrm{ls}^k(t, u)$ for some $k \geq 0$, and

$s, h \models \mathrm{dll}(t, u, v, w) \stackrel{\mathrm{def}}{\iff} s, h \models \mathrm{dll}^k(t, u, v, w)$ for some $k \geq 0$.

We note that there is no heap model (s, h) such that $s, h \models \mathrm{ls}^0(t, u)$. Intuitively a heap model (s, h) that satisfies $s, h \models \mathrm{ls}^{k+1}(t, u)$ contains a singly-linked list of length k starting from $s(t)$ and ends at $s(u)$. This situation is depicted as follows:



We also note that there is no heap model (s, h) such that $s, h \models \mathrm{dll}^0(t, u, v, w)$. A heap model (s, h) that satisfies $s, h \models \mathrm{dll}^{k+1}(t, u, v, w)$ contains a doubly-linked list of length k with a forward-link starting from $s(t)$ and ending at $s(u)$, and a backward-link starting from $s(v)$ and ending at $s(w)$. This situation is depicted as follows:



The notation $F \models \overrightarrow{F}$ is also defined in a similar way to **SLA**.

Formulas of **SLAL** are QF symbolic heaps (denoted by φ) of the form $\Pi \wedge \Sigma$, where its pure part Π is the same as that of **SLA** and its spatial part Σ is defined by

$$\Sigma ::= \mathrm{Emp} \mid t \mapsto (t, t) \mid \mathrm{Arr}(t, t) \mid \mathrm{ls}(t, t) \mid \mathrm{dll}(t, t, t, t) \mid \Sigma * \Sigma.$$

We use notations Π_φ and Σ_φ that mean the pure part and the spatial part of φ, respectively.

Entailments of **SLAL** are QF entailments of the form $\varphi \vdash \{\varphi_i \mid i \in I\}$. The validity of an entailment is defined in a similar way to **SLA**.

6. Unroll Collapse

This section shows the unroll collapse properties for the singly-linked and doubly-linked list predicates. In the decision procedure for **SLAL**, these properties will be used for eliminating the list predicates in the antecedent of a given entailment.

The unroll collapse property for the singly-linked list predicate is given in the next proposition. The key idea of its proof is to replace the list $\mathrm{ls}(t, u)$ by some list $\mathrm{ls}(t, u)$ of length 2 in the antecedent of (1), in order to show (1). Then (2) applies to obtain the succedent. Then we can show that the list of length 2 is contained in some list $\mathrm{ls}(p, q)$ of the succedent. By replacing the list of length 2 by $\mathrm{ls}(t, u)$ of arbitrary length, we obtain the succedent of (1), since the difference by the replacement is absorbed by $\mathrm{ls}(p, q)$.

Proposition 6.1 (Unroll Collapse for ls)**.** *The following clauses are equivalent:*

(1) $\mathrm{ls}(t, u) * \phi \models \overrightarrow{\psi}$,

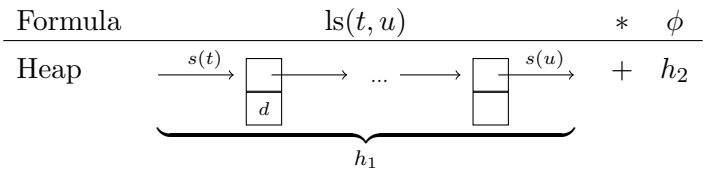
(2) $t = u \wedge \phi \models \overrightarrow{\psi}$ *and* $t \mapsto (z, y_1) * z \mapsto (u, y_2) * \phi \models \overrightarrow{\psi}$, *where* z, y_1, y_2 *are fresh.*

Proof of Proposition 6.1. From (1) to (2) is trivial. We consider the inverse direction. Assume $s, h \models \mathrm{ls}(t, u) * \phi$. We will show $s, h \models \overrightarrow{\psi}$.

By the assumption, there are h_1 and h_2 such that $s, h_1 \models \mathrm{ls}(t, u)$, $s, h_2 \models \phi$, and $h = h_1 + h_2$. Hence $s, h_1 \models \mathrm{ls}^n(t, u)$ for some n (take the smallest one).

We will show $s, h \models \overrightarrow{\psi}$. In the case of $n = 0$, it is not the case since $\mathrm{ls}^0(t, u)$ is unsatisfiable. In the case of $n = 1$ or $n = 3$, we have $s, h \models \overrightarrow{\psi}$ by (2).

We consider the other cases. Let d be the second component of $h_1(s(t))$. Then the current situation of h_1 and h_2 is depicted as follows:

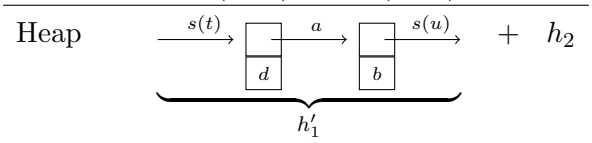


Fix values $a, b \in N$ such that $a \notin \mathrm{Dom}(h) \cup \{s(p) \mid p \mapsto (\text{-}, \text{-}) \in \overrightarrow{\psi}\} \cup \{s(p), s(q) \mid \mathrm{ls}(p, q) \in \overrightarrow{\psi}\} \cup \{s(p), s(q), s(p'), s(q') \mid \mathrm{dll}(p, q, p', q') \text{ is in } \overrightarrow{\psi}\} \cup \bigcup\{[s(p), s(q)] \mid \mathrm{Arr}(p, q) \text{ is in } \overrightarrow{\psi}\}$, and $b \notin \mathrm{Dom}(h) \cup \{a\}$.

Define h_1' by $\mathrm{Dom}(h_1') = \{s(t), a\}$, $h_1'(m) = (a, d)$ if $m = s(t)$, and $h_1'(m) = (s(u), b)$ if $m \neq s(t)$. Let $s' = s[z := a, y_1 := d, y_2 := b]$. Then we have

$$s', h_1' + h_2 \models t \mapsto (z, y_1) * z \mapsto (u, y_2) * \phi,$$

which is depicted as follows:



Hence, by the assumption, we have

$$s', h_1' + h_2 \models \psi_j$$

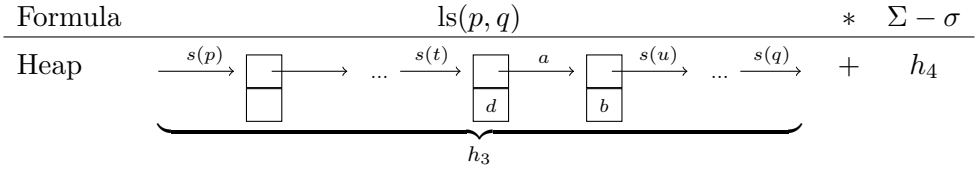
for some $\psi_j \in \overrightarrow{\psi}$. Recall that z, y_1, y_2 do not appear in ψ_j since they are fresh variables. So we have

$$s, h_1' + h_2 \models \psi_j.$$

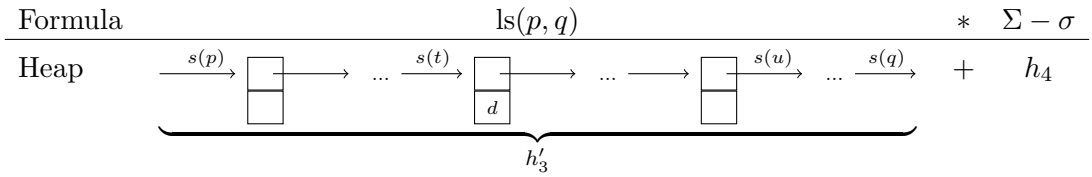
Let $\psi_j \equiv \Pi \wedge \Sigma$. Then $s \models \Pi$, and there are h_3, h_4 and an atomic spatial formula σ such that $s, h_3 \models \sigma$, $s, h_4 \models \Sigma - \sigma$, $a \in \mathrm{Dom}(h_3)$, and $h_3 + h_4 = h_1' + h_2$, where $\Sigma - \sigma$ is the spatial formula obtained by removing σ from Σ.

We can show that σ must have the form $\mathrm{ls}(p, q)$ as follows: it cannot be $p \mapsto (\text{-}, \text{-})$ by $a \neq s(p)$; it cannot be $\mathrm{Arr}(p, q)$ by $a \notin [s(p), s(q)]$; it cannot be $\mathrm{dll}(p, q, p', q')$, otherwise, by $a \neq s(p), s(p'), b$ and $h_3(a) = (s(u), b)$, we have $b \in \mathrm{Dom}(h_3) \setminus \{a\} \subseteq \mathrm{Dom}(h_3 + h_4) \setminus \{a\} = \mathrm{Dom}(h_1' + h_2) \setminus \{a\} \subseteq \mathrm{Dom}(h)$, which contradicts the condition of b.

Note that $a \neq s(p), s(q)$ by the condition of a. Hence we have $s(t), a \in \mathrm{Dom}(h_3)$, $h_3(s(t)) = h_1'(s(t)) = (a, d)$ and $h_3(a) = h_1'(a) = (s(u), b)$. Therefore the current situation of h_3 and h_4 is depicted as follows:



Then define h_3' by $\mathrm{Dom}(h_3') = \mathrm{Dom}(h_1) + (\mathrm{Dom}(h_3) \setminus \{a, s(t)\})$, where the symbol $+$ is the disjoint union symbol, $h_3'(m) = h_1(m)$ if $m \in \mathrm{Dom}(h_1)$, and $h_3'(m) = h_3(m)$ if $m \notin \mathrm{Dom}(h_1)$. Then we have the following situation:



Note that the above picture covers both cases of $n = 2$ and $n \geq 4$. It satisfies $s, h'_3 \models \mathrm{ls}(p, q)$. We have $h'_3 + h_4 = h_1 + h_2 = h$ by removing h'_1 from both sides of $h_3 + h_4 = h'_1 + h_2$ and adding h_1 to them. Hence we have $s, h'_3 + h_4 \models \Sigma$. Therefore we have (1) since $s, h \models \overrightarrow{\psi}$ can be obtained. □

The unroll collapse property for the doubly-linked list predicate is given in the next proposition. The key idea of its proof is similar to that of Proposition 6.1: First the list $\mathrm{dll}(t, u, v, w)$ is replaced by some doubly-linked list $\mathrm{dll}(t, u, v, w)$ of length 3 in the antecedent of (1), in order to show (1). Then (2) applies to obtain the succedent. Then we can show that the doubly-linked list of length 3 is contained in some list either $\mathrm{ls}(p, q)$ or $\mathrm{dll}(p, q, p', q')$ of the succedent. By replacing the doubly-linked list of length 3 by $\mathrm{dll}(t, u, v, w)$ of arbitrary length, we obtain the succedent of (1), since the difference by the replacement is absorbed by $\mathrm{ls}(p, q)$ or $\mathrm{dll}(p, q, p', q')$.

Proposition 6.2 (Unroll Collapse for dll)**.** *The following clauses are equivalent:*

(1) $\mathrm{dll}(t, u, v, w) * \phi \models \overrightarrow{\psi}$.

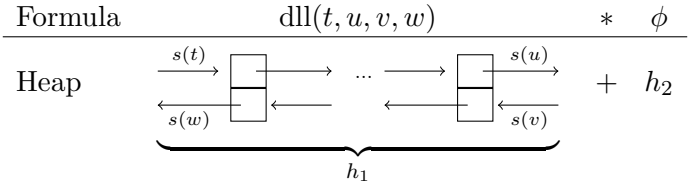
(2) $t = u \wedge v = w \wedge \phi \models \overrightarrow{\psi}$, $\quad t = v \wedge t \mapsto (u, w) * \phi \models \overrightarrow{\psi}$, *and*
$t \mapsto (z, w) * z \mapsto (v, t) * v \mapsto (u, z) * \phi \models \overrightarrow{\psi}$, *where* z *is fresh.*

Proof of Proposition 6.2. From (1) to (2) is trivial. We consider the inverse direction. Assume $s, h \models \mathrm{dll}(t, u, v, w) * \phi$. We will show $s, h \models \overrightarrow{\psi}$.

By the assumption, there are h_1 and h_2 such that $s, h_1 \models \mathrm{dll}(t, u, v, w)$, $s, h_2 \models \phi$, and $h = h_1 + h_2$. Hence $s, h_1 \models \mathrm{dll}^n(t, u, v, w)$ for some n (take the smallest one).

We will show $s, h \models \overrightarrow{\psi}$. The case of $n = 0$ does not happen since $\mathrm{dll}^0(t, u, v, w)$ is unsatisfiable. In the case of $n = 1$, $n = 2$, or $n = 4$, we have $s, h \models \overrightarrow{\psi}$ by (2).

We consider the other cases. The current situation of h_1 and h_2 is depicted as follows:



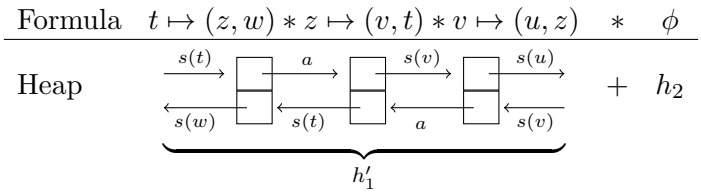
Fix a value $a \in N$ such that $a \notin \mathrm{Dom}(h) \cup \{s(p), s(q), s(r) \mid p \mapsto (q, r) \text{ is in } \overrightarrow{\psi}\} \cup \{s(p), s(q) \mid \mathrm{ls}(p, q) \text{ is in } \overrightarrow{\psi}\} \cup \{s(p), s(q), s(p'), s(q') \mid \mathrm{dll}(p, q, p', q') \text{ is in } \overrightarrow{\psi}\} \cup \bigcup \{[s(p), s(q)] \mid \mathrm{Arr}(p, q) \text{ is in } \overrightarrow{\psi}\}$.

Define h'_1 by $\mathrm{Dom}(h'_1) = \{s(t), a, s(v)\}$, $h'_1(m) = (a, s(w))$ if $m = s(t)$, $h'_1(m) = (s(v), s(t))$ if $m = a$, and $h'_1(m) = (s(u), a)$ if $m = s(v)$.

Then we have

$$s[z := a], h'_1 + h_2 \models t \mapsto (z, w) * z \mapsto (v, t) * v \mapsto (u, z) * \phi.$$

The current situation is as follows:



Then, by the assumption, we have

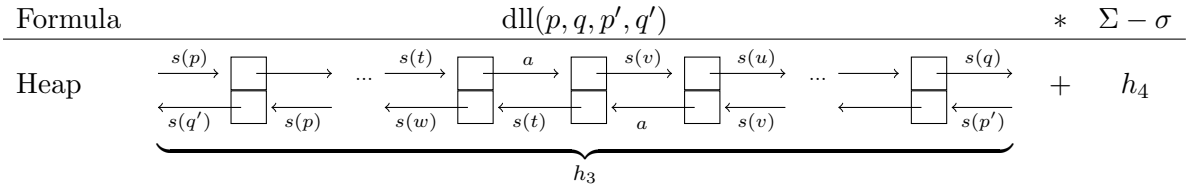
$$s, h'_1 + h_2 \models \psi_j$$

for some $\psi_j \in \overrightarrow{\psi}$, since z does not appear in ψ_j.

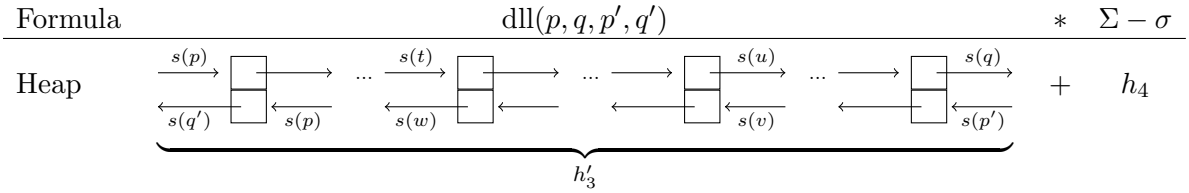
Let $\psi_j \equiv \Pi \wedge \Sigma$. Then $s \models \Pi$, and there are h_3, h_4 and σ such that $s, h_3 \models \sigma$, $s, h_4 \models \Sigma - \sigma$, $a \in \mathrm{Dom}(h_3)$, and $h_3 + h_4 = h'_1 + h_2$.

We can show σ have the form $\mathrm{ls}(p, q)$ or $\mathrm{dll}(p, q, p', q')$ as follows: it cannot be $p \mapsto$ (-,-) by $a \neq s(p)$; it cannot be $\mathrm{Arr}(p, q)$ by $a \notin [s(p), s(q)]$.

Case 1: the case of $\sigma \equiv \mathrm{dll}(p, q, p', q')$. Note that $s(v), s(t), a \in \mathrm{Dom}(h_3)$, since a cannot be the first or last cell of the dll by $a \neq s(p), s(p')$. Hence $h_3(s(t)) = h'_1(s(t)) = (a, s(w))$ and $h_3(a) = h'_1(a) = (s(v), s(t))$. This case is depicted as follows:



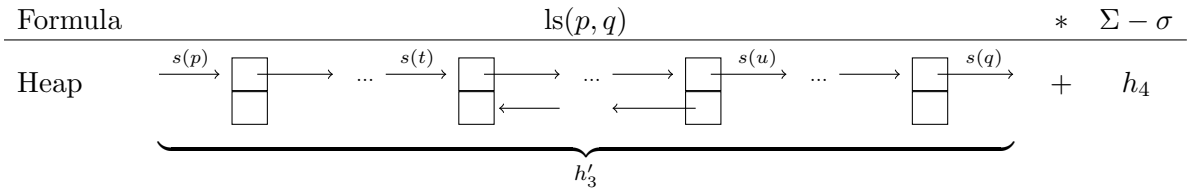
Define h'_3 by $\mathrm{Dom}(h'_3) = \mathrm{Dom}(h_1) + (\mathrm{Dom}(h_3) \setminus \{a, s(t), s(v)\})$, $h'_3(m) = h_1(m)$ if $m \in \mathrm{Dom}(h_1)$, and $h'_3(m) = h_3(m)$ if $m \notin \mathrm{Dom}(h_1)$. Then we have the situation depicted as follows:



Note that the above picture covers both cases of $n = 3$ and $n \geq 5$. It satisfies $s, h'_3 \models \mathrm{dll}(p, q, p', q')$. We have $h'_3 + h_4 = h_1 + h_2 = h$ by removing h'_1 from both sides of $h_3 + h_4 = h'_1 + h_2$ and adding h_1 to them. Hence we have $s, h'_3 + h_4 \models \Sigma$. Therefore we have (1) since $s, h \models \overrightarrow{\psi}$ can be obtained.

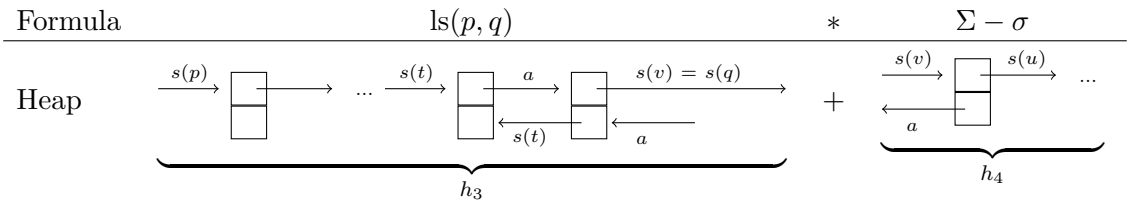
Case 2: the case of $\sigma \equiv \mathrm{ls}(p, q)$. Note that $a, s(t) \in \mathrm{Dom}(h_3)$, since a cannot be the first cell of the list by $a \neq s(p)$. Hence $h_3(s(t)) = h'_1(s(t)) = (a, s(w))$ and $h_3(a) = h'_1(a) = (s(v), s(t))$. We also note that $s(v) \in \mathrm{Dom}(h_3)$ or $s(v) \in \mathrm{Dom}(h_4)$. The latter case implies $s(v) = s(q)$. We consider two subcases about where $s(v)$ is.

Case 2.1: the case of $s(v) \in \mathrm{Dom}(h_3)$. Consider h'_3 taken in the case 1. This case is depicted as follows:

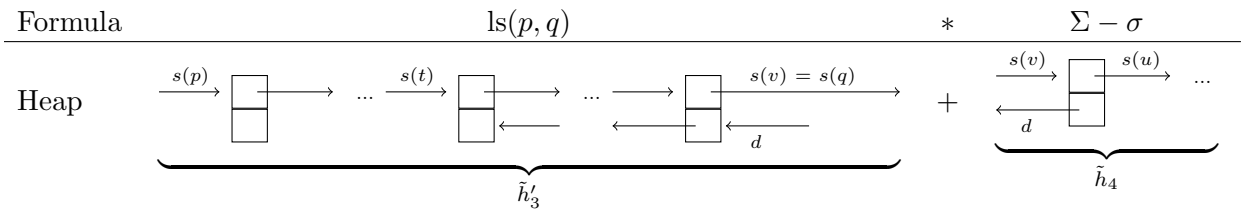


Note that the above picture covers both cases of $n = 3$ and $n \geq 5$. It satisfies $s, h'_3 \models \mathrm{ls}(p, q)$. We have $h'_3 + h_4 = h_1 + h_2 = h$ by removing h'_1 from both sides of $h_3 + h_4 = h'_1 + h_2$ and adding h_1 to them. Hence we have $s, h'_3 + h_4 \models \Sigma$. Therefore we have (1) since $s, h \models \overrightarrow{\psi}$ can be obtained.

Case 2.2: the case of $s(v) \in \mathrm{Dom}(h_4)$. Recall that this case implies $s(v) = s(q)$. This case is depicted as follows:



Let $h_1(s(v)) = (s(u), d)$. Then we define $\tilde{h}'_3$ by $\mathrm{Dom}(\tilde{h}'_3) = (\mathrm{Dom}(h_1) \setminus \{s(v)\}) + (\mathrm{Dom}(h_3) \setminus \{a, s(t)\})$, $\tilde{h}'_3(m) = h_1(m)$ if $m \in \mathrm{Dom}(h_1) \setminus \{s(v)\}$, and $\tilde{h}'_3(m) = h_3(m)$ otherwise. We also define $\tilde{h}_4$ by $\mathrm{Dom}(\tilde{h}_4) = \mathrm{Dom}(h_4)$, $\tilde{h}_4(m) = h_4(m)$ if $m \neq s(v)$, and $\tilde{h}_4(m) = (s(u), d)$ if $m = s(v)$. Then we have the following situation:



Note that the above picture covers both cases of $n = 3$ and $n \geq 5$. Then we have $\tilde{h}'_3 + \tilde{h}_4 = h_1 + h_2 = h$, by removing h'_1 from both sides of $h_3 + h_4 = h'_1 + h_2$ and adding h_1 to them. Hence we have $s, \tilde{h}'_3 + \tilde{h}_4 \models \Sigma$. Therefore we have (1) since $s, h \models \overrightarrow{\psi}$ can be obtained. □

Remark 6.3. We note that our unroll collapse properties (Proposition 6.1 and 6.2) hold for entailments with the points-to predicate, the array predicate, and the (possibly cyclic) singly-linked and doubly-linked list predicates. We also note that ours hold for entailments that contain arithmetic. The original version of unroll collapse is given by Berdine et al. [2]. It holds for entailments with only the points-to predicate and the acyclic singly-linked list predicate. We cannot compare ours and theirs naively, since the singly-linked list predicates in both papers are different for cyclic lists.

7. Decision Procedure for Arrays and Lists

This section gives our algorithm for checking the validity of a given entailment, whose antecedent do not contain list predicates. In the following subsections 7.1 and 7.2, we implicitly assume that the antecedent of each entailment is list-free.

The procedure first eliminates the list predicates in the succedent of a given entailment. The resulting entailments only contain the points-to and array predicates, that is, they are entailments in **SLA**. Then the procedure checks their validity by using the decision procedure of **SLA**.

7.1. Proof System for Elimination of Lists in Succedents.

This subsection gives a proof system for entailments whose antecedents do not contain list predicates. Our decision procedure is given as a proof-search procedure of the proof system.

We define a Presburger formula $\Sigma \to\!\downarrow t$ that means the address t is a cell of Σ. We also define $\mathrm{Tm}(\Sigma)$, which is the set of terms in Σ. For defining them, we will not implicitly use the commutative law for $*$ and the unit law for Emp.

Definition 7.1. $\Sigma \to\!\downarrow t$ is inductively defined as follows:

$$\begin{array}{ll} \mathrm{Emp} \to\!\downarrow t \stackrel{\mathrm{def}}{=} \mathrm{False}, & \mathrm{Emp} * \Sigma' \to\!\downarrow t \stackrel{\mathrm{def}}{=} \Sigma' \to\!\downarrow t, \\ t' \mapsto (\text{-}, \text{-}) \to\!\downarrow t \stackrel{\mathrm{def}}{=} t = t', & t' \mapsto (\text{-}, \text{-}) * \Sigma' \to\!\downarrow t \stackrel{\mathrm{def}}{=} t = t' \vee \Sigma' \to\!\downarrow t, \\ \mathrm{Arr}(t', u') \to\!\downarrow t \stackrel{\mathrm{def}}{=} t' \le t \le u', & \mathrm{Arr}(t', u') * \Sigma \to\!\downarrow t \stackrel{\mathrm{def}}{=} t' \le t \le u' \vee \Sigma \to\!\downarrow t. \end{array}$$

Definition 7.2. $\mathrm{Tm}(\Sigma)$ is inductively defined as follows:

$$\mathrm{Tm}(\mathrm{Emp}) \stackrel{\mathrm{def}}{=} \emptyset, \quad \mathrm{Tm}(t \mapsto (u_1, u_2)) \stackrel{\mathrm{def}}{=} \{t, u_1, u_2\}, \quad \mathrm{Tm}(\mathrm{Arr}(t, u)) \stackrel{\mathrm{def}}{=} \{t, u\},$$
$$\mathrm{Tm}(\Sigma_1 * \Sigma_2) \stackrel{\mathrm{def}}{=} \mathrm{Tm}(\Sigma_1) \cup \mathrm{Tm}(\Sigma_2).$$

We write $\mathrm{Tm}(\overrightarrow{\Sigma})$ for $\bigcup_{\Sigma \in \overrightarrow{\Sigma}} \mathrm{Tm}(\Sigma)$.

Lemma 7.3. Suppose $s, h \models \Sigma$. Then $s \models \Sigma \to\!\downarrow t$ if and only if $s(t) \in \mathrm{Dom}(h)$.

Proof. The claim is shown by induction on Σ. □

Then we define the inference rules which give our algorithm. The rules are shown in Figure 2.

Let h and h' be heaps such that $\mathrm{Dom}(h) = \mathrm{Dom}(h')$. We write $h \sim_d h'$ if $h(x) = h'(x)$ holds for any $x \in \mathrm{Dom}(h) \setminus \{d\}$.

The following lemma is used in the proof of local completeness of the inference rules.

Lemma 7.4. *(1) Suppose that $s, h \models \sigma$, $(a, b) \in \mathrm{Ran}(h)$ and $a \notin \mathrm{Dom}(h) \cup \{s(t) \mid t \in \mathrm{Tm}(\sigma)\}$. Then σ is an array atomic formula.*

(2) Suppose that $s, h \models \Sigma$, $h(d) = (a, b)$, $a \notin \mathrm{Dom}(h) \cup \{s(t) \mid t \in \mathrm{Tm}(\Sigma)\}$ and $h' \sim_d h$. Then $s, h' \models \Sigma$.

Proof. (1) We show the claim by case analysis of σ.

The case that σ is $t \mapsto (u, v)$. By the assumptions, we have $(a, b) \in \mathrm{Ran}(h) = \{(s(u), s(v))\}$. Hence we obtain $a = s(u) \in \{s(t) \mid t \in \mathrm{Tm}(\sigma)\}$, which contradicts the assumption.

The case that σ is $\mathrm{ls}(t, u)$. By the assumptions, we have $h \neq \emptyset$. Hence h contains a non-empty list that starts from t. Then a must be a cell of the list, since $a \neq s(u)$. Therefore we have $a \in \mathrm{Dom}(h)$, which contradicts the assumption.

$$\frac{}{\varphi \vdash \overrightarrow{\psi}}\ (\text{Start}) \quad \text{where } \varphi \text{ is list-free and } \varphi \models \overrightarrow{\psi} \text{ in } \mathbf{SLA}$$

$$\frac{}{\varphi \vdash \overrightarrow{\psi}}\ (\text{UnsatL}) \quad \text{if } \varphi \text{ is unsat} \qquad \frac{\varphi \vdash \overrightarrow{\psi}}{\varphi \vdash \psi, \overrightarrow{\psi}}\ (\text{UnsatR}) \quad \text{if } \varphi \wedge \psi \text{ is unsat}$$

$$\frac{\begin{array}{l} t \neq t' \wedge t \mapsto (v, w) * \varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi} \\ t = t' \wedge t \mapsto (v, w) * \varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi} \end{array}}{t \mapsto (v, w) * \varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}}\ (\mapsto\text{LsEM})$$
$$\text{if } t = t' \wedge \Pi_\varphi \text{ and } t \neq t' \wedge \Pi_\varphi \text{ are satisfiable}$$

$$\frac{t \mapsto (v, w) * \varphi \vdash t' = u' \wedge \psi, t' \mapsto (v, w) * \mathrm{ls}(v, u') * \psi, \overrightarrow{\psi}}{t \mapsto (v, w) * \varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}}\ (\mapsto\text{Ls}) \quad \text{if } \Pi_\varphi \models t = t'$$

$$\frac{\varphi \vdash t' = u' \wedge \psi, \overrightarrow{\psi}}{\varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}}\ (\text{LsElim}) \quad \text{if } \Pi_\varphi \not\models \Sigma_\varphi \rightarrow\!\downarrow t'$$

$$\frac{\begin{array}{l} t \neq t' \wedge t \mapsto (v, w) * \varphi \vdash \mathrm{dll}(t', u', v', w') * \psi, \overrightarrow{\psi} \\ t = t' \wedge t \mapsto (v, w) * \varphi \vdash \mathrm{dll}(t', u', v', w') * \psi, \overrightarrow{\psi} \end{array}}{t \mapsto (v, w) * \varphi \vdash \mathrm{dll}(t', u', v', w') * \psi, \overrightarrow{\psi}}\ (\mapsto\text{DllEM})$$
$$\text{if } t = t' \wedge \Pi_\varphi \text{ and } t \neq t' \wedge \Pi_\varphi \text{ are satisfiable}$$

$$\frac{t \mapsto (v, w) * \varphi \vdash t' = u' \wedge v' = w' \wedge \psi, t \mapsto (v, w') * \mathrm{dll}(v, u', v', t') * \psi, \overrightarrow{\psi}}{t \mapsto (v, w) * \varphi \vdash \mathrm{dll}(t', u', v', w') * \psi, \overrightarrow{\psi}}\ (\mapsto\text{Dll}) \quad \text{if } \Pi_\varphi \models t = t'$$

$$\frac{\varphi \vdash t' = u' \wedge v' = w' \wedge \psi, \overrightarrow{\psi}}{\varphi \vdash \mathrm{dll}(t', u', v', w') * \psi, \overrightarrow{\psi}}\ (\text{DllElim}) \quad \text{if } \Pi_\varphi \not\models \Sigma_\varphi \rightarrow\!\downarrow t'$$

$$\frac{\begin{array}{l} t' < t \wedge \mathrm{Arr}(t, v) * \varphi \vdash L(t', \overrightarrow{u'}) * \psi, \overrightarrow{\psi} \\ v < t' \wedge \mathrm{Arr}(t, v) * \varphi \vdash L(t', \overrightarrow{u'}) * \psi, \overrightarrow{\psi} \\ t \leq t' \leq v \wedge \mathrm{Arr}(t, v) * \varphi \vdash L(t', \overrightarrow{u'}) * \psi, \overrightarrow{\psi} \end{array}}{\mathrm{Arr}(t, v) * \varphi \vdash L(t', \overrightarrow{u'}) * \psi, \overrightarrow{\psi}}\ (\text{ArrListEM}),$$
$$\text{where } t \leq t' \leq v \wedge \Pi_\varphi \text{ and } (t' < t \vee v < t') \wedge \Pi_\varphi \text{ are satisfiable}$$
$$L(t', \overrightarrow{u'}) \text{ is } \mathrm{ls}(t', u') \text{ or } \mathrm{dll}(t', u'_1, u'_2, u'_3)$$

$$\frac{\mathrm{Arr}(t, v) * \varphi \vdash t' = u' \wedge \psi, \overrightarrow{\psi}}{\mathrm{Arr}(t, v) * \varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}}\ (\text{ArrLs}) \quad \text{if } \Pi_\varphi \models t \leq t' \leq v$$

$$\frac{\mathrm{Arr}(t, v) * \varphi \vdash t' = u' \wedge v' = w' \wedge \psi, \overrightarrow{\psi}}{\mathrm{Arr}(t, v) * \varphi \vdash \mathrm{dll}(t', u', v', w') * \psi, \overrightarrow{\psi}}\ (\text{ArrDll}) \quad \text{if } \Pi_\varphi \models t \leq t' \leq v$$

Figure 2: Inference rules for the decision procedure

The case of $\mathrm{dll}(t, u, v, w)$ can be shown in a similar way to the case of $\mathrm{ls}(t, u)$.

(2) By the assumptions, there exist h_1 and h_2 such that $s, h_1 \models \sigma$, $s, h_2 \models \Sigma - \sigma$, $h = h_1 + h_2$ and $d \in \mathrm{Dom}(h_1)$. By (1), σ is an array atomic formula. Let h'_1 be $h'|_{\mathrm{Dom}(h_1)}$. Then we have $s, h'_1 \models \sigma$, since $h_1 \sim_d h'_1$. Thus we obtain $s, h' \models \Sigma$, since $h' = h'_1 + h_2$. □

Proposition 7.5. Each inference rule is sound and locally complete.

Proof. The claims of the rules (Start), (UnsatL) and (UnsatR) are immediately shown. The claims of the rules ($\mapsto$LsEM), ($\mapsto$DllEM), (ArrListEM), ($\mapsto$Ls) and ($\mapsto$Dll) are easily shown (without the side-conditions).

Soundness of the rule (LsElim) is easily shown. We show local completeness of it. Assume $\varphi \models \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}$, the validity $s, h \models \varphi$ and $\Pi_\varphi \not\models \Sigma_\varphi \to\!\!\downarrow t'$. We will show $s, h \models t' = u' \wedge \psi, \overrightarrow{\psi}$. By the assumption, we have $s, h \models \mathrm{ls}(t', u') * \psi$ or $s, h \models \overrightarrow{\psi}$. If the latter case holds, then we have the claim. Otherwise there exist h_1 and h_2 such that $h = h_1 + h_2$, $s, h_1 \models \mathrm{ls}(t', u')$ and $s, h_2 \models \psi$. By the lemma 7.3, we have $s(t') \notin \mathrm{Dom}(h)$ since $s, h \models \Sigma_\varphi$ and $s \not\models \Sigma_\varphi \to\!\!\downarrow t'$. Hence we obtain $h_1 = \emptyset$, $s \models t' = u'$ and $s, h \models \psi$. Thus $s, h \models t' = u' \wedge \psi$ holds.

Soundness of the rule (DllElim) is shown immediately. Local completeness of it can be shown in a similar way to the proof of local completeness of (LsElim).

Soundness of the rule (ArrLs) is easily shown. We show local completeness of it. Assume $\mathrm{Arr}(t, v) * \varphi \models \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}$, the validity $s, h \models \mathrm{Arr}(t, v) * \varphi$ and $\Pi_\varphi \models t \leq t' \leq v$. We will show $s, h \models t' = u' \wedge \psi, \overrightarrow{\psi}$. Let (a, b) be $h(s(t'))$. Fix a fresh value a' such that $a' \notin \mathrm{Dom}(h) \cup \{s(t) \mid t \in \mathrm{Tm}(\mathrm{ls}(t', u') * \Sigma_\psi, \Sigma_{\overrightarrow{\psi}})\}$. Define h' by $h' \sim_{s(t')} h$ and $h'(s(t')) = (a', b)$. Then we have $s, h' \models \mathrm{Arr}(t, v) * \varphi$ since $s \models t \leq t' \leq v$ by the side condition. Hence $s, h' \models \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}$ holds. If $s, h' \models \overrightarrow{\psi}$, then we have the claim by Lemma 7.4 (2). Otherwise we have $s, h' \models \mathrm{ls}(t', u') * \psi$. Hence there exist h'_1 and h'_2 such that $h' = h'_1 + h'_2$, $s, h'_1 \models \mathrm{ls}(t', u')$ and $s, h'_2 \models \psi$. Then we can show $h'_1 = \emptyset$ as follows: Suppose it does not hold, then $s(t') \in \mathrm{Dom}(h'_1)$ and $(a', b) \in \mathrm{Ran}(h'_1)$; hence we have a contradiction since $\mathrm{ls}(t', u')$ is an array atomic formula by Lemma 7.4 (1). Therefore we have $s \models t' = u'$ and $s, h' \models \psi$. Finally we have $s, h \models t' = u' \wedge \psi$ by Lemma 7.4 (2).

The claim of the rule (ArrDll) can be shown in a similar way to (ArrLs). □

7.2. **Proof Search Algorithm.** In our decision procedure, we read each inference rule from the bottom (conclusion) to the top (assumptions). Let $\mathcal{E}$ be the set of entailments whose antecedents do not contain list predicates. For each inference rule $\mathcal{R}$ we define a partial function $\mathrm{Apply}_\mathcal{R}$ from $\mathcal{E}$ to $\mathcal{P}(\mathcal{E})$ as follows. $\mathrm{Apply}_\mathcal{R}(J)$ is defined when J is a conclusion of some instance of $\mathcal{R}$ (including its side condition). $\mathrm{Apply}_\mathcal{R}(J)$ is the assumptions in some instance of $\mathcal{R}$ with the conclusion J (non-deterministically chosen).

Let τ be a derivation tree $\dfrac{\tau_1 \quad \cdots \quad \tau_k}{J}\,\mathcal{R}$. We sometimes represent this tree by a tuple $(\mathcal{R}, J, \tau'_1, \ldots, \tau'_k)$, where $\tau'_1, \ldots, \tau'_k$ are representation of $\tau_1, \ldots, \tau_k$, respectively. Our proof search procedure `search` is given in Figure 3. `search`(J) returns a tuple for a derivation tree of J or returns `fail`.

We first show the termination property of `search`. In order to show this, we define some notations.

Definition 7.6. (1) $\sharp_\mapsto(\psi)$ is the number of $\mapsto$ in ψ, $\sharp_{\mathrm{lists}}(\psi)$ is the number of ls and dll in ψ.

(2) $|\psi|^{\mathrm{Unfold}}_\varphi$ is defined by $\sharp_\mapsto(\varphi) - \sharp_\mapsto(\psi)$.

(3) Let $L(t', \overrightarrow{u'})$ be $\mathrm{ls}(t', \overrightarrow{u'})$ or $\mathrm{dll}(t', \overrightarrow{u'})$, where $\overrightarrow{u'}$ has an appropriate length. $\deg(\Pi, \sigma, \sigma')$ is defined as follows:

$$\deg(\Pi, t \mapsto (\text{-}, \text{-}), L(t', \text{-})) = \begin{cases} 1 & \text{if } t = t' \wedge \Pi \text{ and } t \neq t' \wedge \Pi \text{ are satisfiable,} \\ 0 & \text{otherwise.} \end{cases}$$

function search(J)
  if (UnsatL is applicable to J) $\mathcal{R}$:= UnsatL
    else if (Start is applicable to J) $\mathcal{R}$:= Start
      else if (UnsatR is applicable to J) $\mathcal{R}$:= UnsatR
        else if ($\mapsto$LsEM is applicable to J) $\mathcal{R}$:= $\mapsto$ LsEM
          else if (ArrListEM is applicable to J) $\mathcal{R}$:= ArrListEM
            else if (LsElim is applicable to J) $\mathcal{R}$:= LsElim
              else if (ArrLs is applicable to J) $\mathcal{R}$:= ArrLs
                else if ($\mapsto$Ls is applicable to J) $\mathcal{R}$:= $\mapsto$ Ls
                  else if ($\mapsto$DllEm is applicable to J) $\mathcal{R}$:= $\mapsto$ DllEM
                    else if (DllElim is applicable to J) $\mathcal{R}$:= DllElim
                      else if (ArrDll is applicable to J) $\mathcal{R}$:= ArrDll
                        else if ($\mapsto$Dll is applicable to J) $\mathcal{R}$:= $\mapsto$ Dll
                          else return(fail)
  $\{J_1, \ldots, J_k\}$:= Apply$_{\mathcal{R}}(J)$
  if (each of search(J_i) returns a tuple) **return**$((\mathcal{R}, J, \mathtt{search}(J_1), \ldots, \mathtt{search}(J_k)))$
  else return(fail)

Figure 3: Proof search algorithm

$$\deg(\Pi, \mathrm{Arr}(t,u), L(t', \text{-})) = \begin{cases} 1 \text{ if } t \le t' \le u \wedge \Pi \text{ and } (t' < t \vee u < t') \wedge \Pi \text{ are satisfiable,} \\ 0 \text{ otherwise.} \end{cases}$$

$\deg(\Pi, \sigma, \sigma') = 0$ if σ' is not a list predicate.

Then we define $|\psi|_{\varphi}^{\mathrm{EM}}$ by $\sum_{\sigma \text{ in } \varphi, \sigma' \text{ in } \psi} \deg(\Pi_{\varphi}, \sigma, \sigma')$.

(4) We define the degree $||\psi||_{\varphi}$ of ψ with respect to φ by $(\sharp_{\mathrm{lists}}(\psi), |\psi|_{\varphi}^{\mathrm{Unfold}}, |\psi|_{\varphi}^{\mathrm{EM}})$. The order on degrees is given by the lexicographic order.

(5) Let J be an entailment $\varphi \vdash \{\psi_i\}_{i \in I}$. We define the measure $\widetilde{J}$ of J as the sequence of $||\psi_i||_{\varphi}$ $(i \in I)$ sorted in decreasing order. Then we write $J_1 < J_2$ for $\widetilde{J_1} <_{\mathrm{lex}} \widetilde{J_2}$.

$|\psi|_{\varphi}^{\mathrm{Unfold}}$ gives an upper bound of unfolding of list predicates in ψ under φ. During the proof search, a list predicate in ψ is unfolded if a matched points-to atomic formula in φ is found. So, if unfolding is done more than the upper bound, ψ becomes unsatisfiable since some points-to atomic formula has to match more than once.

Note that the relation $<$ on entailments is a well-founded preorder, that is, there is no infinite decreasing chain, since the lexicographic order $\le_{\mathrm{lex}}$ on measures is a well-order.

Lemma 7.7. *The degrees and the preorder on entailments satisfy the following properties.*

(1) $\overrightarrow{\psi_1} \subsetneq \overrightarrow{\psi_2}$ *implies* $(\varphi \vdash \overrightarrow{\psi_1}) < (\varphi \vdash \overrightarrow{\psi_2})$.

(2) $||\psi_1||_{\varphi}, \ldots, ||\psi_k||_{\varphi} < ||\psi||_{\varphi}$ *implies* $(\varphi \vdash \overrightarrow{\psi}, \psi_1, \ldots, \psi_k) < (\varphi \vdash \overrightarrow{\psi}, \psi)$.

(3) If $\Pi \subseteq \Pi'$, *then* $||\psi||_{\Pi' \wedge \varphi} \le ||\psi||_{\Pi \wedge \varphi}$.

(4) Suppose that $t = t' \wedge \Pi_{\varphi}$ *and* $t \ne t' \wedge \Pi_{\varphi}$ *are satisfiable,* $\varphi = t \mapsto (\text{-}, \text{-}) * \varphi'$ *and* $\psi = L(t', \text{-}) * \psi'$. *Then* $(t = u \wedge \varphi \vdash \overrightarrow{\psi}, \psi), (t \ne u \wedge \varphi \vdash \overrightarrow{\psi}, \psi) < (\varphi \vdash \overrightarrow{\psi}, \psi)$.

(5) Suppose that $t \le t' \le u \wedge \Pi_{\varphi}$ *and* $(t' < t \vee u < t') \wedge \Pi_{\varphi}$ *are satisfiable,* $\varphi = \mathrm{Arr}(t, u) * \varphi'$ *and* $\psi = L(t', \text{-}) * \psi'$. *Then* $(t \le t' \le u \wedge \varphi \vdash \overrightarrow{\psi}, \psi), (t' < t \wedge \varphi \vdash \overrightarrow{\psi}, \psi), (u < t' \wedge \varphi \vdash \overrightarrow{\psi}, \psi) < (\varphi \vdash \overrightarrow{\psi}, \psi)$.

Proof. (1) and (2) are immediately shown since the order $\leq_{lex}$ on measures is a lexicographic order and the preorder $<$ on entailments is defined by $<_{lex}$.

(3) Assume $\Pi \subseteq \Pi'$. Take arbitrary φ and ψ. We have $\deg(\Pi' \wedge \Pi_\varphi, \sigma, \sigma') \leq \deg(\Pi \wedge \Pi_\varphi, \sigma, \sigma')$ since $\Pi \wedge \Pi_\varphi$ is satisfiable if $\Pi' \wedge \Pi_\varphi$ is satisfiable. Hence $|\psi|^{\mathrm{EM}}_{\Pi' \wedge \varphi} \leq |\psi|^{\mathrm{EM}}_{\Pi \wedge \varphi}$ holds. Therefore we have $||\psi||_{\Pi' \wedge \varphi} \leq ||\psi||_{\Pi \wedge \varphi}$.

(4) Suppose the assumption of the claim (4). We have $|\psi|^{\mathrm{EM}}_{t=t' \wedge \varphi} < |\psi|^{\mathrm{EM}}_{\varphi}$ since $\deg(t = t' \wedge \Pi_\varphi, t \mapsto (\text{-}, \text{-}), L(t', \text{-})) < \deg(\Pi_\varphi, t \mapsto (\text{-}, \text{-}), L(t', \text{-}))$. Hence $||\psi||_{t=t' \wedge \varphi} < ||\psi||_\varphi$ holds. We also have $||\psi_1||_{t=t' \wedge \varphi} < ||\psi_1||_\varphi$ by (3). Thus we obtain $(t = t' \wedge \varphi \vdash \overrightarrow{\psi}, \psi) < (\varphi \vdash \overrightarrow{\psi}, \psi)$. Similarly we can also show $(t \neq t' \wedge \varphi \vdash \overrightarrow{\psi}, \psi) < (\varphi \vdash \overrightarrow{\psi}, \psi)$.

(5) is shown in a similar way to (4). □

Lemma 7.8. *Let $\mathcal{R}$ be an inference rule other than (Start) and (UnsatL). Then we have $J' < J$ for any $J' \in \mathrm{Apply}_{\mathcal{R}}(J)$.*

Proof. We show the claim for each rule of $\mathcal{R}$.

The claim for (UnsatR) is shown by Lemma 7.7 (1).

The claims for ($\mapsto$LsEM), ($\mapsto$DllEM) and (ArrListEM) are shown by Lemma 7.7 (4).

The claims for (LsElim), (DllElim), (ArrLs) and (ArrDll) are shown by Lemma 7.7 (2) since the number of ls and dll is reduced.

The claim for ($\mapsto$Ls) is shown as follows: Let J and J' be $t \mapsto (v, w) * \varphi \vdash \mathrm{ls}(t', u') * \psi, \overrightarrow{\psi}$ and $t \mapsto (v, w) * \varphi \vdash t' = u' \wedge \psi, t \mapsto (v, w) * \mathrm{ls}(v, u') * \psi, \overrightarrow{\psi}$, respectively. Then we have $||t' = u' \wedge \psi||_{t \mapsto (v,w) * \varphi} < ||\mathrm{ls}(t', u') * \psi||_{t \mapsto (v,w) * \varphi}$ since the number of ls is reduced. We also have $||t \mapsto (v, w) * \mathrm{ls}(v, u') * \psi||_{t \mapsto (v,w) * \varphi} < ||\mathrm{ls}(t', u') * \psi||_{t \mapsto (v,w) * \varphi}$ since $\sharp_{\mathrm{lists}}(t \mapsto (v, w) * \mathrm{ls}(v, u') * \psi) = \sharp_{\mathrm{lists}}(\mathrm{ls}(t', u') * \psi)$ and $|t \mapsto (v, w) * \mathrm{ls}(v, u') * \psi|^{\mathrm{Unfold}}_{t \mapsto (v,w) * \varphi} < |\mathrm{ls}(t', u') * \psi|^{\mathrm{Unfold}}_{t \mapsto (v,w) * \varphi}$. Hence we have $J' < J$ by Lemma 7.7 (2).

The claim for ($\mapsto$Dll) is shown similarly to the case ($\mapsto$Ls). □

By the above lemma, we can show termination of `search`.

Lemma 7.9 (Termination)**.** `search`(J) *terminates for any J.*

Proof. Suppose that `search` does not terminate with an input J_0. Then there is an infinite sequence of recursive calls `search`(J_0), `search`(J_1), `search`(J_2), ... By Lemma 7.8, we have an infinite decreasing sequence $J_0 > J_1 > J_2 > \ldots$. This contradicts the well-foundedness of $<$. □

Lemma 7.10. *Let J be a valid entailment. Then either of (UnsatL), (Start), (UnsatR), ($\mapsto$LsEM), (ArrListEM), (LsElim), (ArrLs), ($\mapsto$Ls), ($\mapsto$DllEM), (DllElim), (ArrDll), or ($\mapsto$Dll) is applicable to J.*

Proof. If the antecedent of J is unsatisfiable, then (UnsatL) is applicable to J. If J is list-free, then (Start) is applicable, since J is valid. We consider the other cases.

Let $(\varphi \vdash \overrightarrow{\psi}) = J$. If there is $\varphi \in \overrightarrow{\psi}$ such that $\varphi \wedge \psi$ is unsatisfiable, then (UnsatR) is applicable. In the following, we assume that $\varphi \wedge \psi$ is satisfiable for any $\psi \in \overrightarrow{\psi}$. If φ does not contain either $\mapsto$ or Arr, then (LsElim) or (DllElim) is applicable to J. Otherwise φ contains $\mapsto$ or Arr, and $\overrightarrow{\psi}$ contains list predicates.

(1) The case that $\overrightarrow{\psi}$ contains the ls predicate. Fix $\mathrm{ls}(t', u')$ in $\overrightarrow{\psi}$. We consider the following subcases.

(1.1) The case that there is $t \mapsto (\text{-}, \text{-})$ in φ such that $t = t' \wedge \Pi_\varphi$ and $t \neq t' \wedge \Pi_\varphi$ are satisfiable. Then ($\mapsto$EM) is applicable.

(1.2) The case that there is $t \mapsto (\text{-}, \text{-})$ in φ such that $t \neq t' \wedge \Pi_\varphi$ is unsatisfiable (that is $\Pi_\varphi \models t = t'$). Then ($\mapsto$Ls) is applicable.

(1.3) The case that $t = t' \wedge \Pi_\varphi$ is unsatisfiable (that is $\Pi_\varphi \models t \neq t'$) for all $t \mapsto (\text{-}, \text{-})$ in φ. We consider the following subsubcases.

(1.3.1) The case that there is $\mathrm{Arr}(t, u)$ in φ such that $t \leq t' \leq u \wedge \Pi_\varphi$ and $(t' < t \vee u < t') \wedge \varphi_\varphi$ are satisfiable. Then (ArrLsEM) is applicable.

(1.3.2) The case that there is $\mathrm{Arr}(t, u)$ in φ such that $(t' < t \vee u < t') \wedge \Pi_\varphi$ is unsatisfiable (that is $\Pi_\varphi \models t \leq t' \leq u$). Then (ArrLs) is applicable.

(1.3.3) The case that $t \leq t' \leq u \wedge \Pi_\varphi$ is unsatisfiable (that is $\Pi_\varphi \models t' < t \vee u < t'$) for all $\mathrm{Arr}(t, u)$ in φ. This case (LsElim) is applicable since $\Pi_\varphi \not\models \Sigma_\varphi \rightarrow\!\!\downarrow t'$.

(2) The case that $\overrightarrow{\psi}$ does not contain the ls predicate. There exists $\mathrm{dll}(t', u', v', w')$ in $\overrightarrow{\psi}$. We can show the claim for this case in a similar way to (1). □

By using the results of this section, we can show correctness of the proof search algorithm.

Proposition 7.11 (Correctness)**.** *J is valid if and only if* `search`(J) *returns a proof of J.*

Proof. The if-part is shown by soundness of the proof system (Proposition 7.5). We show the only-if part. Assume that J is a valid entailment. By Lemma 7.9, `search`(J) terminates. We show the claim by induction on computation of `search`(J). By Lemma 7.10, some rule $\mathcal{R}$ is applicable to J. Then $J_1, \ldots, J_k$ are obtained by $\mathrm{Apply}_\mathcal{R}(J)$. By local completeness of the proof system (Proposition 7.5), each J_k is valid. Hence, by the induction hypothesis, `search`(J_i) returns a tuple that represents a proof of J_i for any $i \in \{1, \ldots, k\}$. Therefore `search`(J) returns a tuple that represents a proof of J. □

8. Decidability of Entailment Problem for **SLAL**

This section shows the second theorem of this paper, namely the decidability of the entailment problem of **SLAL**, by combining the four results of the previous sections.

Theorem 8.1 (Decidability of **SLAL**)**.** *Checking the validity of entailments in* **SLAL** *is decidable.*

Proof. We first give the decision procedure for **SLAL**. An entailment J of **SLAL** is given as an input for the procedure. Then it performs as follows. (i) The decision procedure eliminates the list predicates that appear in the antecedent of J by using the unroll collapse (Propositions 6.1 and 6.2). Then it obtains entailments $J_1, \ldots, J_k$ whose antecedents are list-free. (ii) It computes `search`(J_i) $(i = 1, \ldots, k)$. It returns "valid" if each of `search`(J_i) returns a tuple. Otherwise it returns "invalid".

Termination property of the decision procedure can be obtained from the termination property of `search` (Proposition 7.9). Correctness of the decision procedure is stated as follows: the decision procedure returns "valid" for an input J if and only if J is valid. We show this. By the unroll collapse, the validity of J is equivalent to that of $J_1, \ldots, J_k$. Hence J is valid if and only if `search`(J_i) returns a proof of J_i for all i by the correctness property of `search` (Proposition 7.11). This is equivalent to that the decision procedure returns "valid". □

Example 8.2. We show how our decision procedure works with an example $\mathrm{Arr}(1,2) * 3 \mapsto (10,0) * \mathrm{ls}(10,20) \vdash \mathrm{Arr}(1,3) * \mathrm{ls}(10,20)$.

By using the unroll collapse (Proposition 6.1) taking $\mathrm{ls}(t,u)$ to be $\mathrm{ls}(10,20)$, ϕ to be $\mathrm{Arr}(1,2) * 3 \mapsto (10,0)$, and $\overrightarrow{\psi}$ to be $\mathrm{Arr}(1,3) * \mathrm{ls}(10,20)$, we obtain the following entailments:

$$(J_a) \quad 10 = 20 \wedge \mathrm{Arr}(1,2) * 3 \mapsto (10,0) \vdash \mathrm{Arr}(1,3) * \mathrm{ls}(10,20),$$
$$(J_b) \quad \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \vdash \mathrm{Arr}(1,3) * \mathrm{ls}(10,20).$$

Then $\texttt{search}(J_a)$ and $\texttt{search}(J_b)$ are performed. $\texttt{search}(J_a)$ immediately returns a tuple (UnsatL, J_a), since the antecedent is unsatisfiable. Computation of $\texttt{search}(J_b)$ is done as follows:

Begin $\texttt{search}(J_b)$: $\texttt{search}(J_b)$ is called. Then $\mathcal{R}$ is set by ($\mapsto$Ls), since it is applicable to J_b. $\mathrm{Apply}_{\mathcal{R}}(J_b)$ is performed. An instance of ($\mapsto$Ls) is non-deterministically chosen: $t \mapsto (v,w)$ is taken to be $10 \mapsto (z,y)$, φ is taken to be $\mathrm{Arr}(1,2) * 3 \mapsto (10,0) * z \mapsto (20,w)$, $\mathrm{ls}(t',u')$ is taken to be $\mathrm{ls}(10,20)$, and ψ is taken to be $\mathrm{Arr}(1,3)$. Then $\mathrm{Apply}_{\mathcal{R}}(J_b)$ produces the following one subgoal:

$$(J_{b1}) \quad \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \vdash 10 = 20 \wedge \mathrm{Arr}(1,3), \mathrm{Arr}(1,3) * 10 \mapsto (z,y) * \mathrm{ls}(z,20).$$

Begin $\texttt{search}(J_{b1})$: $\texttt{search}(J_{b1})$ is called. Then $\mathcal{R}$ is set by (UnsatR). $\mathrm{Apply}_{\mathcal{R}}(J_{b1})$ is performed. An instance of (UnsatR) is chosen, where φ is taken to be the antecedent of J_{b1}, ψ is taken to be $10 = 20 \wedge \mathrm{Arr}(1,3)$, and $\overrightarrow{\psi}$ is taken to be $\mathrm{Arr}(1,3) * 10 \mapsto (z,y) * \mathrm{ls}(z,20)$. Then $\mathrm{Apply}_{\mathcal{R}}(J_{b1})$ produces the following one subgoal:

$$(J_{b2}) \quad \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \vdash \mathrm{Arr}(1,3) * 10 \mapsto (z,y) * \mathrm{ls}(z,20).$$

Begin $\texttt{search}(J_{b2})$: $\texttt{search}(J_{b2})$ is called. Then $\mathcal{R}$ is set by ($\mapsto$Ls). $\mathrm{Apply}_{\mathcal{R}}(J_{b2})$ is performed. An instance of ($\mapsto$Ls) is chosen, where $t \mapsto (v,w)$ is taken to be $z \mapsto (20,w)$, φ is taken to be $\mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y)$, $\mathrm{ls}(t',u')$ is taken to be $\mathrm{ls}(z,20)$, ψ is taken to be $\mathrm{Arr}(1,3) * 10 \mapsto (z,y)$, and $\overrightarrow{\psi}$ is taken to be empty. Then $\mathrm{Apply}_{\mathcal{R}}(J_{b2})$ produces the following one subgoal:

$$(J_{b3}) \quad \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \vdash z = 20 \wedge \mathrm{Arr}(1,3) * 10 \mapsto (z,y), z \mapsto (20,w) * \mathrm{ls}(20,20) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y).$$

Begin $\texttt{search}(J_{b3})$: $\texttt{search}(J_{b3})$ is called. Then $\mathcal{R}$ is set by (UnsatR). $\mathrm{Apply}_{\mathcal{R}}(J_{b3})$ is performed. An instance of (UnsatR) is chosen, where φ is taken to be the antecedent of J_{b3}, ψ is taken to be $z = 20 \wedge \mathrm{Arr}(1,3) * 10 \mapsto (z,y)$, and $\overrightarrow{\psi}$ is taken to be $z \mapsto (20,w) * \mathrm{ls}(20,20) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y)$. Now $\varphi \wedge \psi$ is unsatisfiable, since the sizes of the required heaps by these φ and ψ are different.

Then $\mathrm{Apply}_{\mathcal{R}}(J_{b3})$ produces the following one subgoal:

$$(J_{b4}) \quad \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \vdash z \mapsto (20,w) * \mathrm{ls}(20,20) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y).$$

Begin $\texttt{search}(J_{b4})$: $\texttt{search}(J_{b4})$ is called. Then $\mathcal{R}$ is set by ($\mapsto$LsEM). $\mathrm{Apply}_{\mathcal{R}}(J_{b4})$ is performed. An instance of ($\mapsto$LsEM) is non-deterministically chosen, where $t \mapsto (v,w)$ is taken to be $z \mapsto (20,w)$, φ is taken to be $\mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y)$, $\mathrm{ls}(t',u')$ is

taken to be $\mathrm{ls}(20, 20)$, ψ is taken to be the succedent of J_{b4}, and $\overrightarrow{\psi}$ is taken to be empty. Then $\mathrm{Apply}_{\mathcal{R}}(J_{b4})$ produces the following two subgoals:

$$\begin{aligned}(J_{b51}) \quad & z \neq 20 \wedge \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \\ & \vdash \quad z \mapsto (20,w) * \mathrm{ls}(20,20) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y), \text{ and} \\ (J_{b52}) \quad & z = 20 \wedge \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \\ & \vdash \quad z \mapsto (20,w) * \mathrm{ls}(20,20) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y).\end{aligned}$$

Begin `search`(J_{b51}): `search`(J_{b51}) is called. Then $\mathcal{R}$ is set by (LsElim). $\mathrm{Apply}_{\mathcal{R}}(J_{b51})$ is performed. An instance of (LsElim) is chosen, where φ is taken to be the antecedent of J_{b51}, $\mathrm{ls}(t', u')$ is taken to be $\mathrm{ls}(20, 20)$, ψ is taken to be $z \mapsto (20, w) * \mathrm{Arr}(1, 3) * 10 \mapsto (z, y)$, and $\overrightarrow{\psi}$ is taken to be empty. Then $\mathrm{Apply}_{\mathcal{R}}(J_{b51})$ produces the following one subgoal:

$$\begin{aligned}(J_{b61}) \quad & z \neq 20 \wedge \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \\ & \vdash 20 = 20 \wedge z \mapsto (20,w) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y).\end{aligned}$$

Begin `search`(J_{b61}): `search`(J_{b61}) is called. Then $\mathcal{R}$ is set by (Start). $\mathrm{Apply}_{\mathcal{R}}(J_{b61})$ returns the empty list, since J_{b61} is evaluated to be valid by the decision procedure for **SLA** given by Theorem 3.4.

End `search`(J_{b61}): `search`(J_{b61}) returns a tuple $(\mathrm{Start}, J_{b61})$, which is written as $\mathcal{T}_{b61}$.

End `search`(J_{b51}): `search`(J_{b51}) returns $(\mathrm{LsElim}, J_{b51}, \mathcal{T}_{b61})$, which is written as $\mathcal{T}_{b51}$.

Begin `search`(J_{b52}): `search`(J_{b52}) is called. Then $\mathcal{R}$ is set by ($\mapsto$Ls). $\mathrm{Apply}_{\mathcal{R}}(J_{b52})$ is performed. An instance of ($\mapsto$Ls) is chosen, where $t \mapsto (v, w)$ is taken to be $z \mapsto (20, w)$, φ is taken to be $z = 20 \wedge \mathrm{Arr}(1, 2) * 3 \mapsto (10, 0) * 10 \mapsto (z, y)$, $\mathrm{ls}(t', u')$ is taken to be $\mathrm{ls}(20, 20)$, and ψ is taken to be $z \mapsto (20, w) * \mathrm{Arr}(1, 3) * 10 \mapsto (z, y)$, and $\overrightarrow{\psi}$ is taken to be empty. Then $\mathrm{Apply}_{\mathcal{R}}(J_{b52})$ produces the following one subgoal:

$$\begin{aligned}(J_{b62}) \quad & z = 20 \wedge \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \\ & \vdash \quad 20 = 20 \wedge z \mapsto (20,w) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y), \\ & \qquad z \mapsto (20,w) * 20 \mapsto (20,w) * \mathrm{ls}(20,20) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y).\end{aligned}$$

Begin `search`(J_{b62}): `search`(J_{b62}) is called. Then $\mathcal{R}$ is set by (UnsatR), since the second clause in the succedent of J_{b62} is unsatisfiable under the assumption $z = 20$. Then $\mathrm{Apply}_{\mathcal{R}}(J_{b62})$ produces the following one subgoal:

$$\begin{aligned}(J_{b72}) \quad & z = 20 \wedge \mathrm{Arr}(1,2) * 3 \mapsto (10,0) * 10 \mapsto (z,y) * z \mapsto (20,w) \\ & \vdash 20 = 20 \wedge z \mapsto (20,w) * \mathrm{Arr}(1,3) * 10 \mapsto (z,y).\end{aligned}$$

Begin `search`(J_{b72}): `search`(J_{b72}) is called. It immediately terminates, since J_{b72} is evaluated to be valid by the decision procedure for **SLA**.

End `search`(J_{b72}): `search`(J_{b72}) returns a tuple $(\mathrm{Start}, J_{b72})$, which is written as $\mathcal{T}_{b72}$.

End `search`(J_{b62}): `search`(J_{b62}) returns $(\mathrm{UnsatR}, J_{b62}, \mathcal{T}_{b72})$, which is written as $\mathcal{T}_{b62}$.

End `search`(J_{b52}): `search`(J_{b52}) returns $(\mapsto \mathrm{Ls}, J_{b52}, \mathcal{T}_{b62})$, which is written as $\mathcal{T}_{b52}$.

End `search`(J_{b4}): `search`(J_{b4}) returns $(\mapsto \mathrm{LsEM}, J_{b4}, \mathcal{T}_{b51}, \mathcal{T}_{b52})$, which is written as $\mathcal{T}_{b4}$.

End `search`(J_{b3}): `search`(J_{b3}) returns $(\mathrm{UnsatR}, J_{b3}, \mathcal{T}_{b4})$, which is written as $\mathcal{T}_{b3}$.

End `search`(J_{b2}): `search`(J_{b2}) returns $(\mapsto \mathrm{Ls}, J_{b2}, \mathcal{T}_{b3})$, which is written as $\mathcal{T}_{b2}$.

End `search`(J_{b1}): `search`(J_{b1}) returns $(\mathrm{UnsatR}, J_{b1}, \mathcal{T}_{b2})$, which is written as $\mathcal{T}_{b1}$.

End `search`(J_b): `search`(J_b) returns $(\mapsto \mathrm{Ls}, J_b, \mathcal{T}_{b1})$, which is written as $\mathcal{T}_b$.

Finally our decision procedure answers "Valid", since each of $\texttt{search}(J_a)$ and $\texttt{search}(J_b)$ returns a tuple.

9. Conclusion

We have shown the decidability results for the validity checking problem of entailments for **SLA** and **SLAL**. First we have given the decision procedure for **SLA** and proved its correctness under the condition that the sizes of arrays in the succedent are not existentially quantified. The key idea of the decision procedure is the notion of the sorted entailments. By using this idea, we have defined the translation P of a sorted entailment into a formula in Presburger arithmetic. Secondly we have proved the decidability for **SLAL**. The key idea of the decision procedure is to extend the unroll collapse technique given in [2] to arithmetic and arrays as well as doubly-linked list segments. We have also given a proof system and showed correctness of the proof search algorithm for eliminating the list predicates in the succedent of an entailment.

We require the condition in the decidability for **SLA** (Theorem 3.4) from a technical reason. It would be future work to show the decidability without this condition.

Acknowledgments. This is partially supported by Core-to-Core Program (A. Advanced Research Networks) of the Japan Society for the Promotion of Science.

References

[1] Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max Kanovich, and Joël Ouaknine. Foundations for Decision Problems in Separation Logic with General Inductive Predicates. In: *Proceedings of the 17th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 8412 of *Lecture Notes in Computer Science*, pages 411–425, 2014, Springer.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A Decidable Fragment of Separation Logic. In: *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 3328 of *Lecture Notes in Computer Science*, pages 97–109, 2004, Springer.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In: *Proceedings of the third Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68, 2005, Springer.

[4] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. A Logic-Based Framework for Reasoning About Composite Data Structures. In: *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR)*, volume 5710 of *Lecture Notes in Computer Science*, pages 178–195, 2009, Springer.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma, What's Decidable About Arrays? In: *Proceedings of the 7th International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442, 2006, Springer.

[6] James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan A. Navarro Pérez. A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates. In: *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, Article No.25, ACM.

[7] James Brotherston, Nikos Gorogiannis, and Max Kanovich. Biabduction (and Related Problems) in Array Separation Logic. In: *Proceedings of the 26th International Conference on Automated Deduction (CADE)*, volume 10395 of *Lecture Notes in Computer Science*, pages 472–490, 2017, Springer.

[8] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In: *Proceedings of the the 13rd International Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 182–203, 2006, Springer.

[9] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. In: *Journal of ACM*, volume 58 Issue 6, pages 1–66, 2011, ACM.

[10] Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: *Proceedings of NASA Formal Methods Symposium*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465, 2011, Springer.

[11] Byron Cook, Christoph Haase, Jöel Ouaknine, Matthew Parkinson, and James Worrell. Tractable reasoning in a fragment of Separation Logic. In: *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR)*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249, 2011, Springer.

[12] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures using Separation Logic. In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 372–378, 2011, Springer.

[13] Constantin Enea, Vlad Saveluc, and Mihaela Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In: *Proceeding of 22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 129–148, 2013, Springer.

[14] Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. In: *Proceedings of the 12th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 8858 of *Lecture Notes in Computer Science*, pages 314–333, 2014, Springer.

[15] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. The Tree Width of Separation Logic with Recursive Definitions. In: *Proceedings of the 24th International Conference on Automated Deduction (CADE)*, volume 7898 of *Lecture Notes in Computer Science*, pages 21–38, 2013, Springer.

[16] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Deciding Entailments in Inductive Separation Logic with Tree Automata. In: *Proceedings of International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8837 of *Lecture Notes in Computer Science*, pages 201–218, 2014, Springer.

[17] Daisuke Kimura and Makoto Tatsuta. Decision Procedure for Entailment of Symbolic Heaps with Arrays. In: *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 10695 of *Lecture Notes in Computer Science*, pages 169–189, 2017, Springer.

[18] Shuvendu Lahiri and Shaz Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 171–182, ACM.

[19] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic Using SMT. In: *Proceedings of the 25th international conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 773–789, 2013, Springer.

[20] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002, IEEE Computer Society.

[21] Makoto Tatsuta and Daisuke Kimura. Separation Logic with Monadic Inductive Definitions and Implicit Existentials. In: *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 9458 of *Lecture Notes in Computer Science*, pages 69–89, 2015, Springer.