Separation Logic with One Quantified Variable*

S. Demri[1] and D. Galmiche[2] and D. Larchey-Wendling[2] and D. Mery[2]

[1]New York University & CNRS [2]LORIA – CNRS – University of Lorraine

Abstract. We investigate first-order separation logic with one record field restricted to a unique quantified variable (1SL1). Undecidability is known when the number of quantified variables is unbounded and the satisfiability problem is PSPACE-complete for the propositional fragment. We show that the satisfiability problem for 1SL1 is PSPACE-complete and we characterize its expressive power by showing that every formula is equivalent to a Boolean combination of atomic properties. This contributes to our understanding of fragments of first-order separation logic that can specify properties about the memory heap of programs with singly-linked lists. When the number of program variables is fixed, the complexity drops to polynomial time. All the fragments we consider contain the magic wand operator and first-order quantification over a single variable.

1 Introduction

Separation Logic for Verifying Programs with Pointers. Separation logic [20] is a well-known logic for analysing programs with pointers stemming from BI logic [14]. Such programs have specific errors to be detected and separation logic is used as an assertion language for Hoare-like proof systems [20] that are dedicated to verify programs manipulating heaps. Any procedure mechanizing the proof search requires subroutines that check the satisfiability or the validity of formulæ from the assertion language. That is why, characterizing the computational complexity of separation logic and its fragments and designing optimal decision procedures remain essential tasks. Separation logic contains a structural separating connective and its adjoint (the separating implication, also known as the magic wand). The main concern of the paper is to study a non-trivial fragment of first-order separation logic with one record field as far as expressive power, decidability and complexity are concerned. Herein, the models of separation logic are pairs made of a variable valuation (store) and a partial function with finite domain (heap), also known as memory states.

Decidability and Complexity. The complexity of satisfiability and model-checking problems for separation logic fragments have been quite studied [6, 20, 7] (see also new decidability results in [13] or undecidability results in [5, 16] in an alternative setting). Separation logic is equivalent to a Boolean propositional logic [17, 18] if first-order quantifiers are disabled. Separation logic without first-order quantifiers is decidable, but it becomes undecidable with first-order quantifiers [6]. For instance, model-checking and satisfiability for propositional

* Work partially supported by the ANR grant DynRes (project no. ANR-11-BS02-011) and by the EU Seventh Framework Programme under grant agreement No. PIOF-GA-2011-301166 (DATAVERIF).

separation logic are PSPACE-complete problems [6]. Decidable fragments with first-order quantifiers can be found in [11, 4]. However, these known results crucially rely on the memory model addressing cells with two record fields (undecidability of 2SL in [6] is by reduction to the first-order theory of a finite *binary* relation). In order to study decidability or complexity issues for separation logic, two tracks have been observed in the literature. There is the verification approach with decision procedures for fragments of practical use, see e.g. [2, 7, 12]. Alternatively, fragments, extensions or variants of separation logic are considered from a logical viewpoint, see e.g. [6, 5, 16].

Our Contributions. In this paper, we study first-order separation logic with one quantified variable, with an unbounded number of program variables and with one record field (herein called 1SL1). We introduce *test formulæ* that state simple properties about the memory states and we show that every formula in 1SL1 is equivalent to a Boolean combination of test formulæ, extending what was done in [18, 3] for the propositional case. For instance, separating connectives can be eliminated in a controlled way as well as first-order quantification over the single variable. In that way, we show a quantifier elimination property similar to the one for Presburger arithmetic. This result extends previous ones on propositional separation logic [17, 18, 3] and this is the first time that this approach is extended to a first-order version of separation logic with the magic wand operator. However, it is the best we can hope for since 1SL with two quantified variables and no program variables (1SL2) has been recently shown undecidable in [9]. Of course, other extensions of 1SL1 could be considered, for instance to add a bit of arithmetical constraints, but herein we focus on 1SL1 that is theoretically nicely designed, even though it is still unclear how much 1SL1 is useful for formal verification. We also establish that the satisfiability problem for Boolean combinations of test formulæ is NP-complete thanks to a saturation algorithm for the theory of memory states with test formulæ, paving the way to use SMT solvers to decide 1SL1 (see e.g. the use of such solvers in [19]).

Even though Boolean combinations of test formulæ and 1SL1 have identical expressive power, we obtain PSPACE-completeness for model-checking and satisfiability in 1SL1. The conciseness of 1SL1 explains the difference between these two complexities. PSPACE-completeness is still a relatively low complexity but this result can be extended with more than one record field (but still with one quantified variable). This is the best we can hope for with one quantified variable and with the magic wand, that is notoriously known to easily increase complexity. We also show that 1SL1 with a bounded number of program variables has a satisfiability problem that can be solved in polynomial time.

Omitted proofs can be found in the technical report [10].

2 Preliminaries

2.1 First-order separation logic with one selector 1SL

Let $\mathtt{PVAR} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$ be a countably infinite set of *program variables* and $\mathtt{FVAR} = \{\mathtt{u}_1, \mathtt{u}_2, \ldots\}$ be a countably infinite set of *quantified variables*. A *mem-*

ory state (also called a *model*) is a pair (s, h) such that s is a variable valuation of the form $s : \texttt{PVAR} \to \mathbb{N}$ (the *store*) and h is a partial function $h : \mathbb{N} \rightharpoonup \mathbb{N}$ with finite domain (the *heap*) and we write $\mathrm{dom}(h)$ to denote its *domain* and $\mathrm{ran}(h)$ to denote its *range*. Two heaps h_1 and h_2 are said to be *disjoint*, noted $h_1 \perp h_2$, if their domains are disjoint; when this holds, we write $h_1 \uplus h_2$ to denote the heap corresponding to the disjoint union of the graphs of h_1 and h_2, hence $\mathrm{dom}(h_1 \uplus h_2) = \mathrm{dom}(h_1) \uplus \mathrm{dom}(h_2)$. When the domains of h_1 and h_2 are not disjoint, the composition $h_1 \uplus h_2$ is not defined even if h_1 and h_2 have the same values on $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2)$.

Formulæ of 1SL are built from *expressions* of the form $e ::= \texttt{x} \mid \texttt{u}$ where $\texttt{x} \in \texttt{PVAR}$ and $\texttt{u} \in \texttt{FVAR}$, and *atomic formulæ* of the form $\pi ::= e = e' \mid e \hookrightarrow e' \mid \texttt{emp}$. *Formulæ* are defined by the grammar $\mathcal{A} ::= \bot \mid \pi \mid \mathcal{A} \wedge \mathcal{B} \mid \neg\mathcal{A} \mid \mathcal{A} * \mathcal{B} \mid \mathcal{A} \mathrel{-\!\!*} \mathcal{B} \mid \exists\, \texttt{u}\, \mathcal{A}$, where $\texttt{u} \in \texttt{FVAR}$. The connective $*$ is *separating conjunction* and $\mathrel{-\!\!*}$ is *separating implication*, usually called the *magic wand*. The *size* of a formula $\mathcal{A}$, written $|\mathcal{A}|$, is defined as the number of symbols required to write it. An *assignment* is a map $\mathfrak{f} : \texttt{FVAR} \to \mathbb{N}$. The satisfaction relation $\models$ is parameterized by assignments (clauses for Boolean connectives are omitted):

- $(s, h) \models_{\mathfrak{f}} \texttt{emp}$ iff $\mathrm{dom}(h) = \varnothing$.
- $(s, h) \models_{\mathfrak{f}} e = e'$ iff $[e] = [e']$, with $[\texttt{x}] \stackrel{\text{def}}{=} s(\texttt{x})$ and $[\texttt{u}] \stackrel{\text{def}}{=} \mathfrak{f}(\texttt{u})$.
- $(s, h) \models_{\mathfrak{f}} e \hookrightarrow e'$ iff $[e] \in \mathrm{dom}(h)$ and $h([e]) = [e']$.
- $(s, h) \models_{\mathfrak{f}} \mathcal{A}_1 * \mathcal{A}_2$ iff $h = h_1 \uplus h_2$, $(s, h_1) \models_{\mathfrak{f}} \mathcal{A}_1$, $(s, h_2) \models_{\mathfrak{f}} \mathcal{A}_2$ for some h_1, h_2.
- $(s, h) \models_{\mathfrak{f}} \mathcal{A}_1 \mathrel{-\!\!*} \mathcal{A}_2$ iff for all h', if $h \perp h'$ & $(s, h') \models_{\mathfrak{f}} \mathcal{A}_1$ then $(s, h \uplus h') \models_{\mathfrak{f}} \mathcal{A}_2$.
- $(s, h) \models_{\mathfrak{f}} \exists\, \texttt{u}\, \mathcal{A}$ iff there is $l \in \mathbb{N}$ such that $(s, h) \models_{\mathfrak{f}[\texttt{u} \mapsto l]} \mathcal{A}$ where $\mathfrak{f}[\texttt{u} \mapsto l]$ is the assignment equal to $\mathfrak{f}$ except that $\texttt{u}$ takes the value l.

Whereas '$\exists$' is clearly a first-order quantifier, the connectives $*$ and $\mathrel{-\!\!*}$ are known to be second-order quantifiers. In the paper, we show how to eliminate the three connectives when only one quantified variable is used.

We write 1SL0 to denote the propositional fragment of 1SL, i.e. without any occurrence of a variable from $\texttt{FVAR}$. Similarly, we write 1SL1 to denote the fragment of 1SL restricted to a single quantified variable, say $\texttt{u}$. In that case, the satisfaction relation can be denoted by $\models_l$ where l is understood as the value for the variable under the assignment.

Given $q \geqslant 1$ and $\mathcal{A}$ in 1SL built over $\texttt{x}_1, \ldots, \texttt{x}_q$, we define its *memory threshold* $\texttt{th}(q, \mathcal{A})$: $\texttt{th}(q, \mathcal{A}) \stackrel{\text{def}}{=} 1$ for atomic formula $\mathcal{A}$; $\texttt{th}(q, \mathcal{A}_1 \wedge \mathcal{A}_2) \stackrel{\text{def}}{=} \texttt{max}(\texttt{th}(q, \mathcal{A}_1), \texttt{th}(q, \mathcal{A}_2))$; $\texttt{th}(q, \neg\mathcal{A}_1) \stackrel{\text{def}}{=} \texttt{th}(q, \mathcal{A}_1)$; $\texttt{th}(q, \exists\, \texttt{u}\, \mathcal{A}_1) \stackrel{\text{def}}{=} \texttt{th}(q, \mathcal{A}_1)$; $\texttt{th}(q, \mathcal{A}_1 * \mathcal{A}_2) \stackrel{\text{def}}{=} \texttt{th}(q, \mathcal{A}_1) + \texttt{th}(q, \mathcal{A}_2)$; $\texttt{th}(q, \mathcal{A}_1 \mathrel{-\!\!*} \mathcal{A}_2) \stackrel{\text{def}}{=} q + \texttt{max}(\texttt{th}(q, \mathcal{A}_1), \texttt{th}(q, \mathcal{A}_2))$.

Lemma 1. *Given $q \geqslant 1$ and a formula $\mathcal{A}$ in 1SL, we have $1 \leqslant \texttt{th}(q, \mathcal{A}) \leqslant q \times |\mathcal{A}|$.*

Let $\mathcal{L}$ be a logic among 1SL, 1SL1 and 1SL0. As usual, the *satisfiability problem* for $\mathcal{L}$ takes as input a formula $\mathcal{A}$ from $\mathcal{L}$ and asks whether there is a memory state (s, h) and an assignment $\mathfrak{f}$ such that $(s, h) \models_{\mathfrak{f}} \mathcal{A}$. The *model-checking problem* for $\mathcal{L}$ takes as input a formula $\mathcal{A}$ from $\mathcal{L}$, a memory state (s, h) and an assignment $\mathfrak{f}$ for free variables from $\mathcal{A}$ and asks whether $(s, h) \models_{\mathfrak{f}} \mathcal{A}$. When checking the satisfiability status of a formula $\mathcal{A}$ in 1SL1, we assume that its program variables

are contained in $\{\mathtt{x}_1, \ldots, \mathtt{x}_q\}$ for some $q \geqslant 1$ and the quantified variable is $\mathtt{u}$. So, $\mathtt{PVAR}$ is unbounded but as usual, when dealing with a specific formula, the set of program variables is finite.

Theorem 2. *[6, 4, 9] Satisfiability and model-checking problems for 1SL0 are* PSPACE-*complete, satisfiability problem for 1SL is undecidable, even restricted to two variables.*

2.2 A bunch of properties stated in 1SL1

The logic 1SL1 allows to express different types of properties on memory states. The examples below indeed illustrate the expressivity of 1SL1, and in the paper we characterize precisely what can be expressed in 1SL1.

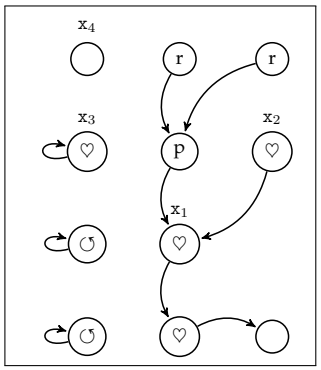
- The domain of the heap has at least k elements: $\neg\mathtt{emp} * \cdots * \neg\mathtt{emp}$ (k times).
- The variable $\mathtt{x}_i$ is allocated in the heap: $\mathtt{alloc}(\mathtt{x}_i) \stackrel{\text{def}}{=} (\mathtt{x}_i \hookrightarrow \mathtt{x}_i) \mathrel{-\!\!*} \bot$.
- The variable $\mathtt{x}_i$ points to a location that is a loop: $\mathtt{toloop}(\mathtt{x}_i) \stackrel{\text{def}}{=} \exists\, \mathtt{u}\, (\mathtt{x}_i \hookrightarrow \mathtt{u} \wedge \mathtt{u} \hookrightarrow \mathtt{u})$; the variable $\mathtt{x}_i$ points to a location that is allocated: $\mathtt{toalloc}(\mathtt{x}_i) \stackrel{\text{def}}{=} \exists\, \mathtt{u}\, (\mathtt{x}_i \hookrightarrow \mathtt{u} \wedge \mathtt{alloc}(\mathtt{u}))$.
- Variables $\mathtt{x}_i$ and $\mathtt{x}_j$ point to a shared location: $\mathtt{conv}(\mathtt{x}_i, \mathtt{x}_j) \stackrel{\text{def}}{=} \exists\, \mathtt{u}\, (\mathtt{x}_i \hookrightarrow \mathtt{u} \wedge \mathtt{x}_j \hookrightarrow \mathtt{u})$; there is a location between $\mathtt{x}_i$ and $\mathtt{x}_j$: $\mathtt{inbetween}(\mathtt{x}_i, \mathtt{x}_j) \stackrel{\text{def}}{=} \exists \mathtt{u}\, (\mathtt{x}_i \hookrightarrow \mathtt{u} \wedge \mathtt{u} \hookrightarrow \mathtt{x}_j)$.
- Location interpreted by $\mathtt{x}_i$ has exactly one predecessor can be expressed in 1SL1: $(\exists\, \mathtt{u}\, \mathtt{u} \hookrightarrow \mathtt{x}_i) \wedge \neg(\exists\, \mathtt{u}\, \mathtt{u} \hookrightarrow \mathtt{x}_i * \exists\, \mathtt{u}\, \mathtt{u} \hookrightarrow \mathtt{x}_i)$.
- Heap has at least 3 self-loops: $(\exists\, \mathtt{u}\, \mathtt{u} \hookrightarrow \mathtt{u}) * (\exists\, \mathtt{u}\, \mathtt{u} \hookrightarrow \mathtt{u}) * (\exists\, \mathtt{u}\, \mathtt{u} \hookrightarrow \mathtt{u})$.

2.3 At the heart of domain partitions

Given (s, h) and a finite set $\mathcal{V} = \{\mathtt{x}_1, \ldots, \mathtt{x}_q\} \subseteq \mathtt{PVAR}$, we introduce two partitions of $\mathrm{dom}(h)$ depending on $\mathcal{V}$: one partition takes care of self-loops and predecessors of interpretations of program variables, the other one takes care of locations closely related to interpretations of program variables (to be defined below). This allows us to decompose the domain of heaps in such a way that we can easily identify the properties that can be indeed expressed in 1SL1 restricted to the variables in $\mathcal{V}$. We introduce a first partition of the domain of h by distinguishing the self-loops and the predecessors of variable interpretations on the one hand, and the remaining locations in the domain on the other hand: $\mathrm{pred}(s, h) \stackrel{\text{def}}{=} \bigcup_i \mathrm{pred}(s, h, i)$ with $\mathrm{pred}(s, h, i) \stackrel{\text{def}}{=} \{l' : h(l') = s(\mathtt{x}_i)\}$ for every $i \in [1, q]$; $\mathrm{loop}(s, h) \stackrel{\text{def}}{=} \{l \in \mathrm{dom}(h) : h(l) = l\}$; $\mathrm{rem}(s, h) \stackrel{\text{def}}{=} \mathrm{dom}(h) \backslash (\mathrm{pred}(s, h) \cup \mathrm{loop}(s, h))$. So, obviously $\mathrm{dom}(h) = \mathrm{rem}(s, h) \uplus (\mathrm{pred}(s, h) \cup \mathrm{loop}(s, h))$. The sets $\mathrm{pred}(s, h)$ and $\mathrm{loop}(s, h)$ are not necessarily disjoint. As a consequence of h being a partial function, the sets $\mathrm{pred}(s, h, i)$ and $\mathrm{pred}(s, h, j)$ intersect only if $s(\mathtt{x}_i) = s(\mathtt{x}_j)$, in which case $\mathrm{pred}(s, h, i) = \mathrm{pred}(s, h, j)$.

We introduce a second partition of $\mathrm{dom}(h)$ by distinguishing the locations related to a cell involving a program variable interpretation on the one hand, and the remaining locations in the domain on the other hand. So, the sets below

are also implicitly parameterized by $\mathcal{V}$: $\mathrm{ref}(s,h) \stackrel{\text{def}}{=} \mathrm{dom}(h) \cap s(\mathcal{V})$, $\mathrm{acc}(s,h) \stackrel{\text{def}}{=} \mathrm{dom}(h) \cap h(s(\mathcal{V}))$, $\heartsuit(s,h) \stackrel{\text{def}}{=} \mathrm{ref}(s,h) \cup \mathrm{acc}(s,h)$, $\overline{\heartsuit}(s,h) \stackrel{\text{def}}{=} \mathrm{dom}(h) \backslash \heartsuit(s,h)$. The *core* of the memory state, written $\heartsuit(s,h)$, contains the locations l in $\mathrm{dom}(h)$ such that either l is the interpretation of a program variable or it is an image by h of a program variable (that is also in the domain). In the sequel, we need to consider locations that belong to the intersection of sets from different partitions. Here are their formal definitions:



- $\mathrm{pred}_{\overline{\heartsuit}}(s,h) \stackrel{\text{def}}{=} \mathrm{pred}(s,h) \backslash \heartsuit(s,h)$,
  $\mathrm{pred}_{\overline{\heartsuit}}(s,h,i) \stackrel{\text{def}}{=} \mathrm{pred}(s,h,i) \backslash \heartsuit(s,h)$,
- $\mathrm{loop}_{\overline{\heartsuit}}(s,h) \stackrel{\text{def}}{=} \mathrm{loop}(s,h) \backslash \heartsuit(s,h)$,
  $\mathrm{rem}_{\overline{\heartsuit}}(s,h) \stackrel{\text{def}}{=} \mathrm{rem}(s,h) \backslash \heartsuit(s,h)$.

For instance, $\mathrm{pred}_{\overline{\heartsuit}}(s,h)$ contains the set of locations l from $\mathrm{dom}(h)$, that are predecessors of a variable interpretation but no program variable $\mathtt{x}$ in $\{\mathtt{x}_1, \ldots, \mathtt{x}_q\}$ satisfies $s(\mathtt{x}) = l$ or $h(s(\mathtt{x})) = l$ (which means $l \notin \heartsuit(s,h)$).

The above figure presents a memory state (s,h) with the variables $\mathtt{x}_1, \ldots, \mathtt{x}_4$. Nodes labelled by '$\heartsuit$' [resp. '$\circlearrowleft$', 'p', 'r'] belong to $\heartsuit(s,h)$ [resp. $\mathrm{loop}_{\overline{\heartsuit}}(s,h)$, $\mathrm{pred}_{\overline{\heartsuit}}(s,h)$, $\mathrm{rem}_{\overline{\heartsuit}}(s,h)$]. The introduction of the above sets provides a canonical way to decompose the heap domain, which will be helpful in the sequel.

Lemma 3 (Canonical decomposition). *For all stores s and heaps h, the following identity holds:* $\mathrm{dom}(h) = \heartsuit(s,h) \uplus \mathrm{pred}_{\overline{\heartsuit}}(s,h) \uplus \mathrm{loop}_{\overline{\heartsuit}}(s,h) \uplus \mathrm{rem}_{\overline{\heartsuit}}(s,h)$.

The proof is by easy verification since $\mathrm{pred}(s,h) \cap \mathrm{loop}(s,h) \subseteq \heartsuit(s,h)$.

Proposition 4. $\left\{\mathrm{pred}_{\overline{\heartsuit}}(s,h,i) \mid i \in [1,q]\right\}$ *is a partition of* $\mathrm{pred}_{\overline{\heartsuit}}(s,h)$.

Remark that both $\mathrm{pred}_{\overline{\heartsuit}}(s,h,i) = \emptyset$ or $\mathrm{pred}_{\overline{\heartsuit}}(s,h,i) = \mathrm{pred}_{\overline{\heartsuit}}(s,h,j)$ are possible. Below, we present properties about the canonical decomposition.

Proposition 5. *Let s, h, h_1, h_2 be such that $h = h_1 \uplus h_2$. Then,* $\heartsuit(s,h) \cap \mathrm{dom}(h_1) = \heartsuit(s,h_1) \uplus \Delta(s,h_1,h_2)$ *with* $\Delta(s,h_1,h_2) \stackrel{\text{def}}{=} \mathrm{dom}(h_1) \cap h_2(s(\mathcal{V})) \cap \overline{s(\mathcal{V})} \cap \overline{h_1(s(\mathcal{V}))}$ *(where $\overline{X} \stackrel{\text{def}}{=} \mathbb{N} \backslash X$).*

The set $\Delta(s,h_1,h_2)$ contains the locations belonging to the core of h and to the domain of h_1, without being in the core of h_1. Its expression in Proposition 5 uses only basic set-theoretical operations. From Proposition 5, we conclude that $\heartsuit(s,h_1 \uplus h_2)$ can be different from $\heartsuit(s,h_1) \uplus \heartsuit(s,h_2)$.

2.4 How to count in 1SL1

Let us define a formula that states that $\mathrm{loop}_{\overline{\heartsuit}}(s,h)$ has size at least k. First, we consider the following set of formulæ: $T_q = \{\mathtt{alloc}(\mathtt{x}_1), \ldots, \mathtt{alloc}(\mathtt{x}_q)\} \cup \{\mathtt{toalloc}(\mathtt{x}_1), \ldots, \mathtt{toalloc}(\mathtt{x}_q)\}$. For any map $\mathfrak{f} : T_q \to \{0,1\}$, we associate a

formula $\mathcal{A}_{\mathfrak{f}}$ defined by $\mathcal{A}_{\mathfrak{f}} \stackrel{\text{def}}{=} \bigwedge\{\mathcal{B} \mid \mathcal{B} \in T_q \text{ and } \mathfrak{f}(\mathcal{B}) = 1\} \wedge \bigwedge\{\neg\mathcal{B} \mid \mathcal{B} \in T_q \text{ and } \mathfrak{f}(\mathcal{B}) = 0\}$. Formula $\mathcal{A}_{\mathfrak{f}}$ is a conjunction made of literals from T_q such that a *positive* literal $\mathcal{B}$ occurs exactly when $\mathfrak{f}(\mathcal{B}) = 1$ and a *negative* literal $\neg\mathcal{B}$ occurs exactly when $\mathfrak{f}(\mathcal{B}) = 0$. We write $\mathcal{A}_{\mathfrak{f}}^{\text{pos}}$ to denote $\bigwedge\{\mathcal{B} \mid \mathcal{B} \in T_q \text{ and } \mathfrak{f}(\mathcal{B}) = 1\}$.

Let us define the formula $\#\,\mathtt{loop} \geqslant k$ by $(\exists \mathtt{u}\ \mathtt{u} \hookrightarrow \mathtt{u}) * \cdots * (\exists \mathtt{u}\ \mathtt{u} \hookrightarrow \mathtt{u})$ (repeated k times). We can express that $\text{loop}_{\overline{\heartsuit}}(s,h)$ has size at least k (where $k \geqslant 1$) with $\#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant k \stackrel{\text{def}}{=} \bigvee_{\mathfrak{f}} \mathcal{A}_{\mathfrak{f}} \wedge \Big(\mathcal{A}_{\mathfrak{f}}^{\text{pos}} * (\#\,\mathtt{loop} \geqslant k)\Big)$, where $\mathfrak{f}$ spans over the finite set of maps $T_q \to \{0,1\}$. So, the idea behind the construction of the formula is to divide the heap into two parts: one subheap contains the full core. Then, any loop in the other subheap is out of the core because of the separation.

Lemma 6. *(I) For any $k \geqslant 1$, there is a formula $\#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant k$ s.t. for any (s,h), we have $(s,h) \models \#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant k$ iff $\text{card}(\text{loop}_{\overline{\heartsuit}}(s,h)) \geqslant k$. (II) For any $k \geqslant 1$ and any $i \in [1,q]$, there is a formula $\#\,\mathtt{pred}^{i}_{\overline{\heartsuit}} \geqslant k$ s.t. for any (s,h), we have $(s,h) \models \#\,\mathtt{pred}^{i}_{\overline{\heartsuit}} \geqslant k$ iff $\text{card}(\text{pred}_{\overline{\heartsuit}}(s,h,i)) \geqslant k$. (III) For any $k \geqslant 1$, there is a $\#\,\mathtt{rem}_{\overline{\heartsuit}} \geqslant k$ s.t. for any (s,h), we have $(s,h) \models \#\,\mathtt{rem}_{\overline{\heartsuit}} \geqslant k$ iff $\text{card}(\text{rem}_{\overline{\heartsuit}}(s,h)) \geqslant k$.*

All formulae from the above lemma have threshold polynomial in $q + \alpha$.

3 Expressive Completeness

3.1 On comparing cardinalities: equipotence

We introduce the notion of *equipotence* and state a few properties about it. This will be useful in the forthcoming developments. Let $\alpha \in \mathbb{N}$. We say that two finite sets X and Y are *α-equipotent* and we write $X \sim_\alpha Y$ if, either $\text{card}(X) = \text{card}(Y)$ or both $\text{card}(X)$ and $\text{card}(Y)$ are greater that α. The equipotence relation is also decreasing, i.e. $\sim_{\alpha_2} \subseteq \sim_{\alpha_1}$ holds for all $\alpha_1 \leqslant \alpha_2$. We state below two lemmas that will be helpful in the sequel.

Lemma 7. *Let $\alpha \in \mathbb{N}$ and X, X', Y, Y' be finite sets such that $X \cap X' = \emptyset$, $Y \cap Y' = \emptyset$, $X \sim_\alpha Y$ and $\text{card}(X') = \text{card}(Y')$ hold. Then $X \uplus X' \sim_\alpha Y \uplus Y'$ holds.*

Lemma 8. *Let $\alpha_1, \alpha_2 \in \mathbb{N}$ and X, X', Y_0 be finite sets s.t. $X \uplus X' \sim_{\alpha_1+\alpha_2} Y_0$ holds. Then there are two finite sets Y, Y' s.t. $Y_0 = Y \uplus Y'$, $X \sim_{\alpha_1} Y$ and $X' \sim_{\alpha_2} Y'$ hold.*

3.2 All we need is test formulæ

Test formulæ express simple properties about the memory states; this includes properties about program variables but also global properties about numbers of predecessors or loops, following the decomposition in Section 2.3. These test formulæ allow us to characterize the expressive power of 1SL1, similarly to what has been done in [17, 18, 3] for 1SL0. Since every formula in 1SL1 is shown equivalent to a Boolean combination of test formulæ (forthcoming Theorem 19), this process can be viewed as a means to eliminate separating connectives in

a controlled way; elimination is not total since the test formulæ require such separating connectives. However, this is analogous to quantifier elimination in Presburger arithmetic for which simple modulo constraints need to be introduced in order to eliminate the quantifiers (of course, modulo constraints are defined with quantifiers but in a controlled way too).

Let us introduce the *test formulæ*. We distinguish two types, leading to distinct sets. There are test formulæ that state properties about the direct neighbourhood of program variables whereas others state global properties about the memory states. The test formulæ of the form $\#\,\mathtt{pred}^{i}_{\overline{\heartsuit}} \geqslant k$ are of these two types but they will be included in Size_α since these are cardinality constraints.

Definition 9 (Test formulæ). *Given $q, \alpha \geqslant 1$, we define sets of test formulæ:*

$$
\begin{array}{rcl}
\mathsf{Equality} & \stackrel{def}{=} & \{\mathtt{x}_i = \mathtt{x}_j \mid i, j \in [1, q]\} \\
\mathsf{Pattern} & \stackrel{def}{=} & \{\mathtt{x}_i \hookrightarrow \mathtt{x}_j, \mathtt{conv}(\mathtt{x}_i, \mathtt{x}_j), \mathtt{inbetween}(\mathtt{x}_i, \mathtt{x}_j) \mid i, j \in [1, q]\} \\
 & \cup & \{\mathtt{toalloc}(\mathtt{x}_i), \mathtt{toloop}(\mathtt{x}_i), \mathtt{alloc}(\mathtt{x}_i) \mid i \in [1, q]\} \\
\mathsf{Extra}^{\mathtt{u}} & \stackrel{def}{=} & \{\mathtt{u} \hookrightarrow \mathtt{u}, \mathtt{alloc}(\mathtt{u})\} \cup \{\mathtt{x}_i = \mathtt{u}, \mathtt{x}_i \hookrightarrow \mathtt{u}, \mathtt{u} \hookrightarrow \mathtt{x}_i \mid i \in [1, q]\} \\
\mathsf{Size}_\alpha & \stackrel{def}{=} & \{\#\,\mathtt{pred}^{i}_{\overline{\heartsuit}} \geqslant k \mid i \in [1, q], k \in [1, \alpha]\} \\
 & \cup & \{\#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant k, \#\,\mathtt{rem}_{\overline{\heartsuit}} \geqslant k \mid k \in [1, \alpha]\}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{Basic} & \stackrel{def}{=} & \mathsf{Equality} \cup \mathsf{Pattern} & \mathsf{Test}_\alpha & \stackrel{def}{=} & \mathsf{Basic} \cup \mathsf{Size}_\alpha \cup \{\bot\} \\
\mathsf{Basic}^{\mathtt{u}} & \stackrel{def}{=} & \mathsf{Basic} \cup \mathsf{Extra}^{\mathtt{u}} & \mathsf{Test}^{\mathtt{u}}_\alpha & \stackrel{def}{=} & \mathsf{Test}_\alpha \cup \mathsf{Extra}^{\mathtt{u}}
\end{array}
$$

Test formulæ express simple properties about the memory states, even though quite large formulæ in 1SL1 may be needed to express them, while being of memory threshold polynomial in $q+\alpha$. An *atom* is a conjunction of test formulæ or their negation (literal) such that each formula from $\mathsf{Test}^{\mathtt{u}}_\alpha$ occurs once (saturated conjunction of literals). Any memory state satisfying an atom containing $\neg\mathtt{alloc}(\mathtt{x}_1) \ \wedge\ \neg\#\,\mathtt{pred}^{1}_{\overline{\heartsuit}} \geqslant 1 \ \wedge\ \neg\#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant 1 \ \wedge\ \neg\#\,\mathtt{rem}_{\overline{\heartsuit}} \geqslant 1$ (with $q = 1$) has an empty heap.

Lemma 10. *Satisfiability of conjunctions of test formulæ or their negation can be checked in polynomial time (q and α are not fixed and the bounds k in test formulæ from Size_α are encoded in binary).*

The tedious proof of Lemma 10 is based on a saturation algorithm. The *size* of a Boolean combination of test formulæ is the number of symbols to write it, when integers are encoded in binary (those from Size_α). Lemma 10 entails the following complexity characterization, which indeed makes a contrast with the complexity of the satisfiability problem for 1SL1 (see Theorem 28).

Theorem 11. *Satisfiability problem for Boolean combinations of test formulæ in set $\bigcup_{\alpha \geqslant 1} \mathsf{Test}^{\mathtt{u}}_\alpha$ (q and α are not fixed, and bounds k are encoded in binary) is* NP*-complete.*

Checking the satisfiability status of a Boolean combination of test formulæ is typically the kind of tasks that could be performed by an SMT solver, see e.g. [8, 1]. This is particularly true since no quantification is involved and test formulæ are indeed atomic formulæ about the theory of memory states.

Below, we introduce equivalence relations depending on whether memory states are indistinguishable w.r.t. some set of test formulæ.

Definition 12. *We say that (s, h, l) and (s', h', l') are basically equivalent [resp. extra equivalent, resp. α-equivalent] and we denote $(s, h, l) \simeq_b (s', h', l')$ [resp. $(s, h, l) \simeq_{\mathtt{u}} (s', h', l')$, resp. $(s, h, l) \simeq_\alpha (s', h', l')$] when the condition $(s, h) \models_l \mathcal{B}$ iff $(s', h') \models_{l'} \mathcal{B}$ is fulfilled for any $\mathcal{B} \in \mathsf{Basic}^{\mathtt{u}}$ [resp. $\mathcal{B} \in \mathsf{Extra}^{\mathtt{u}}$, resp. $\mathcal{B} \in \mathsf{Test}^{\mathtt{u}}_\alpha$].*

Hence (s, h, l) and (s', h', l') are basically equivalent [resp. extra equivalent, resp. α-equivalent] if and only if they cannot be distinguished by the formulæ of $\mathsf{Basic}^{\mathtt{u}}$ [resp. $\mathsf{Extra}^{\mathtt{u}}$, resp. $\mathsf{Test}^{\mathtt{u}}_\alpha$]. Since $\mathsf{Extra}^{\mathtt{u}} \subseteq \mathsf{Basic}^{\mathtt{u}} \subseteq \mathsf{Test}^{\mathtt{u}}_\alpha$, it is obvious that the inclusions $\simeq_\alpha \subseteq \simeq_b \subseteq \simeq_{\mathtt{u}}$ hold.

Proposition 13. *$(s, h, l) \simeq_\alpha (s', h', l')$ is equivalent to (1) $(s, h, l) \simeq_b (s', h', l')$ and (2) $\mathrm{pred}_{\overline{\heartsuit}}(s, h, i) \sim_\alpha \mathrm{pred}_{\overline{\heartsuit}}(s', h', i)$ for any $i \in [1, q]$, and (3) $\mathrm{loop}_{\overline{\heartsuit}}(s, h) \sim_\alpha \mathrm{loop}_{\overline{\heartsuit}}(s', h')$ and (4) $\mathrm{rem}_{\overline{\heartsuit}}(s, h) \sim_\alpha \mathrm{rem}_{\overline{\heartsuit}}(s', h')$.*

The proof is based on the identity $\mathsf{Basic}^{\mathtt{u}} \cup \mathsf{Size}_\alpha = \mathsf{Test}^{\mathtt{u}}_\alpha$. The *pseudo-core* of (s, h), written $\mathfrak{p}\heartsuit(s, h)$, is defined as $\mathfrak{p}\heartsuit(s, h) = s(\mathcal{V}) \cup h(s(\mathcal{V}))$ and $\heartsuit(s, h)$ is equal to $\mathfrak{p}\heartsuit(s, h) \cap \mathrm{dom}(h)$.

Lemma 14 (Bijection between pseudo-cores). *Let $l_0, l_1 \in \mathbb{N}$ and (s, h) and (s', h') be two memory states s.t. $(s, h, l_0) \simeq_b (s', h', l_1)$. Let $\mathfrak{R}$ be the binary relation on $\mathbb{N}$ defined by: $l \mathrel{\mathfrak{R}} l'$ iff (a) [$l = l_0$ and $l' = l_1$] or (b) there is $i \in [1, q]$ s.t. [$l = s(\mathtt{x}_i)$ and $l' = s'(\mathtt{x}_i)$] or [$l = h(s(\mathtt{x}_i))$ and $l' = h'(s'(\mathtt{x}_i))$]. Then $\mathfrak{R}$ is a bijective relation between $\mathfrak{p}\heartsuit(s, h) \cup \{l_0\}$ and $\mathfrak{p}\heartsuit(s', h') \cup \{l_1\}$. Its restriction to $\heartsuit(s, h)$ is in bijection with $\heartsuit(s', h')$ too if case (a) is dropped out from definition of $\mathfrak{R}$.*

3.3 Expressive completeness of 1SL1 with respect to test formulæ

Lemmas 15, 16 and 17 below roughly state that the relation $\simeq_\alpha$ behaves properly. Each lemma corresponds to a given quantifier, respectively separating conjunction, magic wand and first-order quantifier. Lemma 15 below states how two equivalent memory states can be split, while loosing a bit of precision.

Lemma 15 (Distributivity). *Let us consider s, h, h_1, h_2, s', h' and $\alpha, \alpha_1, \alpha_2 \geqslant 1$ such that $h = h_1 \uplus h_2$ and $\alpha = \alpha_1 + \alpha_2$ and $(s, h, l) \simeq_\alpha (s', h', l')$. Then there exists h'_1 and h'_2 s.t. $h' = h'_1 \uplus h'_2$ and $(s, h_1, l) \simeq_{\alpha_1} (s', h'_1, l')$ and $(s, h_2, l) \simeq_{\alpha_2} (s', h'_2, l')$.*

Given (s, h), we write $\mathtt{maxval}(s, h)$ to denote $\mathtt{max}(s(\mathcal{V}) \cup \mathrm{dom}(h) \cup \mathrm{ran}(h))$. Lemma 16 below states how it is possible to add subheaps while partly preserving precision.

Lemma 16. *Let $\alpha, q \geqslant 1$ and $l_0, l'_0 \in \mathbb{N}$. Assume that $(s, h, l_0) \simeq_{q+\alpha} (s', h', l'_0)$ and $h_0 \perp h$. Then there is $h'_0 \perp h'$ such that (1) $(s, h_0, l_0) \simeq_\alpha (s', h'_0, l'_0)$ (2) $(s, h \uplus h_0, l_0) \simeq_\alpha (s', h' \uplus h'_0, l'_0)$; (3) $\mathtt{maxval}(s', h'_0) \leqslant \mathtt{maxval}(s', h') + l'_0 + 3(q+1)\alpha + 1$.*

Note the precision lost from $(s, h, l_0) \simeq_{q+\alpha} (s', h', l'_0)$ to $(s, h_0, l_0) \simeq_\alpha (s', h'_0, l'_0)$.

Lemma 17 (Existence). *Let $\alpha \geqslant 1$ and let us assume $(s, h, l_0) \simeq_\alpha (s', h', l_1)$. We have: (1) for every $l \in \mathbb{N}$, there is $l' \in \mathbb{N}$ such that $(s, h, l) \simeq_{\mathtt{u}} (s', h', l')$; (2) for all l, l', $(s, h, l) \simeq_{\mathtt{u}} (s', h', l')$ iff $(s, h, l) \simeq_\alpha (s', h', l')$.*

Now, we state the main property in the section, namely test formulæ provide the proper abstraction.

Lemma 18 (Correctness). *For any $\mathcal{A}$ in 1SL1 with at most $q \geqslant 1$ program variables, if $(s, h, l) \simeq_\alpha (s', h', l')$ and $\mathtt{th}(q, \mathcal{A}) \leqslant \alpha$ then $(s, h) \models_l \mathcal{A}$ iff $(s', h') \models_{l'} \mathcal{A}$.*

The proof is by structural induction on $\mathcal{A}$ using Lemma 15, 16 and 17. Here is one of our main results characterizing the expressive power of 1SL1.

Theorem 19 (Quantifier Admissibility). *Every formula $\mathcal{A}$ in 1SL1 with q program variables is logically equivalent to a Boolean combination of test formulæ in $\mathsf{Test}^{\mathtt{u}}_{\mathtt{th}(q,\mathcal{A})}$.*

The proof of Theorem 19 does not provide a constructive way to eliminate quantifiers, which will be done in Section 4 (see Corollary 30).

Proof. Let $\alpha = \mathtt{th}(q, \mathcal{A})$ and consider the set of literals $\mathcal{S}_\alpha(s, h, l) \stackrel{\text{def}}{=} \{\mathcal{B} \mid \mathcal{B} \in \mathsf{Test}^{\mathtt{u}}_\alpha \text{ and } (s, h) \models_l \mathcal{B}\} \cup \{\neg\mathcal{B} \mid \mathcal{B} \in \mathsf{Test}^{\mathtt{u}}_\alpha \text{ and } (s, h) \not\models_l \mathcal{B}\}$. As $\mathsf{Test}^{\mathtt{u}}_\alpha$ is finite, the set $\mathcal{S}_\alpha(s, h, l)$ is finite and let us consider the well-defined atom $\bigwedge \mathcal{S}_\alpha(s, h, l)$. We have $(s', h') \models_{l'} \bigwedge \mathcal{S}_\alpha(s, h, l)$ iff $(s, h, l) \simeq_\alpha (s', h', l')$. The disjunction $\mathcal{T}_\mathcal{A} \stackrel{\text{def}}{=} \bigvee\{\bigwedge \mathcal{S}_\alpha(s, h, l) \mid (s, h) \models_l \mathcal{A}\}$ is a (finite) Boolean combination of test formulæ in $\mathsf{Test}^{\mathtt{u}}_\alpha$ because $\bigwedge \mathcal{S}_\alpha(s, h, l)$ ranges over the finite set of atoms built from $\mathsf{Test}^{\mathtt{u}}_\alpha$. By Lemma 18, we get that $\mathcal{A}$ is logically equivalent to $\mathcal{T}_\mathcal{A}$. □

When $\mathcal{A}$ in 1SL1 has no free occurrence of $\mathtt{u}$, one can show that $\mathcal{A}$ is equivalent to a Boolean combination of formulæ in $\mathsf{Test}_{\mathtt{th}(q,\mathcal{A})}$. Similarly, when $\mathcal{A}$ in 1SL1 has no occurrence of $\mathtt{u}$ at all, $\mathcal{A}$ is equivalent to a Boolean combination of formulæ of the form $\mathtt{x}_i = \mathtt{x}_j$, $\mathtt{x}_i \hookrightarrow \mathtt{x}_j$, $\mathtt{alloc}(\mathtt{x}_i)$ and $\#\,\mathtt{rem}_{\overline{\heartsuit}} \geqslant k$ with the *alternative* definition $\heartsuit(s, h) = \{s(\mathtt{x}_i) : s(\mathtt{x}_i) \in \mathrm{dom}(h),\ i \in [1, q]\}$ (see also [17, 18, 3]). Theorem 19 witnesses that the test formulæ we introduced properly abstract memory states when 1SL1 formulæ are involved. Test formulæ from Definition 9 were not given to us and we had to design such formulæ to conclude Theorem 19. Let us see what the test formulæ satisfy. Above all, all the test formulæ can be expressed in 1SL1, see developments in Section 2.2 and Lemma 6. Then, we aim at avoiding redundancy among the test formulæ. Indeed, for any kind of test formulæ from $\mathsf{Test}^{\mathtt{u}}_\alpha$ leading to the subset $X \subseteq \mathsf{Test}^{\mathtt{u}}_\alpha$ (for instance $X = \{\#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant k \mid k \leqslant \alpha\}$), there are (s, h), (s', h') and $l, l' \in \mathbb{N}$ such that (1) for every $\mathcal{B} \in \mathsf{Test}^{\mathtt{u}}_\alpha \backslash X$, we have $(s, h) \models_l \mathcal{B}$ iff $(s', h') \models_{l'} \mathcal{B}$ but (2) there is $\mathcal{B} \in X$ such that not ($(s, h) \models_l \mathcal{B}$ iff $(s', h') \models_{l'} \mathcal{B}$). When $X = \{\#\,\mathtt{loop}_{\overline{\heartsuit}} \geqslant k \mid k \leqslant \alpha\}$, clearly, the other test formulæ cannot systematically enforce constraints on the cardinality of the set of loops outside of the core. Last but not least, we need to prove that the set of test formulæ is expressively complete to get Theorem 19. Lemmas 15, 16 and 17 are helpful to obtain Lemma 18 taking care of the different quantifiers. It is in their proofs that the completeness of the set $\mathsf{Test}^{\mathtt{u}}_\alpha$ is

best illustrated. Nevertheless, to apply these lemmas in the proof of Lemma 18, we designed the adequate definition for the function $\mathtt{th}(\cdot,\cdot)$ and we arranged different thresholds in their statements. So, there is a real interplay between the definition of $\mathtt{th}(\cdot,\cdot)$ and the lemmas used in the proof of Lemma 18.

A small model property can be also proved as a consequence of Theorem 19 and the proof of Lemma 10, for instance.

Corollary 20 (Small Model Property). *Let $\mathcal{A}$ be a formula in 1SL1 with q program variables. Then, if $\mathcal{A}$ is satisfiable, then there is a memory state (s,h) and $l \in \mathbb{N}$ such that $(s,h) \models_l \mathcal{A}$ and $\mathtt{max}(\mathtt{maxval}(s,h),l) \leqslant 3(q+1)+(q+3)\mathtt{th}(q,\mathcal{A})$.*

There is no need to count over $\mathtt{th}(q,\mathcal{A})$ (e.g., for the loops outside the core) and the core uses at most $3q$ locations. Theorem 19 provides a characterization of the expressive power of 1SL1, which is now easy to differenciate from 1SL2.

Corollary 21. *1SL2 is strictly more expressive than 1SL1.*

4 Deciding 1SL1 Satisfiability and Model-Checking Problems

4.1 Abstracting further memory states

Satisfaction of $\mathcal{A}$ depends only on the satisfaction of formulæ from $\mathsf{Test}^{\mathtt{u}}_{\mathtt{th}(q,\mathcal{A})}$. So, to check satisfiability of $\mathcal{A}$, there is no need to build memory states but rather only abstractions in which only the truth value of test formulæ matters. In this section we introduce *abstract memory states* and we show how it matches indistinguihability with respect to test formulae in $\mathsf{Test}^{\mathtt{u}}_{\alpha}$ (Lemma 24). Then, we use these abstract structures to design a model-checking decision procedure that runs in nondeterministic polynomial space.

Definition 22. *Let $q,\alpha \geqslant 1$. An* abstract memory state $\mathfrak{a}$ *over (q,α) is a structure $((V,E),\mathfrak{l},\mathfrak{r},\mathfrak{p}_1,\ldots,\mathfrak{p}_q)$ such that:*

1. *There is a partition P of $\{\mathtt{x}_1,\ldots,\mathtt{x}_q\}$ such that $P \subseteq V$. This encodes the store.*
2. *(V,E) is a functional directed graph and a node v in (V,E) is at distance at most two of some set of variables X in P. This allows to encode only the pseudo-core of memory states and nothing else.*
3. *$\mathfrak{l},\mathfrak{p}_1,\ldots,\mathfrak{p}_q,\mathfrak{r} \in [0,\alpha]$ and this corresponds to the number of self-loops [resp. numbers of predecessors, number of remaining allocated locations] out of the core, possibly truncated over α. We require that if $\mathtt{x}_i$ and $\mathtt{x}_j$ belong to the same set in the partition P, then $\mathfrak{p}_i = \mathfrak{p}_j$.*

Given $q,\alpha \geqslant 1$, the number of abstract memory states over (q,α) is not only finite but reasonably bounded. Given (s,h), we define its *abstraction* $\mathrm{abs}(s,h)$ over (q,α) as the abstract memory state $((V,E),\mathfrak{l},\mathfrak{r},\mathfrak{p}_1,\ldots,\mathfrak{p}_q)$ such that

- $\mathfrak{l} = \mathtt{min}(\mathrm{loop}_{\overline{\heartsuit}}(s,h),\alpha)$, $\mathfrak{r} = \mathtt{min}(\mathrm{rem}_{\overline{\heartsuit}}(s,h),\alpha)$, $\mathfrak{p}_i = \mathtt{min}(\mathrm{pred}_{\overline{\heartsuit}}(s,h,i),\alpha)$ for every $i \in [1,q]$.
- P is a partition of $\{\mathtt{x}_1,\ldots,\mathtt{x}_q\}$ so that for all $\mathtt{x},\mathtt{x}'$, $s(\mathtt{x}) = s(\mathtt{x}')$ iff $\mathtt{x}$ and $\mathtt{x}'$ belong to the same set in P.

– V is made of elements from P as well as of locations from the set below:

$$\big(\{h(s(\mathtt{x}_i)) : s(\mathtt{x}_i) \in \mathrm{dom}(h), i \in [1,q]\} \cup$$

$$\{h(h(s(\mathtt{x}_i))) : h(s(\mathtt{x}_i)) \in \mathrm{dom}(h), i \in [1,q]\}\big) \backslash \{s(\mathtt{x}_i) : i \in [1,q]\}$$

– The graph (V, E) is defined as follows:
  1. $(X, X') \in E$ if $X, X' \in P$ and $h(s(\mathtt{x})) = s(\mathtt{x}')$ for some $\mathtt{x} \in X$, $\mathtt{x}' \in X'$.
  2. $(X, l) \in E$ if $X \in P$ and $h(s(\mathtt{x})) = l$ for some variable $\mathtt{x}$ in X and $l \notin \{s(\mathtt{x}_i) : i \in [1,q]\}$.
  3. $(l, l') \in E$ if there is a set $X \in P$ such that $(X, l) \in E$ and $h(l) = l'$ and $l' \notin \{s(\mathtt{x}_i) : i \in [1,q]\}$.
  4. $(l, X) \in E$ if there is $X' \in P$ such that $(X', l) \in E$ and $h(l) = s(\mathtt{x})$ for some $\mathtt{x} \in X$ and $l \notin \{s(\mathtt{x}_i) : i \in [1,q]\}$.

We define abstract memory states to be *isomorphic* if (1) the partition P is identical, (2) the finite digraphs satisfy the same formulæ from Basic when the digraphs are understood as heap graphs restricted to locations at distance at most two from program variables, and (3) all the numerical values are identical. A *pointed abstract memory state* is a pair $(\mathfrak{a}, \mathfrak{u})$ such that $\mathfrak{a} = ((V, E), \mathfrak{l}, \mathfrak{r}, \mathfrak{p}_1, \ldots, \mathfrak{p}_q)$ is an abstract memory state and $\mathfrak{u}$ takes one of the following values: $\mathfrak{u} \in V$ and $\mathfrak{u}$ is at distance at most one from some $X \in P$, or $\mathfrak{u} = L$ but $\mathfrak{l} > 0$ is required, or $\mathfrak{u} = R$ but $\mathfrak{r} > 0$ is required, or $\mathfrak{u} = P(i)$ for some $i \in [1,q]$ but $\mathfrak{p}_i > 0$ is required, or $\mathfrak{u} = \overline{D}$. Given a memory state (s, h) and $l \in \mathbb{N}$, we define its *abstraction* $\mathrm{abs}(s, h, l)$ with respect to (q, α) as the pointed abstract memory state $(\mathfrak{a}, \mathfrak{u})$ such that $\mathfrak{a} = \mathrm{abs}(s, h)$ and

– $\mathfrak{u} \in V$ if either $l \in V$ and distance is at most one from some $X \in P$, or $\mathfrak{u} = X$ and there is $\mathtt{x} \in X \in P$ such that $s(\mathtt{x}) = l$,
– or $\mathfrak{u} \stackrel{\text{def}}{=} L$ if $l \in \mathrm{loop}_{\overline{\heartsuit}}(s, h)$, or $\mathfrak{u} \stackrel{\text{def}}{=} R$ if $l \in \mathrm{rem}_{\overline{\heartsuit}}(s, h)$,
– or $\mathfrak{u} \stackrel{\text{def}}{=} P(i)$ if $l \in \mathrm{pred}_{\overline{\heartsuit}}(s, h, i)$ for some $i \in [1,q]$,
– or $\mathfrak{u} \stackrel{\text{def}}{=} \overline{D}$ if none of the above conditions applies (so $l \notin \mathrm{dom}(h)$).

Pointed abstract memory states $(\mathfrak{a}, \mathfrak{u})$ and $(\mathfrak{a}', \mathfrak{u}')$ are *isomorphic* $\stackrel{\text{def}}{\Leftrightarrow}$ $\mathfrak{a}$ and $\mathfrak{a}'$ are isomorphic and, $\mathfrak{u} = \mathfrak{u}'$ or $\mathfrak{u}$ and $\mathfrak{u}'$ are related by the isomorphism.

Lemma 23. *Given a pointed abstract memory state $(\mathfrak{a}, \mathfrak{u})$ over (q, α), there exist a memory state (s, h) and $l \in \mathbb{N}$ such that $\mathrm{abs}(s, h, l)$ and $(\mathfrak{a}, \mathfrak{u})$ are isomorphic*

Abstract memory states is the right way to abstract memory states when the language 1SL1 is involved, which can be formally stated as follows.

Lemma 24. *Let (s, h), (s', h') be memory states and $l, l' \in \mathbb{N}$. The next three propositions are equivalent: (1) $(s, h, l) \simeq_\alpha (s', h', l')$; (2) $\mathrm{abs}(s, h, l)$ and $\mathrm{abs}(s', h', l')$ are isomorphic; (3) there is a unique atom $\mathcal{B}$ from $\mathsf{Test}^{\mathfrak{u}}_\alpha$ s.t. $(s, h) \models_l \mathcal{B}$ and $(s', h') \models_{l'} \mathcal{B}$.*

Equivalence between (1) and (3) is a consequence of the definition of the relation $\simeq_\alpha$. Hence, a pointed abstract memory state represents an atom of $\mathsf{Test}^{\mathfrak{u}}_\alpha$, except that it is a bit more concise (only space in $\mathcal{O}(q + \log(\alpha))$ is required whereas an atom requires polynomial space in $q + \alpha$).

1: **if** $\mathcal{B}$ is atomic **then** return AMC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B})$;
2: **if** $\mathcal{B} = \neg\mathcal{B}_1$ **then** return not MC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B}_1)$;
3: **if** $\mathcal{B} = \mathcal{B}_1 \wedge \mathcal{B}_2$ **then** return (MC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B}_1)$ and MC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B}_2)$);
4: **if** $\mathcal{B} = \exists\, \mathtt{u}\, \mathcal{B}_1$ **then** return $\top$ iff there is $\mathfrak{u}'$ such that MC$((\mathfrak{a}, \mathfrak{u}'), \mathcal{B}_1) = \top$;
5: **if** $\mathcal{B} = \mathcal{B}_1 * \mathcal{B}_2$ **then** return $\top$ iff there are $(\mathfrak{a}_1, \mathfrak{u}_1)$ and $(\mathfrak{a}_2, \mathfrak{u}_2)$ such that $*_a((\mathfrak{a}, \mathfrak{u}), (\mathfrak{a}_1, \mathfrak{u}_1), (\mathfrak{a}_2, \mathfrak{u}_2))$ and MC$((\mathfrak{a}_1, \mathfrak{u}_1), \mathcal{B}_1) =$ MC$((\mathfrak{a}_2, \mathfrak{u}_2), \mathcal{B}_2) = \top$;
6: **if** $\mathcal{B} = \mathcal{B}_1 \mathrel{-\!\!*} \mathcal{B}_2$ **then** return $\bot$ iff for some $(\mathfrak{a}', \mathfrak{u}')$ and $(\mathfrak{a}'', \mathfrak{u}'')$ such that $*_a((\mathfrak{a}'', \mathfrak{u}''), (\mathfrak{a}', \mathfrak{u}'), (\mathfrak{a}, \mathfrak{u}))$, MC$((\mathfrak{a}', \mathfrak{u}'), \mathcal{B}_1) = \top$ and MC$((\mathfrak{a}'', \mathfrak{u}''), \mathcal{B}_2) = \bot$;

Fig. 1. Function MC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B})$

1: **if** $\mathcal{B}$ is $\mathtt{emp}$ **then** return $\top$ iff $E = \emptyset$ and all numerical values are zero;
2: **if** $\mathcal{B}$ is $\mathtt{x}_i = \mathtt{x}_j$ **then** return $\top$ iff $\mathtt{x}_i, \mathtt{x}_j \in X$, for some $X \in P$;
3: **if** $\mathcal{B}$ is $\mathtt{x}_i = \mathtt{u}$ **then** return $\top$ iff $\mathtt{u} = X$ for some $X \in P$ such that $\mathtt{x}_i \in X$;
4: **if** $\mathcal{B}$ is $\mathtt{u} = \mathtt{u}$ **then** return $\top$;
5: **if** $\mathcal{B}$ is $\mathtt{x}_i \hookrightarrow \mathtt{x}_j$ **then** return $\top$ iff $(X, X') \in E$ where $\mathtt{x}_i \in X \in P$ and $\mathtt{x}_j \in X \in P$;
6: **if** $\mathcal{B}$ is $\mathtt{x}_i \hookrightarrow \mathtt{u}$ **then** return $\top$ iff $(X, \mathtt{u}) \in E$ for some $X \in P$ such that $\mathtt{x}_i \in X$;
7: **if** $\mathcal{B}$ is $\mathtt{u} \hookrightarrow \mathtt{x}_i$ **then** return $\top$ iff either $\mathtt{u} = P(i)$ or ($\mathtt{u} \in V$ and there is some $X \in P$ such that $\mathtt{x}_i \in X$ and $(\mathtt{u}, X) \in E$);
8: **if** $\mathcal{B}$ is $\mathtt{u} \hookrightarrow \mathtt{u}$ **then** return $\top$ iff either $\mathtt{u} = L$ or $(\mathtt{u}, \mathtt{u}) \in E$;

Fig. 2. Function AMC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B})$

Definition 25. *Given pointed abstract memory states* $(\mathfrak{a}, \mathfrak{u})$, $(\mathfrak{a}_1, \mathfrak{u}_1)$ *and* $(\mathfrak{a}_2, \mathfrak{u}_2)$, *we write* $*_a((\mathfrak{a}, \mathfrak{u}), (\mathfrak{a}_1, \mathfrak{u}_1), (\mathfrak{a}_2, \mathfrak{u}_2))$ *if there exist* $l \in \mathbb{N}$, *a store* s *and disjoint heaps* h_1 *and* h_2 *such that* $\mathrm{abs}(s, h_1 \uplus h_2, l) = (\mathfrak{a}, \mathfrak{u})$, $\mathrm{abs}(s, h_1, l) = (\mathfrak{a}_1, \mathfrak{u}_1)$ *and* $\mathrm{abs}(s, h_2, l) = (\mathfrak{a}_2, \mathfrak{u}_2)$.

The ternary relation $*_a$ is not difficult to check even though it is necessary to verify that the abstract disjoint union is properly done.

Lemma 26. *Given* $q, \alpha \geqslant 1$, *the ternary relation* $*_a$ *can be decided in polynomial time in* $q + log(\alpha)$ *for all the pointed abstract memory states built over* (q, α).

4.2 A polynomial-space decision procedure

Figure 1 presents a procedure MC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B})$ returning a Boolean value in $\{\bot, \top\}$ and taking as arguments, a pointed abstract memory state over (q, α) and a formula $\mathcal{B}$ with $\mathtt{th}(q, \mathcal{B}) \leqslant \alpha$. All the quantifications over pointed abstract memory states are done over (q, α). A case analysis is provided depending on the outermost connective. Its structure is standard and mimicks the semantics for 1SL1 *except* that we deal with abstract memory states. The auxiliary function AMC$((\mathfrak{a}, \mathfrak{u}), \mathcal{B})$ also returns a Boolean value in $\{\bot, \top\}$, makes no recursive calls and is dedicated to atomic formulæ (see Figure 2). The design of MC is similar to nondeterministic polynomial space procedures, see e.g. [15, 6].

Lemma 27. *Let $q, \alpha \geqslant 1$, $(\mathfrak{a}, \mathfrak{u})$ be a pointed abstract memory state over (q, α) and $\mathcal{A}$ be in 1SL1 built over $\mathtt{x}_1, \ldots, \mathtt{x}_q$ s.t. $\mathtt{th}(q, \mathcal{A}) \leqslant \alpha$. The propositions below are equivalent: (I) MC$((\mathfrak{a}, \mathfrak{u}), \mathcal{A})$ returns $\top$; (II) There exist (s, h) and $l \in \mathbb{N}$ such that $\mathrm{abs}(s, h, l) = (\mathfrak{a}, \mathfrak{u})$ and $(s, h) \models_l \mathcal{A}$; (III) For all (s, h) and $l \in \mathbb{N}$ s.t. $\mathrm{abs}(s, h, l) = (\mathfrak{a}, \mathfrak{u})$, we have $(s, h) \models_l \mathcal{A}$.*

Consequently, we get the following complexity characterization.

Theorem 28. *Model-checking and satisfiability pbs. for 1SL1 are* PSPACE*-complete.*

Below, we state two nice by-products of our proof technique.

Corollary 29. *Let $q \geqslant 1$. The satisfiability problem for 1SL1 restricted to formulæ with at most q program variables can be solved in polynomial time.*

Corollary 30. *Given a formula $\mathcal{A}$ in 1SL1, computing a Boolean combination of test formulæ in $\mathsf{Test}^{\mathtt{u}}_{\mathtt{th}(q,\mathcal{A})}$ logically equivalent to $\mathcal{A}$ can be done in polynomial space (even though the outcome formula can be of exponential size).*

Here is another by-product of our proof technique. The PSPACE bound is preserved when formulæ are encoded as DAGs instead of trees. The size of a formula is then simply its number of subformulæ. This is similar to machine encoding, provides a better conciseness and complexity upper bounds are more difficult to obtain. With this alternative notion of length, $\mathtt{th}(q, \mathcal{A})$ is only bounded by $q \times 2^{|\mathcal{A}|}$ (compare with Lemma 1). Nevertheless, this is fine to get PSPACE upper bound with this encoding since the algorithm to solve the satisfiability problem runs in logarithmic space in α, as we have shown previously.

5 Conclusion

In [4], the undecidability of 1SL with a unique record field is shown. 1SL0 is also known to be PSPACE-complete [6]. In this paper, we provided an extension with a unique quantified variable and we show that the satisfiability problem for 1SL1 is PSPACE-complete by presenting an original and fine-tuned abstraction of memory states. We proved that in 1SL1 separating connectives can be eliminated in a controlled way as well as first-order quantification over the single variable. In that way, we show a quantifier elimination property. Apart from the complexity results and the new abstraction for memory states, we also show a quite surprising result: when the number of program variables is bounded, the satisfiability problem can be solved in polynomial time. Last but not least, we have established that satisfiability problem for Boolean combinations of test formulæ is NP-complete. This is reminiscent of decision procedures used in SMT solvers and it is a challenging question to take advantage of these features to decide 1SL1 with an SMT solver. Finally, the design of fragments between 1SL1 and undecidable 1SL2 that can be decided with an adaptation of our method is worth being further investigated.

Acknowledgments: We warmly thank the anonymous referees for their numerous and helpful suggestions, improving significantly the quality of the paper and its extended version [10]. Great thanks also to Morgan Deters (New York University) for feedback and discussions about this work.

References

1. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV'11*, LNCS, pages 171–177. Springer, 2011.
2. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, LNCS, pages 115–137. Springer, 2005.
3. R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. *APAL*, 161(3):305–323, 2009.
4. R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. *IC*, 211:106–137, 2012.
5. J. Brotherston and M. Kanovich. Undecidability of propositional separation logic and its neighbours. In *LICS'10*, pages 130–139. IEEE, 2010.
6. C. Calcagno, P. O'Hearn, and H. Yang. Computability and complexity results for a spatial assertion language for data structures. In *FSTTCS'01*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.
7. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR'11*, number 6901 in LNCS, pages 235–249. Springer, 2011.
8. L. de Moura and N. Björner. Z3: An Efficient SMT Solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. S. Demri and M. Deters. Two-variable separation logic and its inner circle, September 2013. Submitted.
10. S. Demri, D. Galmiche, D. Larchey-Wendling, and D. Mery. Separation logic with one quantified variable. arXiv, 2014.
11. D. Galmiche and D. Méry. Tableaux and resource graphs for separation logic. *JLC*, 20(1):189–231, 2010.
12. C. Haase, S. Ishtiaq, J. Ouaknine, and M. Parkinson. SeLoger: A Tool for Graph-Based Reasoning in Separation Logic. In *CAV'13*, volume 8044 of *LNCS*, pages 790–795. Springer, 2013.
13. R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *CADE'13*, LNCS, pages 21–38. Springer, 2013.
14. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26, 2001.
15. R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 6(3):467–480, 1977.
16. D. Larchey-Wendling and D. Galmiche. The undecidability of boolean BI through phase semantics. In *LICS'10*, pages 140–149. IEEE, 2010.
17. E. Lozes. *Expressivité des logiques spatiales*. PhD thesis, LIP, ENS Lyon, France, 2004.
18. E. Lozes. Separation logic preserves the expressive power of classical logic. In *Workshop SPACE'04*, 2004.
19. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *CAV'13*, volume 2013 of *LNCS*, pages 773–789. Springer, 2013.
20. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.