Decision Procedure for Entailment of Symbolic Heaps with Arrays

Daisuke Kimura[1] and Makoto Tatsuta[2]

[1] Toho University
kmr@is.sci.toho-u.ac.jp
[2] National Institute of Informatics
tatsuta@nii.ac.jp

Abstract. This paper gives a decision procedure for the validity of entailment of symbolic heaps in separation logic with Presburger arithmetic and arrays. The correctness of the decision procedure is proved under the condition that sizes of arrays in the succedent are not existentially bound. This condition is independent of the condition proposed by the CADE-2017 paper by Brotherston et al, namely, one of them does not imply the other. For improving efficiency of the decision procedure, some techniques are also presented. The main idea of the decision procedure is a novel translation of an entailment of symbolic heaps into a formula in Presburger arithmetic, and to combine it with an external SMT solver. This paper also gives experimental results by an implementation, which shows that the decision procedure works efficiently enough to use.

1 Introduction

Separation logic can be used to verify/analyze heap-manipulating imperative programs with pointers, and mainly it is successful for verify/analyze memory safety [3]. The aim of our paper is also memory safety. The advantage of separation logic is modularity by the frame rule, by which we can independently verify/analyze each function that may manipulate heaps [9]. The study in this paper goes along this line.

The final goal of our research is to develop a fully-automated program verifier of pointer programs based on separation logic. For this, this paper introduces a formal system of symbolic heap fragment of separation logic with arrays, shows decidability of its entailment problem, then gives an implementation (called **SLar**) of our decision procedure, and finally discusses its improvement for efficiency.

Symbolic heaps are formulas of separation logic in a simple form $\exists\overrightarrow{x}(\Pi \wedge \Sigma)$. The pure part Π describes properties of between terms (denoted by t, u), which represent memory addresses and values. The spatial part Σ is a separating conjunction of the empty predicate emp, the points-to predicate $t \mapsto u$, and the array predicate $\mathrm{Arr}(t, u)$. It represents some shape of heaps: emp means the empty heap, $t \mapsto u$ means the single heap that uses only one address t and the value at t is u, $\mathrm{Arr}(t, u)$ means the heap that contains only an array starting from t ending at u, a separating conjunction $\Sigma_1 * \Sigma_2$ means a heap which can be split into two disjoint sub-heaps that are represented by Σ_1 and Σ_2.

In order to achieve our final goal, it is necessary to develop a solver for the entailment problem. An *entailment* has the form $\phi \vdash \phi_1, \ldots, \phi_k$, where ϕ and ϕ_i are symbolic heaps. It is said to be valid when $\phi \to \bigvee_i \phi_i$ is valid with respect to the usual heap model. The *entailment problem* is the validity checking problem of given entailments.

In the literature, many researches for verification of pointer programs based on symbolic-heap systems have been done. In particular symbolic-heap systems with inductive predicates have been studied intensively [2, 3, 1, 6, 10–14, 19]. Berdine et al. [2, 3] introduced the symbolic-heap system with hard-coded list and tree predicates, and showed decidability of its entailment problem. Iosif et al. [13, 14] considered the system with general inductive predicates, and showed its decidability under the bounded tree-width condition. Tatsuta et al. [19] introduced the system with general monadic inductive predicates.

Array is one of the primitive data structures of pointer programs. It is an important issue of verifying pointer programs to ensure that there is no buffer overflow. So we need to consider the array structure as primitive. However, as far as we know, there are two researches about symbolic-heap systems that have arrays as primitive [8, 7]. Calcagno et al. [8] studied shape analysis based on symbolic-heap system in the presence of pointer arithmetic. Brotherston et al. [7] investigated several problems about a symbolic-heap system with arrays.

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. [15] discussed logics for arrays but their systems are essentially different from separation logic. We cannot apply their techniques to our case. Piskac et al. [16] proposed a separation logic system with list segments, and it can be combined with various SMT solvers, including array logics. However, if 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. So their techniques cannot solve our case.

In [7], they proposed a decision procedure for the entailment problem by giving an equivalent condition to 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 a points-to predicate in the succedent of an entailment is not existentially bound.

Our motivating example is $\mathrm{Arr}(x, x) \vdash \exists y(x \mapsto y)$, since it is simple and trivially true and we expect an entailment checker could decide it. This example is important because it expresses a basic property of arrays and is one of the simplest example that contains all of the points-to predicate, the array predicate, and existential quantifier. So far there was no decision procedure of a class of entailments which contains this example, since it does not satisfy the restriction of [7].

The current paper shows decidability of the entailment problem under a condition: the sizes of arrays in the succedent of the given entailment do not contain any existential variables. It means that the shape of heaps represented by the succedent is completely determined by the antecedent. We need this condition for proving correctness of our decision procedure. Our result decides an independent class of entailments (including our motivating example) to the class decided by [6]. That is, our class neither contains the class of [6] nor is contained by it.

The basic idea of our decision procedure is a novel translation of a given entailment into an equivalent formula in Presburger arithmetic. The key idea used in the translation is the notion of "*sorted*" symbolic heaps. Any heap represented by a sorted symbolic heap has addresses arranged in the order of the spatial part of the symbolic heap. If we assume the both sides of given entailment are sorted, the entailment is valid if no contradiction is found in comparing spatial parts on both sides starting from left to right.

We also propose two ideas for improving the performance of our decision procedure. The performance heavily depends on the size (number of the separating conjunction symbol $*$) of a given entailment. Consider a single conclusion entailment $\phi_1 \vdash \phi_2$. Let n and m be numbers of the separating conjunction in ϕ_1 and ϕ_2, respectively. Then this entailment will be decomposed into $n!$ sorted entailments with $m!$ disjunctions on the right-hand side. So it is quite important to reduce the number of $*$ as much as possible at an early stage of the procedure.

This paper also presents our entailment checker **SLar**, which is an implementation of our decision procedure. **SLar** first (1) optimizes a given entailment according to the improvement idea mentioned above, (2) decomposes the resulting entailment into some sorted entailments, then (3) translates the decomposed entailments into the corresponding Presburger formulas, and finally (4) checks their validity by invoking an external SMT solver **Z3** [20]. The original entailment is answered valid if and only-if all of the decomposed sorted entailments are valid. The improvement techniques made our system run in a second in most cases of experiments, and made our system 200 times faster in some cases.

We introduce our system of separation logic with arrays in Section 2. Section 3 defines the decision procedure of the entailment problem of the system. The correctness of the decision procedure is shown in Section 4. Two improvement ideas of the decision procedure are discussed in Section 5. Section 6 discusses the entailment checker **SLar** based on this decision algorithm, and evaluates its performance with experimental data. Section 7 concludes.

2 Separation Logic with Arrays

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

2.1 Syntax of System of Separation Logic with Arrays

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

Terms $t ::= x \mid 0 \mid 1 \mid 2 \mid ... \mid t + t$.

Formulas $\varphi ::= t = t \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x\varphi \mid \mathrm{emp} \mid t \mapsto t \mid \mathrm{Arr}(t, t) \mid \varphi * \varphi$.

An atomic formula of the form $t \mapsto u$ or $\mathrm{Arr}(t, u)$ is called a points-to atomic formula or an array atomic formula, respectively. The truth of each formula is interpreted under a state of variables and a heap: emp is true when the heap is empty; $t \mapsto u$ is true when the heap only has a single memory cell of address t that contains the value u; $\mathrm{Arr}(t, u)$ is true when the heap only has an array of index from t to u; a separating conjunction $\varphi_1 * \varphi_2$ is true when the heap is split into two disjoint sub-heaps, φ_1 is true under one, and φ_2 is true under the other. The formal definition of these interpretation is given in the next subsection.

The set of free variables (denoted by $\mathrm{FV}(\varphi)$) of φ is defined as usual. We also define $\mathrm{FV}(\overrightarrow{\varphi})$ as the union of $\mathrm{FV}(\varphi)$, where $\varphi \in \overrightarrow{\varphi}$.

We sometimes use the symbol σ to denote emp, $t \mapsto u$, or $\mathrm{Arr}(t, u)$.

We use abbreviations $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\forall x\varphi$ defined in a usual way. We also write $t \neq u$, $t \leq u$, $t < u$, and true as abbreviations of $\neg(t = u)$, $\exists x.(u = t + x)$, $t + 1 \leq u$, and $0 = 0$, respectively.

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

2.2 Semantics of System of Separation Logic with Arrays

Let N be the set of natural numbers. We define the following semantic domains:

$\mathrm{Val} =_{\mathrm{def}} N$, $\quad \mathrm{Loc} =_{\mathrm{def}} N \setminus \{0\}$, $\quad \mathrm{Stores} =_{\mathrm{def}} \mathrm{Vars} \rightarrow \mathrm{Val}$, $\quad \mathrm{Heaps} =_{\mathrm{def}} \mathrm{Loc} \rightarrow_{\mathrm{fin}} \mathrm{Val}$.

Loc means addresses of heaps. 0 means Null. An element s in Stores is called a *store* that means a valuation of variables. 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 $i = 1, 2$. 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 $s(t + u) = s(t) + s(u)$.

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

$s, h \models t = u$ iff $s(t) = s(u)$,

$s, h \models \varphi_1 \wedge \varphi_2$ iff $s, h \models \varphi_1$ and $s, h \models \varphi_2$,

$s, h \models \neg\varphi$ iff $s, h \not\models \varphi$,

$s, h \models \exists x\varphi$ iff $s[x := a], h \models \varphi$ for some $a \in \mathrm{Val}$,

$s, h \models \mathrm{emp}$ iff $\mathrm{Dom}(h) = \emptyset$,

$s, h \models t \mapsto u$ iff $\mathrm{Dom}(h) = \{s(t)\}$ and $h(s(t)) = s(u)$,

$s, h \models \mathrm{Arr}(t, u)$ iff $\mathrm{Dom}(h) = \{x \in N \mid s(t) \leq x \leq s(u)\}$ and $s(t) \leq s(u)$,

$s, h \models \varphi_1 * \varphi_2$ iff $s, h_1 \models \varphi_1$, $s, h_2 \models \varphi_2$, and $h = h_1 + h_2$ for some h_1, h_2.

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

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

Let I be a finite set. In this paper we implicitly assume a linear order on I (this order will be used in the definition of the translation P given in the next section). Then we sometimes write $\{\phi_i \mid i \in I\}$ for $\bigvee\{\phi_i \mid i \in I\}$, that is, disjunction $\bigvee_{i \in I} \phi_i$ of formulas ϕ_i $(i \in I)$ under the order of I. We sometimes abbreviate $\{\phi_i \mid i \in I\}$ by $\{\phi_i\}_{i \in I}$. It may be further abbreviated by $\overrightarrow{\phi}$ when I is not important.

2.3 Symbolic-Heap System with Arrays

The symbolic-heap system **SLAR** is defined as a fragment of **G**. The syntax of **SLAR** is given as follows. Terms of **SLAR** are the same as the terms of **G**. Formulas of **SLAR** (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

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

We sometimes write $\exists \overrightarrow{x} \Sigma$ as an abbreviation of $\exists \overrightarrow{x}(\mathrm{true} \wedge \Sigma)$. We use notations Π_ϕ and Σ_ϕ that mean the pure part and the spatial part of ϕ, respectively.

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

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

The symbolic heap on the left-hand side is called the antecedent of the entailment. The symbolic heaps on the right-hand side are called the succedents of the entailment. As we noted before, 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 $\phi \models \{\phi_i \mid i \in I\}$ 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 for 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 [17] except we extend the allocation command for allocating more than one cells. We define our programming language in programming language C style.

Expressions $e ::= x \mid 0 \mid 1 \mid 2 \ldots \mid e + e$.

Boolean expressions $b ::= e == e \mid e < e \mid b \&\& b \mid b || b \mid !b$.

Programs $P ::= x = e; \mid \text{if } (b)\{P\} \text{ else } \{P\}; \mid \text{while } (b)\{P\}; \mid P\ P \mid$
$x = \texttt{malloc}(y); \mid x = *y; \mid *x = y; \mid \text{free}(x);$.

$x = \texttt{malloc}(y);$ 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,

Assertions $A ::= \phi_1 \vee \cdots \vee \phi_n$.

In the same way as [17], we use a triple $\{A\}\,\texttt{P}\,\{B\}$ 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 for memory operations. We write $\text{Arr2}(x, y)$ for $\exists z(\text{Arr}(x, z) \wedge x + y = z + 1)$. $\text{Arr2}(x, y)$ means the memory block at address x of size y. We sometimes write a formula that is not a disjunction of symbolic heaps for an equivalent assertion obtained by ordinary logical equivalence rules.

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

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

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

For the consequence rule

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

we have to check the side condition $A \to A'$. 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

3.1 Sorted Entailments

This subsection describes our key idea, namely *sorted* symbolic heaps. The addresses of heaps represented by a sorted symbolic heap must be sorted, that is, their order can be determined by the order of the spatial part of the sorted symbolic heap.

We sometimes regard Σ as a list of emp, $t \mapsto u$, and $\mathrm{Arr}(t, u)$. We also regard the symbol $*$ written like $t \mapsto u * \Sigma$ as the list constructor. By abuse of notation, we write emp in order to represent the empty list.

A symbolic heap ϕ is called *sorted* at (s, h) if s, h satisfies ϕ and the addresses of the heap h are arranged in the order of the spatial part of ϕ.

In order to express this notion, we introduce pure formulas $t < \Sigma$ and $\mathrm{Sorted}(\Sigma)$, which mean the first address expressed by Σ is greater than t, and Σ is sorted, respectively. They are inductively defined as follows:

$t < \mathrm{emp} =_{\mathrm{def}} \mathrm{true}$, $\quad t < (\mathrm{emp} * \Sigma_1) =_{\mathrm{def}} t < \Sigma_1$,

$t < (t_1 \mapsto u_1 * \Sigma_1) =_{\mathrm{def}} t < t_1$, $\quad t < (\mathrm{Arr}(t_1, u_1) * \Sigma_1) =_{\mathrm{def}} t < t_1$,

$\mathrm{Sorted}'(\mathrm{emp}) =_{\mathrm{def}} \mathrm{true}$,

$\mathrm{Sorted}'(\mathrm{emp} * \Sigma_1) =_{\mathrm{def}} \mathrm{Sorted}'(\Sigma_1)$,

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

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

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

The formula $\mathrm{Sorted}(\Sigma) \wedge \Sigma$ is sometimes abbreviated by $\widetilde{\Sigma}$ or $\Sigma^{\sim}$. We also write $\widetilde{\phi}$ (or $\phi^{\sim}$) for the symbolic heap which is obtained from ϕ by replacing Π_ϕ by $\Pi_\phi \wedge \mathrm{Sorted}(\Sigma_\phi)$.

We claim that ϕ is a sorted symbolic heap at (s, h) iff $\widetilde{\phi}$ is true in (s, h).

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 $\Sigma_{\phi'} \in \mathrm{Perm}(\Sigma_\phi)$ and the other parts of ϕ and ϕ' are same. We write $\mathrm{Perm}(\phi)$ for the set of permutations of ϕ.

Note that $s, h \models \phi$ iff $s, h \models \widetilde{\phi}'$ for some $\phi' \in \mathrm{Perm}(\phi)$.

An entailment is said to be sorted if all of its antecedent and succedents have the form $\widetilde{\phi}$. We claim that checking validity of entailments can be reduced to checking validity of sorted entailments. The formal statement of this property will be given later (see Lemma 1).

The basic idea of our decision procedure is as follows: (1) A given entailment is decomposed into sorted entailments according to Lemma 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).

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 $\bigwedge_{j \in J} u < t_j$.

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 top to 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

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

$$
\begin{array}{ll}
P(\varPi, t \mapsto u * \varSigma, \{(\varPi_i, t_i \mapsto u_i * \varSigma_i)\}_{i \in I}) & (\mapsto\mapsto) \\
=_{\mathrm{def}} P(\varPi \wedge t < \varSigma, \varSigma, \{(\varPi_i \wedge t = t_i \wedge u = u_i \wedge t_i < \varSigma_i, \varSigma_i)\}_{i \in I}) &
\end{array}
$$

$$
\begin{array}{ll}
P(\varPi, t \mapsto u * \varSigma, \{(\varPi_i, \mathrm{Arr}(t_i, t_i') * \varSigma_i)\} \cup S) & (\mapsto\textbf{Arr}) \\
=_{\mathrm{def}} P(\varPi \wedge t_i' = t_i, t \mapsto u * \varSigma, \{(\varPi_i, t_i \mapsto u * \varSigma_i)\} \cup S) & \\
\quad \wedge\ P(\varPi \wedge t_i' > t_i, t \mapsto u * \varSigma, \{(\varPi_i, t_i \mapsto u * \mathrm{Arr}(t_i + 1, t_i') * \varSigma_i)\} \cup S) & \\
\quad \wedge\ P(\varPi \wedge t_i' < t_i, t \mapsto u * \varSigma, S) &
\end{array}
$$

$$
\begin{array}{ll}
P(\varPi, \mathrm{Arr}(t, t') * \varSigma, S) & (\textbf{Arr}\mapsto) \\
=_{\mathrm{def}} P(\varPi \wedge t' > t, t \mapsto z * \mathrm{Arr}(t + 1, t') * \varSigma, S) & \\
\quad \wedge\ P(\varPi \wedge t' = t, t \mapsto z' * \varSigma, S), \quad \text{where } (\varPi'', t'' \mapsto u'' * \varSigma'') \in S \text{ and } z, z' \text{ are fresh} &
\end{array}
$$

$$
\begin{array}{l}
P(\varPi, \mathrm{Arr}(t, t') * \varSigma, \{(\varPi_i, \mathrm{Arr}(t_i, t_i') * \varSigma_i)\}_{i \in I}) \qquad (\textbf{ArrArr}) \\
=_{\mathrm{def}} \bigwedge_{I' \subseteq I} P\left(\begin{array}{l} \varPi \wedge m = m_{I'} \wedge m < m_{I \setminus I'} \wedge t \le t' \wedge t' < \varSigma, \varSigma, \\ \{(\varPi_i \wedge t_i + m < \varSigma_i, \varSigma_i)\}_{i \in I'} \cup \{(\varPi_i, \mathrm{Arr}(t_i + m + 1, t_i') * \varSigma_i)\}_{i \in I \setminus I'} \end{array} \right) \\
\wedge \bigwedge_{\emptyset \neq I' \subseteq I} P\left(\begin{array}{l} \varPi \wedge m' < m \wedge m' = m_{I'} \wedge m' < m_{I \setminus I'}, \mathrm{Arr}(t + m' + 1, t') * \varSigma, \\ \{(\varPi_i \wedge t_i + m' < \varSigma_i, \varSigma_i)\}_{i \in I'} \cup \{(\varPi_i, \mathrm{Arr}(t_i + m' + 1, t_i') * \varSigma_i)\}_{i \in I \setminus I'} \end{array} \right),
\end{array}
$$

where m, m_i, and m' are abbreviations of $t' - t$, $t_i' - t_i$, and $m_{\min I'}$, respectively.

Fig. 1. The translation P

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.

The $=_{\mathrm{def}}$ steps terminate since $(|\varSigma| + \sum_{i \in I} |\varSigma_i|, |S|)$ decreases where $|\varSigma|$ is the number of $*$ in $\varSigma$ and $|S|$ is the number of elements in S. Note that this measure does not decrease for some $=_{\mathrm{def}}$, but the left-hand sides of the definition are mutually exclusive and hence combination of $=_{\mathrm{def}}$ eventually decreases the measure. For example, $(\mapsto\mapsto)$ will eventually come after $(\textbf{Arr}\mapsto)$.

The formula $P(\varPi, \varSigma, \{(\varPi_i, \varSigma_i)\}_{i \in I})$ means that the QF entailment $\varPi \wedge \widetilde{\varSigma} \vdash \{\varPi_i \wedge \widetilde{\varSigma}\}_{i \in I}$ is valid (in fact, we can show their equivalence by induction on the definition of P). From this intuition, we sometimes call $\varSigma$ the left spatial formula, and also call $\{\varSigma_i\}_{i \in I}$ 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 Fig.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 $(\varPi', \varSigma')$ in the third argument of P is removed if $\varSigma'$ is not emp, since $\varPi' \wedge \varSigma'$ 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 $\neg(\mathrm{true} \wedge x < y)$.

The clause **($\mapsto\mapsto$)** handles the case where all the head atoms of Σ and $\{\Sigma_i\}_{i \in I}$ are the points-to predicate. 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, $P(3 < 4, 3 \mapsto 10 * 4 \mapsto 11, \{(\mathrm{true}, 3 \mapsto 10 * \mathrm{Arr}(4,4))\})$ becomes

$P(3 < 4 \wedge 3 < 4, 4 \mapsto 11, \{(\mathrm{true} \wedge 3 = 3 \wedge 4 = 4 \wedge 3 < 4, \mathrm{Arr}(4,4))\})$

This can be simplified to $P(3 < 4, 4 \mapsto 11, \{(3 = 3 \wedge 4 = 4 \wedge 3 < 4, \mathrm{Arr}(4,4))\})$, since $3 < 4 \wedge 3 < 4$ is logically equivalent to $3 < 4$ and $\mathrm{true} \wedge 3 = 3$ is logically equivalent to $3 = 3$. In the following examples, we implicitly use similar simplifications.

The clause **($\mapsto$Arr)** handles the case where the head atom of the left spatial formula is the points-to predicate and some right spatial formula Σ_i has the array predicate as its head atom. Then we split the array atomic formula into the points-to and the rest. We have three subcases according to the length of the head array. The first case is when the length of the array is 1: We replace the head array by a points-to atomic formula. The second case is when the length of the head array is greater than 1: We split the head array into the points-to predicate and the rest array. The last case is when the length of the head array is less than 1: We just remove (Π_i, Σ_i), since the array predicate is false. We note that this rule can be applied repeatedly until all head arrays 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, $P(\mathrm{true}, 4 \mapsto 11, \{(3 = 3 \wedge 10 = 10, \mathrm{Arr}(4,4))\})$ becomes

$P(3 < 4 \wedge 4 = 4, 4 \mapsto 11, \{(3 = 3 \wedge 10 = 10, 4 \mapsto 11)\})$
$\wedge P(3 < 4 \wedge 4 < 4, 4 \mapsto 11, \{(3 = 3 \wedge 10 = 10, 4 \mapsto 11 * \mathrm{Arr}(5,4))\})$
$\wedge P(3 < 4 \wedge 4 > 4, 4 \mapsto 11, \emptyset)$.

The clause **(Arr$\mapsto$)** handles the case where the head atom of the left spatial formula is array and there is a right spatial formula whose head atom is the points-to predicate. We have two subcases according to the length of the head array. The first case is when the length of the array is 1: The array is unfolded and it is replaced by a points-to atomic formula with a fresh variable z. The second case is the case where the length of the array is greater than 1: The array is split into the points-to predicate (with a fresh variable z') and the rest array. 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, $P(\mathrm{true}, \mathrm{Arr}(x, x), \{(\mathrm{true}, x \mapsto 10)\})$ becomes

$$
\begin{aligned}
&P(\mathrm{true} \land x > x, x \mapsto z * \mathrm{Arr}(x+1, x), \{(\mathrm{true}, x \mapsto 10)\})\\
&\quad \land P(\mathrm{true} \land x = x, x \mapsto z', \{(\mathrm{true}, x \mapsto 10)\}).
\end{aligned}
$$

The last clause **(ArrArr)** handles the case where all the head atoms in the left and right spatial formulas are arrays. We first find the head arrays with 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 to 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 of the left spatial formula has the shortest size and disappears by the operation. The second case is when the array of the left spatial formula has a longer size, it is split into two arrays, and the second part remains. We note that the measure is strictly reduced, since at least one shortest array is removed. For example, $P(\mathrm{true}, \mathrm{Arr}(3, 5), \{(\mathrm{true}, \mathrm{Arr}(3, 3) * \mathrm{Arr}(4, 5))\})$ becomes

$$
\begin{aligned}
&P(\mathrm{true} \land 2 < 0 \land 3 \leq 5, \mathrm{emp}, \{(\mathrm{true}, \mathrm{Arr}(6, 3) * \mathrm{Arr}(4, 5))\})\\
&\quad \land P(\mathrm{true} \land 2 = 0 \land 3 \leq 5, \mathrm{emp}, \{(\mathrm{true} \land 3 < 4, \mathrm{Arr}(4, 5))\})\\
&\quad \land P(\mathrm{true} \land 0 < 2 \land 0 = 0, \mathrm{Arr}(4, 5), \{(\mathrm{true}, \mathrm{Arr}(4, 5))\}).
\end{aligned}
$$

Note that the sizes of $\mathrm{Arr}(3, 5)$ and $\mathrm{Arr}(3, 3)$ are 3 and 1 respectively, and we have three cases for $2 < 0$, $2 = 0$, and $0 < 2$, by comparing them (actually comparing (them - 1)).

Example. The sorted entailment $(3 \mapsto 10 * 4 \mapsto 11)^{\sim} \vdash \mathrm{Arr}(3, 4)^{\sim}$ is translated by computing $P(\mathrm{true}, 3 \mapsto 10 * 4 \mapsto 11, \{(\mathrm{true}, \mathrm{Arr}(3, 4))\})$. We will see its calculation step by step. It first becomes

$$
\begin{aligned}
&P(3 = 4, 3 \mapsto 10 * 4 \mapsto 11, \{(\mathrm{true}, 3 \mapsto 10)\})\\
&\quad \land P(3 < 4, 3 \mapsto 10 * 4 \mapsto 11, \{(\mathrm{true}, 3 \mapsto 10 * \mathrm{Arr}(4, 4))\})\\
&\quad \land P(3 > 4, 3 \mapsto 10 * 4 \mapsto 11, \emptyset)
\end{aligned}
$$

by ($\mapsto$Arr). The first conjunct becomes $P(3 = 4 \land 3 < 4, 4 \mapsto 11, \{(3 = 3 \land 10 = 10, \mathrm{emp})\})$ by ($\mapsto\mapsto$), then it becomes $\neg(3 = 4 \land 3 < 4)$ by (NEmpEmp) and (empty). The third conjunct becomes $\neg(3 > 4 \land 3 < 4)$ by (empty). The second conjunct becomes $P(3 < 4, 4 \mapsto 11, \{(3 = 3 \land 10 = 10, \mathrm{Arr}(4, 4))\})$ by ($\mapsto\mapsto$), then it becomes

$$
\begin{aligned}
&P(3 < 4 \land 4 = 4, 4 \mapsto 11, \{(3 = 3 \land 10 = 10, 4 \mapsto 11)\})\\
&\quad \land P(3 < 4 \land 4 < 4, 4 \mapsto 11, \{(3 = 3 \land 10 = 10, 4 \mapsto 11 * \mathrm{Arr}(5, 4))\})\\
&\quad \land P(3 < 4 \land 4 > 4, 4 \mapsto 11, \emptyset)
\end{aligned}
$$

by ($\mapsto$Arr). Hence we have

$$
\begin{aligned}
&P(3 < 4 \land 4 = 4, \mathrm{emp}, \{(3 = 3 \land 10 = 10 \land 4 = 4 \land 11 = 11, \mathrm{emp})\})\\
&\quad \land P(3 < 4 \land 4 < 4, \mathrm{emp}, \{(3 = 3 \land 10 = 10 \land 4 = 4 \land 11 = 11, \mathrm{Arr}(5, 4))\})\\
&\quad \land \neg(3 < 4 \land 4 > 4)
\end{aligned}
$$

by ($\mapsto\mapsto$) and (empty). We note that the second one becomes $P(3 < 4 \land 4 < 4, \mathrm{emp}, \emptyset\})$ by (EmpNEmp). Thus we obtain

$$
(3 < 4 \land 4 = 4) \to (3 = 3 \land 10 = 10 \land 4 = 4 \land 11 = 11)
$$

$\wedge\neg(3 < 4 \wedge 4 < 4) \wedge\neg(3 < 4 \wedge 4 > 4)$
by (EmpEmp) and (empty). Finally we obtain $\neg(3 = 4 \wedge 3 < 4) \wedge (3 < 4 \wedge 4 = 4) \to (3 = 3 \wedge 10 = 10 \wedge 4 = 4 \wedge 11 = 11) \wedge \neg(3 < 4 \wedge 4 < 4) \wedge \neg(3 > 4 \wedge 3 < 4)$.

In our decision procedure the produced Presburger formula will be checked by an external SMT solver.

3.3 Decidability

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.

Theorem 1 (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*

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

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

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 finally 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 **SLAR** under the condition from the above theorem and the property of sorted entailments (stated in Lemma 1).

Corollary 1 (Decidability of Validity Checking of Entailments). *Validity checking of entailments $\Pi \wedge \Sigma \vdash \{\exists\overrightarrow{y_i}(\Pi_i \wedge \Sigma_i)\}_{i\in I}$ of **SLAR** 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. Our motivating example $\mathrm{Arr}(x, x) \vdash \exists y(x \mapsto y)$ satisfies the condition, and its validity is checked in the following way.

- It is decomposed into several sorted entailments: in this case, the decomposing process will produce the same entailment.

- Compute $P(\mathrm{true}, \mathrm{Arr}(x, x), \{(\mathrm{true}, x \mapsto y)\})$: It becomes
$P(x < x, x \mapsto z * \mathrm{Arr}(x + 1, x), \{(\mathrm{true}, x \mapsto y)\})$
$\wedge P(x = x, x \mapsto z, \{(\mathrm{true}, x \mapsto y)\})$,
then it becomes

$P(x < x \wedge x < x+1, \mathrm{Arr}(x+1, x), \{(x = x \wedge z = y, \mathrm{emp})\})$
$\wedge P(x = x, \mathrm{emp}, \{(x = x \wedge z = y, \mathrm{emp})\})$,
then it becomes $P(x < x \wedge x < x+1, \mathrm{Arr}(x+1, x), \emptyset) \wedge (x = x \to z = y)$. Finally we have $\neg(x < x \wedge x < x+1 \wedge x+1 \leq x) \wedge (x = x \to z = y)$.

Note that the resulting formula is logically equivalent to $z = y$.

- Check validity of the formula $\forall z \exists y P(\mathrm{true}, \mathrm{Arr}(x, x), \{(\mathrm{true}, x \mapsto y)\})$, which is equivalent to $\forall z \exists y (z = y)$ by the decision procedure of Presburger arithmetic. Finally the procedure answers that the given entailment is valid, since the Presburger formula is valid.

Remark 1. Our result can be extend to the class of entailments which are semantically equivalent to entailments that satisfy the condition. For example, the entailment

$\mathrm{Arr}(1, 10) \vdash \mathrm{Arr}(1, x) * \mathrm{Arr}(x+1, 10)$

does not satisfy the condition because the occurrences of the array atomic formulas on the succedent are not in the form required by the condition. However it can be decided, since it is equivalent to the following entailment which satisfy the condition:

$x = 1 + z \wedge x + w = 9 \wedge \mathrm{Arr}(1, 10) \vdash \mathrm{Arr}(1, 1+z) * \mathrm{Arr}(z+2, (z+2)+w)$.

In other words, our procedure can decide an entailment that the lengths of the array predicate on the succedent do not depend on the existential variables.

3.4 Discussions

Brotherston et al. [7] gave an independent condition for decidability of the entailment problem of the same symbolic-heap system. Their condition disallows existential variables in u for each points-to predicate $t \mapsto u$ in the succedent of an entailment. In order to clarify the difference between our condition and their condition, we consider the following entailments:

(i) $\mathrm{Arr}(x, x) \vdash \exists y (x \mapsto y)$
(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) $\mathrm{Arr}(1, 5) \vdash \exists y, y' (\mathrm{Arr}(1, 1+y) * 2+y \mapsto y' * \mathrm{Arr}(3+y, 5))$

As we observed above, (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 Section 2. 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. We explain them by example.

The triple
$\{\exists z(p \mapsto z)\}$ $\mathtt{y = x + 1; {*}p = y; x = 2;}$ $\{\exists x'(y = x' + 1 \land x = 2 \land p \mapsto y)\}$
is true and provable in our system, but it is not provable in their system, since $\exists x'(y = x' + 1 \land x = 2 \land p \mapsto y)$ is out of their assertions.

The triple
$\{x = 3 \land \mathrm{emp}\}$ $\mathtt{y = malloc(x); x = x + 1;}$ $\{\exists x'(x' = 3 \land x = x' + 1 \land \mathrm{Arr2}(y, x'))\}$
is true and provable in their system, but it is not provable in our system, since $\exists x'(x' = 3 \land x = x' + 1 \land \mathrm{Arr2}(y, x'))$ is out of our assertions.

Both kinds of triples are necessary for program verification in the real world, and in this sense both our system and their system have advantage and disadvantage. We can use both systems together to prove a single triple, by using one of them to check each necessary side condition, depending on the shape of the side condition.

4 Correctness of Decision Procedure

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

Lemma 1. *$s \models \varphi \to \bigvee_{i \in I} \phi_i$ is equivalent to*

$$s \models \widetilde{\varphi'} \to \bigvee\{\widetilde{\phi'} \mid i \in I, \phi' \in \mathrm{Perm}(\phi_i)\} \quad \textit{for all } \varphi' \in \mathrm{Perm}(\varphi)$$

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

This lemma shows that validity checking problem of a given entailment can be reduced to that of several 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. Notice 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

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

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 $\widetilde{S}$ for $\{\Pi_i \wedge \widetilde{\Sigma_i}\}_{i \in I}$. 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 2. (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* $s, h|_{\mathrm{Dom}(s,\Sigma)} \models \Pi \wedge \mathrm{Sorted}(\Sigma) \to \bigvee \widetilde{S}$.

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

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

(4) $\forall sh(s, h \models \Pi \wedge \widetilde{\Sigma} \to s, h \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S))$ *iff* $\models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$.

Proof. The claim (1) is shown by induction on the steps $=_{\mathrm{def}}$. The claim (2) can be obtained by using (1). The claim (3) is proved by induction on the steps of $=_{\mathrm{def}}$. The claim (4) is shown by using (3). □

Recall that our condition requires that lengths of arrays on the succedent does not depend on existential variables. We note that, under our condition, each t that appears as $t \mapsto u$ 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'.

Proof of Theorem 1 Let S be $\{(\Pi_i, \Sigma_i)\}_{i \in I}$. Then the left-hand side is equivalent to $\Pi \wedge \widetilde{\Sigma} \models \exists \overrightarrow{y} \bigvee \widetilde{S}$. Moreover, by Lemma 2 (2), it is equivalent to

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

By Lemma 2 (4), the right-hand side is equivalent to

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

Now we will show the equivalence of (a) and (b). Here we assume $[t_1], \ldots, [t_n]$ appear in $P'(\Pi, \Sigma, S)$ and $s \models t_1 < \ldots < t_n$, we let $\overrightarrow{z} = z_1, \ldots, z_n$. Notice that each $[t_j]$ does not contain any existential variable because of the condition. So we can obtain $P'(\Pi, \Sigma, S) = P(\Pi, \Sigma, S)[\overrightarrow{z} := [\overrightarrow{t}]]$. Hence (a) is obtained from (b) by taking z_i to be $[t_i]$ for $1 \leq i \leq n$.

We show the inverse direction. Assume (a). Fix s, h, and $\overrightarrow{d}$ for $\overrightarrow{z}$. Let s' be $s[\overrightarrow{z} := \overrightarrow{d}]$. Let h' be $h[s(\overrightarrow{t}) := \overrightarrow{d}]$. Then by (a), we have

$$s, h' \models \Pi \wedge \widetilde{\Sigma} \to s, h' \models \exists \overrightarrow{y} P'(\Pi, \Sigma, S).$$

We claim that $s, h' \models \exists \overrightarrow{y}.P'(\Pi, \Sigma, S)$ is equivalent to $s' \models \exists \overrightarrow{y}.P(\Pi, \Sigma, S)$. We also claim that $s, h' \models \Pi \wedge \widetilde{\Sigma}$ is equivalent to $s, h \models \Pi \wedge \widetilde{\Sigma}$, since each t_i appears as an address of an array predicate in Σ. Therefore we have (b). □

5 Improvement of Decision Procedure

Our decision procedure is not efficient because of the decomposition. The given unsorted entailment is decomposed into some sorted entailments with very large number of succedents. Recall that $|\Sigma|$ is the number of $*$ in Σ. Then an unsorted entailment $\Pi \wedge \Sigma \vdash \{\exists \overrightarrow{y_i}(\Pi_i \wedge \Sigma_i)\}_{i \in I}$ is decomposed into $|\Sigma|!$ -sorted entailments with $\sum_{i \in I} |\Sigma_i|!$ -succedents.

We currently adapt the following two ideas to improve this situation.

(U) Elimination of redundancy by using unsatisfiability checking We can easily observe that sorted entailments after decomposition often contain many redundant parts. For example, let σ_1, σ_2 and σ_3 be $1 \mapsto 10$, $2 \mapsto 20$ and $3 \mapsto 30$, respectively. An unsorted entailment $\sigma_2 * \sigma_1 \vdash \sigma_1 * \sigma_2, \sigma_3$ is decomposed into the following sorted entailments (after some simplification)

$$2 < 1 \wedge \sigma_2 * \sigma_1 \vdash 1 < 2 \wedge \sigma_1 * \sigma_2,\ 2 < 1 \wedge \sigma_2 * \sigma_1,\ \sigma_3$$
$$1 < 2 \wedge \sigma_1 * \sigma_2 \vdash 1 < 2 \wedge \sigma_1 * \sigma_2,\ 2 < 1 \wedge \sigma_2 * \sigma_1,\ \sigma_3$$

The first entailment is trivially valid, since its antecedent is unsatisfiable. So we can skip checking this entailment. The last two succedents of the second entailment are redundant, since they never satisfied with the antecedent. So we can drop them. More formally, we apply the following reduction rule:

$\varphi \vdash \overrightarrow{\phi_1}, \phi, \overrightarrow{\phi_2}$ is reduced to $\varphi \vdash \overrightarrow{\phi_1}, \overrightarrow{\phi_2}$ if $\varphi \wedge \phi$ is unsatisfiable.

(F) Frame elimination by using the invertible frame rule Our second improvement is to reduce the number of separating conjunctions in the given unsorted entailment (before the decomposition process). This improvement is effective since reducing the number of separating conjunctions reduces the number of the succedents after decomposition.

In order to reduce separating conjunctions, we use the frame rule: $\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$ implies $\Pi \wedge \sigma * \Sigma \vdash \Pi' \wedge \sigma * \Sigma'$, where σ is $t \mapsto u$ or $\mathrm{Arr}(t, u)$. However the frame rule of this form is not invertible, that is, the inverse direction of the implication does not generally hold. In our setting, since we have inequalities, we can define $\mathrm{Disj}(\sigma, \Sigma)$ as a pure formula, which means the memory cells used by σ and Σ are disjoint. Hence we have the following **invertible frame rule**:

$$\Pi \wedge \sigma * \Sigma \vdash \{\exists \overrightarrow{y_i}(\Pi_i \wedge \sigma * \Sigma_i)\}_{i \in I} \iff \Pi \wedge \mathrm{Disj}(\sigma, \Sigma) \wedge \Sigma \vdash \{\exists \overrightarrow{y_i}(\Pi_i \wedge \Sigma_i)\}_{i \in I}$$

We consider this invertible frame rule as a rewriting rule from the left-hand side to the right-hand side. By applying this rewriting rule as much as possible, the given entailment can be rewritten to entailments with a smaller number of $*$.

This procedure has a great effect on efficiency improvement of our decision procedure, since reducing the number of $*$ highly contributes to reduce the sizes and the number of sorted entailments that are generated by the decomposition.

6 Implementation and Experiments

This section explains our tool **SLar**, which is the implementation of our decision procedure.

6.1 Entailment Checker SLar

The behavior of **SLar** is based on the decision procedure discussed in the previous sections. It consists of the following three parts:

(1) The optimizing part, which reduces the size of a given entailment by the invertible frame rule (**F**). It also checks satisfiability of the antecedent of the entailment. **SLar** immediately answers "valid" if the antecedent is unsatisfiable.

(2) The decomposing part, which decomposes the reduced entailment into several sorted entailments, that is, the given unsorted entailment $\varphi \vdash \{\phi_i\}_{i\in I}$ is decomposed into sorted entailments $\widetilde{\varphi'} \vdash \{\widetilde{\phi'} \mid i \in I, \phi' \in Perm(\phi_i)\}$, where $\varphi' \in Perm(\varphi)$. The correctness of this part is guaranteed by Lemma 1. After decomposition, redundant parts are reduced by the unsatisfiability checking (**U**).

(3) The translating part, which translates sorted entailments into Presburger formulas according to the translation P given in the section 3. The theoretical correctness of this part is guaranteed by Theorem 1.

(4) The checking part, in which the SMT-solver **Z3** [20] is invoked to check validity of the generated Presburger formula.

The current version of **SLar** is written in about 3900 lines of OCaml codes (360 lines for the decomposing part and some optimization, 1900 lines for the translating part, and 1200 lines for the checking part).

The improvements **U** and **F** mentioned above are optional, that is, **SLar** has options which changes its behavior with (or without) them.

6.2 Experiments and Evaluation

This subsection reports on the performance of **SLar**. As far as we know, there is no suitable benchmark of entailments with arrays. So we automatically generated 120 entailments by using our analyzer of the C language. Each entailment has a single conclusion and is of small size (there are 1 or 2 separating conjunctions on each side). We call this set of entailments Base. 40 out of 120 are entailments with only the points-to predicate (called the group Pto). Another 40 are entailments with only arrays (called the group Array). The rest 40 entailments contain both the points-to and array predicates (called the group Mix). In each case, half of the entailments are valid.

Then we automatically produced the following sets of entailments from Base. All experimental files can be found in the web [3].

- SingleFrame-n $(n = 2, 3)$: A set of single-conclusion entailments produced by putting a frame of size n to the entailments of Base. The frames are chosen to keeping the grouping, that is, frames of the points-to predicate are used for Pto, frames of arrays are used for Array, and random frames are used for Mix;

- SingleNFrame-n $(n = 2, 3)$: A set of single-conclusion entailments produced by putting different spatial predicates of length n to each side of the entailments of Base. The spatial predicates are chosen to keeping the grouping;

- Multi : A set of multi-conclusion entailments with at most 3 disjuncts. They are produced by adding extra conclusions to the entailments of Base.

In order to evaluate the effect of our improvement discussed in the previous subsection, we provided the options that change whether **SLar** uses the unsatisfiability checking (**U**) and the invertible frame rule (**F**).

For each categories Base, SingleFrame-n, SingleNFrame-n and Multi, we executed our tool with (or without) the options of **U** and **F**, and recorded its execution time (with timeout 300 sec). Our PC environment is MacOS X machine with Intel Core i5 3.1GHz processor and 8GB memory.

The results summarized in the tables of Fig. 2, where $\mathbf{U}+\mathbf{F}$ means that the both options of the invertible frame rule and of the unsatisfiability checking are turned on. **U** and **F** means that only corresponding option is used. None means that none of them are used. Each table shows the number of entailments (out of 120) whose solved time satisfy the time condition displayed on the top of the table. For example, the table of Base shows that 111 entailments are solved in less than 0.1 sec by using both options ($\mathbf{U}+\mathbf{F}$).

For Base, almost the entailments (111 out of 120) are answered within 0.1 sec if both options are used. The rest 9 entailments are solved within 0.2 sec. All are answered within 2.1 sec without the acceleration options. Comparing $\mathbf{U}+\mathbf{F}$ and None, our tool becomes up to 15 times, average 4 times faster.

The categories SingleFrame-n $(n = 2, 3)$ show the effect of using the invertible frame rule, because entailments of these categories are in the form for which the invertible frame rule can be applied. In SingleFrame-3, the cases of $\mathbf{U}+\mathbf{F}$ are about 10 times faster on average than that of **U** (without the invertible frame rule). Sometimes the tool becomes more than 200 times faster!

The categories SingleNFrame-n $(n = 2, 3)$ are intended to limit use of the invertible frame rule. In the case of $\mathbf{U}+\mathbf{F}$ of SingleNFrame-3, almost entailments (102 out of 120) are solved within 1 sec.

Checking the category Multi is mainly accelerated by the unsatisfiability checking. This is because unsatisfiable disjuncts of the succedent are eliminated in early stage of the decision procedure. The cases of $\mathbf{U}+\mathbf{F}$ are about 8 times faster on average than that of **F** (without the unsatisfiability checking).

[3] `https://github.com/DaisukeKimura/slar`

Base					
					time out
	< 0.1s	< 1s	< 10s	< 300s	(300s)
U + F	111	120	120	120	0
U	103	120	120	120	0
F	106	120	120	120	0
None	82	116	120	120	0

Multi					
					time out
	< 0.1s	< 1s	< 10s	< 300s	(300s)
U + F	57	108	120	120	0
U	45	106	120	120	0
F	24	101	115	120	0
None	23	101	111	118	2

SingleFrame-2					
					time out
	< 0.1s	< 1s	< 10s	< 300s	(300s)
U + F	101	118	120	120	0
U	78	119	120	120	0
F	97	116	120	120	0
None	31	68	88	120	0

SingleFrame-3					
					time out
	< 0.1s	< 1s	< 10s	< 300s	(300s)
U + F	81	107	118	120	0
U	37	95	120	120	0
F	80	105	116	119	1
None	14	66	84	106	14

SingleNFrame-2					
					time out
	< 0.1s	< 1s	< 10s	< 300s	(300s)
U + F	79	119	120	120	0
U	71	119	120	120	0
F	81	117	120	120	0
None	31	80	111	119	1

SingleNFrame-3					
					time out
	< 0.1s	< 1s	< 10s	< 300s	(300s)
U + F	34	102	119	120	0
U	33	87	118	120	0
F	39	97	117	120	0
None	14	64	99	116	4

Fig. 2. Experimental Results

In the case of $\mathbf{U} + \mathbf{F}$ of each category, the average time of the group Pto is less than 0.3 sec, the average time of Array is less than 0.6 sec, the average time of Mix is less than 0.7 sec. The average time about invalid entailments is faster than that of valid entailments. It is because that our procedure immediately answers "invalid" when it finds an invalid decomposed entailment.

From the results of Base, our tool works very quickly for small entailments (the number of separating conjunctions on each side is less than or equal to 2). However, as we expected, it becomes slower for entailments with greater sizes. Hence it is quite important to reduce the sizes of entailments. The results of SingleFrame-3 and SingleNFrame-3 shows that the invertible frame rule greatly contributes for reducing sizes and improving efficiency. The effect of our improvement remarkably appears when sizes of entailments are large. In this experiment the effect worked well in the case of SingleNFrame-3 (Fig. 2).

7 Conclusion and Future Work

In this paper we investigated the separation logic with arrays, and showed decidability of its entailment problem under the size condition. We also implemented our decision procedure that checks validity of entailments. The performance of our algorithm deeply depends on the number of separating conjunctions. So it is quite important to reduce their number. From the results of the experiments, we are convinced that the invertible frame rule and the undecidability checking work well and contribute for improving the entailment checking tool.

Currently we have put a condition for proving correctness of the decision procedure. However our tool also seems to work well for entailments that are

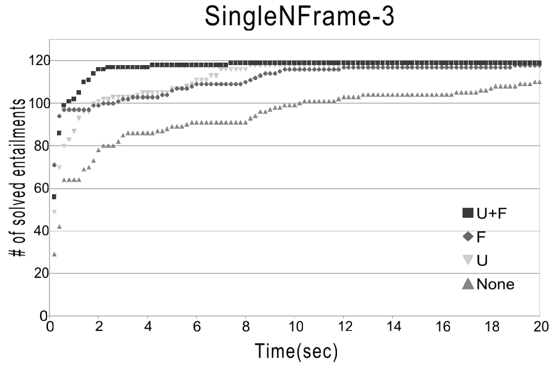

Fig. 3. Experimental Result of SingleNFrame-3

out of the condition. So we conjecture that correctness of the decision procedure also can be shown without the condition.

Our algorithm is still inefficient since the performance of our tool slows sharply as the number of separating conjunction increases. One possible way is to split the given entailment into some smaller entailments, that is, split $\mathit{\Pi}_1 \wedge \mathit{\Sigma}_1 * \mathit{\Sigma}'_1 \vdash \mathit{\Pi}_2 \wedge \mathit{\Sigma}_2 * \mathit{\Sigma}'_2$ into $\mathit{\Pi}_1 \wedge \mathit{\Sigma}_1 \vdash \mathit{\Pi}_2 \wedge \mathit{\Sigma}_2$ and $\mathit{\Pi}_1 \wedge \mathit{\Sigma}'_1 \vdash \mathit{\Pi}_2 \wedge \mathit{\Sigma}'_2$. Of course the difficult point of this approach is to find the correct splitting point. However the memory model of the C language can be considered as the set of pairs of the form (p, n), where p is an identifier that indicates a domain and n is an offset integer [18]. If we carefully translate C codes into formulas of separation logic without losing the information of domains, it would be reasonable to choose a boundary of two different domains as a splitting point.

References

1. T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine, Foundations for Decision Problems in Separation Logic with General Inductive Predicates, In: Proceedings of FoSSaCS 2014, *LNCS* 8412 (2014) 411–425.
2. J. Berdine, C. Calcagno, P. W. O'Hearn, A Decidable Fragment of Separation Logic, In: Proceedings of FSTTCS 2004, *LNCS* 3328 (2004) 97–109.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn, Symbolic Execution with Separation Logic, In: Proceedings of APLAS 2005, *LNCS* 3780 (2005) 52–68.
4. A. Bouajjani, C. Drăgoi, C. Enea, M. Sighireanu, A Logic-Based Framework for Reasoning About Composite Data Structures, In: Proceedings of CONCUR 2009, 178–195.
5. A. R. Bradley, Z. Manna, H. B. Sipma, Whats Decidable About Arrays?, In: Proceedings of VMCAI 2006, *LNCS* 3855 (2006) 427–442.
6. J. Brotherston, C. Fuhs, N. Gorogiannis, and J.A. Navarro Pérez, A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates, In: Proceedings of CSL-LICS, 2014, Article No. 25.

7. J. Brotherston, N. Gorogiannis, and M. Kanovich, Biabduction (and Related Problems) in Array Separation Logic, In: Proceedings of CADE-26, *LNAI* 10395 (2017) 472–490.
8. C. Calcagno, D. Distefano, P. W. O'Hearn, H.Yang, Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic, In: Proceedings of SAS 2006, *LNCS* 4134 (2006) 182–203.
9. C. Calcagno, D. Distefano, P. W. O'Hearn, H.Yang, Compositional Shape Analysis by Means of Bi-Abduction, In: Journal of ACM, Vol.58 (6), 2011, 1–66.
10. B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, J. Worrell, Tractable reasoning in a fragment of Separation Logic, In: Proceedings of CONCUR'11, *LNCS* 6901 (2011) 235–249.
11. C. Enea, V. Saveluc, M. Sighireanu, Compositional invariant checking for overlaid and nested linked lists, In: Proceeding of ESOP 2013, *LNCS* 7792 (2013) 129–148.
12. C. Enea, O. Lengál, M. Sighireanu, T. Vojnar, Compositional Entailment Checking for a Fragment of Separation Logic, In: Proceedings of APLAS 2014, *LNCS* 8858 (2014) 314–333.
13. R. Iosif, A. Rogalewicz, and J. Simacek, The Tree Width of Separation Logic with Recursive Definitions, In: Proceedings of CADE-24, *LNCS* 7898 (2013) 21–38.
14. R. Iosif, A. Rogalewicz, and T. Vojnar, Deciding Entailments in Inductive Separation Logic with Tree Automata, In: Proceedings of ATVA2014, *LNCS* 8837 (2014) 201–218.
15. S. Lahiri, S. Qadeer, Back to the Future: Revisiting Precise Program Verification Using SMT Solvers, In: Proceedings of POPL 2008, 171–182.
16. R. Piskac, T. Wies, D. Zufferey, Automating Separation Logic Using SMT, In: Proceedings of CAV 2013, *LNCS* 8044 (2013) 773–789.
17. J.C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, In: *Proceedings of Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS2002)* (2002) 55–74.
18. T. Sekiguchi, A Practical Pointer Analysis for C Language, In: Computer Software, Vol.21, No.6 (2004) 456–471.
19. M. Tatsuta and D. Kimura, Separation Logic with Monadic Inductive Definitions and Implicit Existentials, In: Proceedings of APLAS 2015, *LNCS* 9458 (2015) 69–89.
20. L. de Moura and N. Bjørner, Z3: An Efficient SMT Solver, In: Proceedings of TACAS 2008, 337–340.

Appendix

Proof of Lemma 2

Lemma 2. (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* $s, h|_{\mathrm{Dom}(s,\Sigma)} \models \Pi \wedge \mathrm{Sorted}(\Sigma) \to \bigvee \widetilde{S}$.

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

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

(4) $\forall sh(s, h \models \Pi \wedge \widetilde{\Sigma} \to s, h \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S))$ *iff* $\models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\Pi, \Sigma, S)$.

Proof of Lemma 2 (1). This is shown by induction on the steps $=_{\mathrm{def}}$. Consider cases according to the definition of P'.

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

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

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

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

iff

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

The only-if part. Assume (c) and the antecedent of (d). Then the antecedent of (c) holds, since they are equivalent. Then the succedent of (c) is true for s, h_1. Hence the succedent of (d) is true for s, h_2.

The if part. Assume (d) and the antecedent of (c). Then the antecedent of (d) holds, since they are equivalent. Then the succedent of (d) is true for s, h_2. Hence the succedent of (c) is true for s, h_1.

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

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

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

$$s, h_3 \models \Pi \wedge \mathrm{Sorted}(\mathrm{Arr}(t, t') * \Sigma) \to \bigvee \widetilde{S} \tag{e}$$

is equivalent to the conjunction of the following two clauses:

$$s, h_4 \models \Pi \wedge t' = t \wedge t < \Sigma \wedge \mathrm{Sorted}(\Sigma) \to \bigvee \widetilde{S} \quad \text{and} \tag{f}$$

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

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

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

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

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

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

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

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

$$s, h' \models \Pi \wedge \mathrm{Sorted}_L \to \Pi_i \wedge (\mathrm{Arr}(t_i, t'_i) * \Sigma_i)^{\sim} \vee \bigvee \widetilde{S}$$

is equivalent to the conjunction of the following three clauses:

$s, h' \models \Pi \wedge t'_i = t_i \wedge \mathrm{Sorted}_L \to \Pi_i \wedge (t_i \mapsto u * \Sigma_i)^{\sim} \vee \bigvee \widetilde{S}$

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

$s, h' \models \Pi \wedge t'_i < t_i \wedge \mathrm{Sorted}_L \to \bigvee \widetilde{S}$,

where Sorted_L is an abbreviation of $\mathrm{Sorted}(t \mapsto u * \Sigma)$.

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

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

for all $I' \subseteq I$ and

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

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

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

is equivalent to the conjunction of the following

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

for any $I' \subseteq I$, and

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

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 (i) with respect to I' has the case condition. All conjuncts of (i) other than this conjunct and conjuncts of (j) 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: Assume (h) and the antecedent of the conjunct. Then the antecedent of (h) holds, since they are equivalent. Then the succedent of (h) is true for s, h_6. Hence the succedent of the conjunct is true for s, h_7.

The if part: Assume (i) and the antecedent of (h). Then the antecedent of the conjunct holds, since they are equivalent. Then the succedent of the conjunct is true for s, h_7. Hence the succedent of (h) is true for s, h_6.

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

Proof of Lemma 2 (3). It is shown by induction on the steps of $=_{\mathrm{def}}$.

Proof of Lemma 2 (4). We note that $\forall sh(s, h \models \mathit{\Pi} \wedge \widetilde{\mathit{\Sigma}} \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\mathit{\Pi}, \mathit{\Sigma}, S))$ is equivalent to $\forall s(\exists h(s, h \models \mathit{\Pi} \wedge \widetilde{\mathit{\Sigma}}) \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\mathit{\Pi}, \mathit{\Sigma}, S))$. Moreover it is equivalent to $\forall s(s \models \mathit{\Pi} \wedge \mathrm{Sorted}(\mathit{\Sigma}) \to s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\mathit{\Pi}, \mathit{\Sigma}, S))$. By using (3), it is equivalent to $\forall s(s \models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\mathit{\Pi}, \mathit{\Sigma}, S))$, namely, $\models \forall \overrightarrow{z} \exists \overrightarrow{y} P(\mathit{\Pi}, \mathit{\Sigma}, S)$. □