# Reasoning about sequences of memory states (preliminary version) *

Rémi Brochenin, Stéphane Demri, and Etienne Lozes

LSV, ENS Cachan, CNRS, INRIA
61, av. Pdt. Wilson, 94235 Cachan Cedex, France
email: {brocheni,demri,lozes}@lsv.ens-cachan.fr

**Abstract.** In order to verify programs with pointer variables, we introduce a temporal logic $\mathrm{LTL}^{\mathrm{mem}}$ whose underlying assertion language is the quantifier-free fragment of separation logic and the temporal logic on the top of it is the standard linear-time temporal logic LTL. We state the complexity of various model-checking and satisfiability problems for $\mathrm{LTL}^{\mathrm{mem}}$, considering various fragments of separation logic (including pointer arithmetic), various classes of models (with or without constant heap), and the influence of fixing the initial memory state. Our main decidability result is PSPACE-completeness of the satisfiability problems on the record fragment and on a classical fragment allowing pointer arithmetic. $\Sigma_1^0$-completeness or $\Sigma_1^1$-completeness results are established for various problems, and underline the tightness of our decidability results. This paper is a preliminary version of [BDL07].

## 1 Introduction

*Verification of programs with pointers.* Programs with pointer variables are worth to be analyzed and certified, since they easily contain programming errors that are sometimes difficult to detect. Such errors include the existence of memory leaks, memory violation, or undesirable aliasing. Prominent logics for analyzing such programs are Separation Logic [Rey02], pointer assertion logic PAL [JJKS97], TVLA [LAS00] and alias logic [BIL04], to quote a few examples.

*Temporal Separation Logic: what for?* Since [Pnu77], temporal logics are also used as languages for formal specification of programs. General and powerful automata-based techniques for verification have been developed, see e.g. [VW94]. On the other hand, Separation Logic is a static logic having a great success for program annotation [Rey02], and more recently for symbolic computation [BCO05]. Extending the scope of application of Separation Logic to standard temporal logic-based verification technique has many potential interests, either for model-checking programs, or for defining restricted forms of recursive predicates. For instance, if we write $\mathsf{X}\mathtt{x}$ to denote the next value of $\mathtt{x}$ (also sometimes written

* Work supported by the RNTL project "AVERILES". The first author is supported by a fellowship from CNRS/DGA.

$\mathtt{x}'$), the formula $(\mathtt{x} \hookrightarrow \mathsf{X}\mathtt{x})\mathsf{U}(\mathtt{x} \hookrightarrow \mathtt{null})$, understood on a model with constant heap, characterises the existence of a simple flat list, which is usually written $\mu L(\mathtt{x}).\ \mathtt{x} \hookrightarrow \mathtt{null} \vee \exists \mathtt{x}'.\mathtt{x} \hookrightarrow \mathtt{x}' \wedge L(\mathtt{x}')$.

Temporal logics also allow to work in the very convenient framework of "programs-as-formulae" and decision procedures for logical problems can be directly used for program verification, see a standard reduction in [SC85]. For instance, the previous formula can be seen as a program walking on a list, and more generally programs without updates can be expressed as formulae. Some programs with update that perform a simple pass on the heap, have an input-output relation that may be described by a formula. For instance, the formula $(\mathtt{x} \hookrightarrow_0 \mathsf{X}\mathtt{x} \wedge \mathsf{X}\mathtt{x} \hookrightarrow_1 \mathtt{x})\mathsf{U}\mathtt{x} \hookrightarrow_0 \mathtt{null}$ expresses broadly that the list in the initial heap $h_0$ is reversed in the final heap $h_1$ (see also Sect. 4.2).

As a side interest, up to our knowledge, few decision procedures for programs working with pointer arithmetic have been proposed up to now, whereas arithmetical constraints in temporal logics are known to lead to undecidability, see e.g. [CC00]. Actually, there is a growing interest in understanding the interplay of pointer arithmetic, temporal reasonning, and non aliasing properties.

*Our contribution.* In this note, we introduce a linear-time temporal logic $\mathrm{LTL}^{\mathrm{mem}}$ to specify sequences of memory states with underlying assertion language based on quantifier-free Separation Logic [Rey02]. Our logic addresses a very general notion of models, including the aspects of pointer arithmetic and recursive structures with records. We distinguish the satisfiability problems from the model-checking problems, as well as distinct subclasses of interesting programs, like for instance the programs without destructive update. We have shown the PSPACE-completeness of the satisfiability problems SAT(CL) and SAT(RF) where CL is the classical fragment without separation connectives and RF is the record fragment with no pointer arithmetic but with separation connectives. This result is very tight, as both propositional LTL and static Separation Logic are already PSPACE-complete [SC85,CYO01a]. We have obtained these results by reduction to the nonemptiness problem for Büchi automata on an alphabet of symbolic memory states obtained by an abstraction that we have shown sound and complete. This is a variant of the automata-based approach introduced in [VW94] for plain LTL and further developed with concrete domains of interpretation in [DD07]. This result is not a direct consequence of the decidability of the state logic used in this temporal logic. For instance, Presburger arithmetic is decidable, but LTL with Presburger constraints is not. As a matter of fact, the abstraction method we use based on [Loz04] does not scale to the whole temporal logic, due to a subtle interplay between separation connectives and pointer arithmetic. Similar techniques can be found in [GM05]. Observe that the satisfiability problem for the whole state logic (SL) is decidable. Moreover, we have obtained new undecidability results for several problems, for instance for $\mathrm{SAT}^{ct}$(LF) (satisfiability with constant heap on the list fragment).

*Related work.* Previous temporal logics designed for pointer verification include Evolution Temporal Logic [YRSW03], based on the three-valued logic abstrac-

tion method that made the success of TVLA [LAS00], and Navigation temporal logic [DKR04], based on a tableau method for model-checking quite similar to our automaton-based reduction. In these works, the assertion language for states is quite rich, as it includes for instance list predicate, quantification over adresses, and a freshness predicate. The price of this expressiveness is that only incomplete abstractions are proposed, whereas we stick to exact methods. More importantly, our work addresses models with constant heaps and pointer arithmetic, which has not been done so far, and leads to a quite different perspective.

This paper is a preliminary version of [BDL07] even though Figure 1 contains few new results, see also a discussion in Sect. 4.

## 2 Memory model and specification language

We introduce below a separation logic dealing with pointer arithmetic and record values, and a temporal logic $\mathrm{LTL}^{\mathrm{mem}}$. Model-checking programs with pointer variables over $\mathrm{LTL}^{\mathrm{mem}}$ specifications is our main problem of interest.

### 2.1 A separation logic with pointer arithmetic

*Memory states.* First, let us introduce our model of memory. It captures features of programs with pointer variables that use pointer arithmetic and records. We assume a countably infinite set $\mathtt{Var}$ of variables (as usual, for a fixed formula we need only a finite amount), and an infinite set $\mathtt{Val}$ of values containing the set $\mathbb{N}$ of naturals, thought as address indexes, and a special value $nil$. For simplicity, we assume that $\mathtt{Val} = \mathbb{N} \uplus \{nil\}$. In order to model field selectors, we also consider some infinite set $\mathtt{Lab}$ of labels. In the remainder, we will assume some fixed injection $(\mathtt{x}, i) \in \mathtt{Var} \times \mathbb{N} \mapsto \langle \mathtt{x}, i \rangle \in \mathtt{Var}$.

We use the notation $E \rightharpoonup_{fin} F$ for the set of partial functions from $E$ to $F$ of finite domain; and $E \rightharpoonup_{fin+} F$ for the set of partial functions from $E$ to $F$ of finite and nonempty domain. The sets $\mathcal{S}$ of stores and $\mathcal{H}$ of heaps are then defined as follows: $\mathcal{S} \stackrel{\text{def}}{\equiv} \mathtt{Var} \to \mathtt{Val}$ and $\mathcal{H} \stackrel{\text{def}}{\equiv} \mathbb{N} \rightharpoonup_{fin} (\mathtt{Lab} \rightharpoonup_{fin+} \mathtt{Val})$. We call *memory state* a couple $(s, h) \in \mathcal{S} \times \mathcal{H}$.

We will refer to the domain of a heap $h$ by $\mathtt{dom}(h) \subseteq \mathbb{N}$. Intuitively, in our memory model, each index is thought as an entry point on some record cell containing several fields. Cells are either not allocated, or allocated with some record stored in. In a memory state $(s, h)$, the memory cell at index $i$ is *allocated* if $i \in \mathtt{dom}(h)$; in this case the stored record is $h(i) = \{l_1 \mapsto v_1, .., l_n \mapsto v_n\}$.

Note that the size of the information hold in a memory cell is not fixed, nor bounded. Our models could be more concrete considering labels as offsets and relying on pointer arithmetic. But for our purpose, it will be convenient to consider pointer arithmetic and recursive structures independently.

Expressions
$e ::= \mathtt{x} \mid \mathtt{null}$

Atomic formulae
$\pi ::= e = e' \mid e + i \overset{l}{\hookrightarrow} e$

State Formulae
$\mathcal{A} ::= \pi$
$\quad \mid \mathcal{A} * \mathcal{B} \mid \mathcal{A} \mathrel{-\!\!*} \mathcal{B} \mid \mathtt{emp}$ (spatial fragment)
$\quad \mid \mathcal{A} \wedge \mathcal{B} \mid \mathcal{A} \to \mathcal{A} \mid \top \mid \bot$ (classical fragment)

Satisfaction

| | |
|---|---|
| $(s,h) \models_{\text{SL}} e = e'$ | iff $[\![ e ]\!]_s = [\![ e' ]\!]_s$, with $[\![ \mathtt{x} ]\!]_s = s(\mathtt{x})$ and $[\![ \mathtt{null} ]\!]_s = nil$ |
| $(s,h) \models_{\text{SL}} e + i \overset{l}{\hookrightarrow} e$ | iff $[\![ e ]\!]_s \in \mathbb{N}$ and $[\![ e ]\!] + i \in \mathtt{dom}(h)$ and $h(s(\mathtt{x}) + i)(l) = [\![ e ]\!]_s$ |
| $(s,h) \models_{\text{SL}} \mathtt{emp}$ | iff $\mathtt{dom}(h) = \emptyset$ |
| $(s,h) \models_{\text{SL}} \mathcal{A}_1 * \mathcal{A}_2$ | iff $\exists\, h_1, h_2$ s.t. $h = h_1 * h_2$, $(s,h_1) \models_{\text{SL}} \mathcal{A}_1$ and $(s,h_2) \models_{\text{SL}} \mathcal{A}_2$ |
| $(s,h) \models_{\text{SL}} \mathcal{A}' \mathrel{-\!\!*} \mathcal{A}$ | iff for all $h'$, if $h \perp h'$ and $(s,h') \models_{\text{SL}} \mathcal{A}'$ then $(s, h * h') \models_{\text{SL}} \mathcal{A}$ |
| $(s,h) \models_{\text{SL}} \mathcal{A}_1 \wedge \mathcal{A}_2$ | iff $(s,h) \models_{\text{SL}} \mathcal{A}_1$ and $(s,h) \models_{\text{SL}} \mathcal{A}_2$ |
| $(s,h) \models_{\text{SL}} \mathcal{A}' \to \mathcal{A}$ | iff $(s,h) \models_{\text{SL}} \mathcal{A}'$ implies $(s,h) \models_{\text{SL}} \mathcal{A}$ |
| $(s,h) \models_{\text{SL}} \bot$ | never and $(s,h) \models_{\text{SL}} \top$ always |

**Table 1.** The syntax and semantics of SL with pointer arithmetic and records

*Separation Logic.* We now introduce the separation logic (SL), see e.g. [Rey02], on top of which we will define our temporal logic. The syntax of the logic is given in Table 1. As separation logic is about reasoning on disjoint heaps, and we need to define what we mean by "disjoint heaps" in our model. We choose to allow to reason at the granularity of record cells, so that a record cell cannot be decomposed in disjoint parts. Let $h_1$ and $h_2$ be two heaps; we say that $h_1$ and $h_2$ are disjoint, noted $h_1 \perp h_2$, if $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \emptyset$. The operation $h_1 * h_2$ is defined for disjoint heap as the *disjoint union* of the two partial functions. Semantics of formulae is defined by the satisfaction relation $\models_{\text{SL}}$ (see Table 1).

Formulae $\pi$ are *atomic formulae*, in which $\mathtt{x} + i \overset{l}{\hookrightarrow} e$ states that the value of the $l$ field of the record stored at the address pointed by $\mathtt{x}$ with offset $i$ is equal to the value of the expression $e$. The formula $e = e'$ states the equality between two expressions, and $\mathtt{emp}$ means that the current heap has no memory cell allocated (empty heap).

Formulae $\mathcal{A}$ of SL are called *state formulae*. A formula $\mathcal{A} * \mathcal{B}$ with the *separation conjunction* states that $\mathcal{A}$ holds on some portion of the memory heap and $\mathcal{B}$ holds on a disjoint portion. A formula $\mathcal{A} \mathrel{-\!\!*} \mathcal{B}$ states that the current heap, when extended with any disjoint heap verifying $\mathcal{A}$, will verify $\mathcal{B}$.

In the remainder, we focus on several specific fragments of this separation logic. We say that a formula is in the *record fragment* (RF) if all subformulae $\mathtt{x} + i \overset{l}{\hookrightarrow} e$ use $i = 0$ (we then write $\mathtt{x} \overset{l}{\hookrightarrow} e$). We say that a formula is in the *classical fragment* (CL) if it does not use the connectives $*, \mathrel{-\!\!*}$. Finally, we say that a formula is in the *list fragment* (LF) if it is in the classical fragment and all subformulae $\mathtt{x} + i \overset{l}{\hookrightarrow} e$ use $i = 0$ and $l = next$ (we simply write $\mathtt{x} \hookrightarrow e$). Clearly, the classical and record fragments are incomparable, while the list fragment is included in both of them.

Let us illustrate the expressive power of SL on examples. The formula $\neg\mathtt{emp} * \neg\mathtt{emp}$ means that at least two memory cells are allocated. The formula $\mathtt{x} \overset{l}{\mapsto} e$, defined as $\neg(\neg\mathtt{emp} * \neg\mathtt{emp}) \wedge \mathtt{x} \overset{l}{\hookrightarrow} e$, is the local version of $\mathtt{x} \overset{l}{\hookrightarrow} e$: $s, h \models_{\mathrm{SL}} \mathtt{x} \overset{l}{\mapsto} e$ iff $\mathsf{dom}(h) = \{s(\mathtt{x})\}$ and $h(s(\mathtt{x}))(l) = [\![\ e\ ]\!]_s$. The formula $(\mathtt{x} \overset{l}{\hookrightarrow} \mathtt{null}) {-\!\!*} \bot$ is satisfied at $(s_0, h_0)$ whenever there is no heap $h_1$ with $h_1 \bot h_0$ that allocates the variable $\mathtt{x}$ to *nil* on $l$ field, that is $\mathtt{x}$ is allocated in $h_0$.

$\mathcal{A}$ is valid iff for every memory state $(s, h)$, we have $(s, h) \models_{\mathrm{SL}} \mathcal{A}$ (written $\models_{\mathrm{SL}} \mathcal{A}$). Satisfiability is defined dually.

**Proposition 1.** *The model-checking, satisfiability and validity problems for* SL *are* PSPACE*-complete.*

PSPACE-hardness results are consequences of [CYO01b, Sect. 5.2].

### 2.2 Temporal extension

*Memory states sequences* Models of the logic $\mathrm{LTL}^{\mathrm{mem}}$ are $\omega$-sequences of memory states, that is elements in $(\mathcal{S} \times \mathcal{H})^\omega$ and they are understood as infinite computations of programs with pointer variables. In order to analyze computations from programs without destructive update, we shall also consider models with constant heap, that is elements in $\mathcal{S}^\omega \times \mathcal{H}$.

*The logic* $\mathrm{LTL}^{\mathrm{mem}}$. Formulae of $\mathrm{LTL}^{\mathrm{mem}}$ are defined in Table 2. Atomic formulae of $\mathrm{LTL}^{\mathrm{mem}}$ are state formulae from SL except that variables can be prefixed by the symbol "X". For instance, $\mathsf{X}\mathtt{x}$ is interpreted by the value of $\mathtt{x}$ at the next memory state. The temporal operators are the standard next-time operator $\mathsf{X}$ and until operator $\mathsf{U}$ present in LTL, see e.g. [SC85]. The satisfaction relation $\rho, t \models \Phi$ where $\rho$ is a model of $\mathrm{LTL}^{\mathrm{mem}}$, $t \in \mathbb{N}$ and $\Phi$ is a formula is also defined in Table 2. We use standard abbreviations such as $\Diamond\Phi$, $\Box\Phi$ ...

Given a fragment F of SL, we write SAT(F) to denote the satisfiability problem for F: given a temporal formula $\phi$ in $\mathrm{LTL}^{\mathrm{mem}}$ with state formulae built over F, is there a model $\rho$ such that $\rho, 0 \models \phi$? The variant problem in which we require that the model has a constant heap [resp. that the initial memory state is fixed, say $(s, h)$] is denoted by $\mathrm{SAT}^{ct}$(F) [resp. $\mathrm{SAT}_{init}$(F)]. The problem $\mathrm{SAT}^{ct}_{init}$(F) is defined analogously.

### 2.3 Programs with pointer variables

In this section, we define the model-checking problems for programs with pointer variables over $\mathrm{LTL}^{\mathrm{mem}}$ specifications. The set $\mathtt{I}$ of *instructions* used in the programs is defined by the grammar below:

$$\begin{aligned}\mathtt{instr} ::= {} & \mathtt{x} := \mathtt{y} \mid \mathtt{skip} \\ & \mid \mathtt{x} := \mathtt{y} \rightarrow l \mid \mathtt{x} \rightarrow l := \mathtt{y} \mid \mathtt{x} := \mathtt{cons}(l_1 : x_1, .., l_k : x_k) \mid \mathtt{free\ x} \\ & \mid \mathtt{x} := \mathtt{y}[i] \mid \mathtt{x}[i] := \mathtt{y} \mid \mathtt{x} = \mathtt{malloc}(i) \mid \mathtt{free\ x}, i\end{aligned}$$

| | | |
|---|---|---|
| Enriched expressions | $\eta ::=$ | $\mathtt{x} \mid \mathsf{X}\eta \mid \mathtt{null}$ |
| Atomic formulae | $\pi ::=$ | $\eta = \eta' \mid \eta + i \overset{l}{\hookrightarrow} \eta'$ |
| State formulae | $\mathcal{A} ::=$ | $\pi \mid \mathtt{emp} \mid \mathcal{A} * \mathcal{B} \mid \mathcal{A} \mathrel{-\!\!*} \mathcal{B} \mid \mathcal{A} \wedge \mathcal{B} \mid \mathcal{A} \rightarrow \mathcal{B} \mid \bot$ |
| Temporal formulae | $\Phi ::=$ | $\mathcal{A} \mid \mathsf{X}\Phi \mid \Phi \mathsf{U} \Phi' \mid \Phi \wedge \Phi' \mid \neg\Phi$ |

Semantics

$\rho, t \models \mathsf{X}\Phi$ iff $\rho, t+1 \models \Phi$.

$\rho, t \models \Phi \mathsf{U} \Phi'$ iff there is $t_1 \geq t$ s.t. $\rho, t_1 \models \Phi'$ and $\rho, t' \models \Phi$ for all $t' \in \{t, .., t_1 - 1\}$.

$\rho, t \models \Phi \wedge \Psi$ iff $\rho, t \models \Phi$ and $\rho, t \models \Psi$.

$\rho, t \models \neg\Phi$ iff $\rho, t \not\models \Phi$

$\rho, t \models \mathcal{A}$ iff $s'_t, h_t \models_{\mathrm{SL}} \mathcal{A}[\mathsf{X}^k \mathtt{x} \leftarrow (\mathtt{x}, k)]$ where $\rho = (s_t, h_t)_{t \geq 0}$ and $s'_t$ is defined by $s'_t(\langle \mathtt{x}, k \rangle) = s_{t+k}(\mathtt{x})$.

**Table 2.** The syntax and semantics of $\mathrm{LTL}^{\mathrm{mem}}$

---

The denotational semantics of an instruction `instr` is defined as a partial function $[\![ \mathtt{instr} ]\!] : \mathcal{S} \times \mathcal{H} \rightarrow \mathcal{S} \times \mathcal{H}$, undefined when the instruction would cause a memory violation. We list in Table 3 the formal denotational semantics of our instruction set. Boolean combinations of equalities between expressions are called guards and their set is denoted by $G$. A program is defined as a triple $(Q, \delta, q_I)$ such that $Q$ is a finite set of control states, $q_I$ is the initial state and $\delta$ is the transition relation, a subset of $Q \times G \times \mathtt{I} \times Q$. We use $q \xrightarrow{g, \mathtt{instr}} q'$ to denote a transition. We say that a program is *without destructive update* if transitions are labeled only with instructions of the form $\mathtt{x} := \mathtt{y}$, $\mathtt{x} := \mathtt{y} \rightarrow l$, and $\mathtt{x} := \mathtt{y}[i]$.

A program is a finite object whose interpretation can be viewed as an infinite-state system. More precisely, given a program $\mathtt{p} = (Q, \delta, q_I)$, the transition system $\mathcal{S}_{\mathtt{p}} = (S, \rightarrow)$ is defined as follows:

- $S = Q \times (\mathcal{S} \times \mathcal{H})$ (set of configurations),
- $(q, (s, h)) \rightarrow (q', (s', h'))$ iff there is a transition $q \xrightarrow{g, \mathtt{instr}} q' \in \delta$ such that $(s, h) \models g$ and $(s', h') = [\![ \mathtt{instr} ]\!](s, h)$.

Note that $\mathcal{S}_{\mathtt{p}}$ is not necessarily linear. A computation (or execution) of $\mathtt{p}$ is defined as an infinite path in $\mathcal{S}_{\mathtt{p}}$ starting with control state $q_I$. Computations of $\mathtt{p}$ can be viewed as $\mathrm{LTL}^{\mathrm{mem}}$ models, using propositional variables to encode the extra information about the control states.

Model-checking aims at checking properties expressible in $\mathrm{LTL}^{\mathrm{mem}}$ along computations of programs. To a logical fragment (SL, CL, RF, or LF), we associate a set of programs : all programs for SL and CL, programs with instructions having $i = 0$ for RF, and moreover with only the label *next* for LF. Given one of these fragments F of SL, we write MC(F) to denote the model-checking problem for F: given a temporal formula $\phi$ in $\mathrm{LTL}^{\mathrm{mem}}$ with state formulae built over F and a program $\mathtt{p}$ of the associated fragment, is there an infinite computation $\rho$

| | | |
|---|---|---|
| $[\![\ \mathtt{x} := \mathtt{y}\ ]\!]\ (s,h)$ | $\stackrel{\text{def}}{\equiv}$ | $(s[\mathtt{x} \mapsto s(\mathtt{y})], h).$ |
| $[\![\ \mathtt{x} := \mathtt{y} \to l\ ]\!]\ (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$ | $\stackrel{\text{def}}{\equiv}$ | $(s[\mathtt{x} \mapsto v], h * \{i \mapsto \{l \mapsto v, \dots\}\})$ with $s(\mathtt{y}) = i$ |
| $[\![\ \mathtt{x} \to l := \mathtt{y}\ ]\!]\ (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$ | $\stackrel{\text{def}}{\equiv}$ | $(s, h * \{i \mapsto \{l \mapsto s(\mathtt{y}), \dots\}\})$ with $s(\mathtt{x}) = i$ |
| $[\![\ \mathtt{x} := \mathtt{cons}(l_1 : \mathtt{x}_1, .., l_k : \mathtt{x}_k)\ ]\!]\ \ (s,h)$ | $\stackrel{\text{def}}{\equiv}$ | $\big(s[\mathtt{x} \mapsto i], h * \{i \mapsto \{l_1 \mapsto s(\mathtt{x}_1), \dots, l_k \mapsto s(\mathtt{x}_k)\}\}\big)$ with $i \notin \mathtt{dom}(h)$ |
| $[\![\ \mathtt{free}\ \mathtt{x}, l\ ]\!]\ (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$ | $\stackrel{\text{def}}{\equiv}$ | $(s, h * \{i \mapsto \{\dots\}\})$ with $s(\mathtt{x}) = i$ |
| $[\![\ \mathtt{skip}\ ]\!]\ (s,h)$ | $\stackrel{\text{def}}{\equiv}$ | $(s,h)$ |
| $[\![\ \mathtt{x} := \mathtt{y}[i]\ ]\!]\ (s, h * \{i + i' \mapsto \{next \mapsto v\}\})$ | $\stackrel{\text{def}}{\equiv}$ | $(s[\mathtt{x} \mapsto v], h * \{i \mapsto \{next \mapsto v\}\}))$ with $s(\mathtt{y}) = i'$ |
| $[\![\ \mathtt{x}[i] := \mathtt{y}\ ]\!]\ (s, h * \{i' + i \mapsto \{next \mapsto v\}\})$ | $\stackrel{\text{def}}{\equiv}$ | $(s, h * \{i + i' \mapsto \{next \mapsto s(\mathtt{y})\}\})$ with $s(\mathtt{x}) = i'$ |
| $[\![\ \mathtt{x} := \mathtt{malloc}(i)\ ]\!](s,h)$ | $\stackrel{\text{def}}{\equiv}$ | $\big(s[\mathtt{x} \mapsto i'], h * \{i' \mapsto \{next \mapsto nil\}\} * \dots * \{i' + i \mapsto \{next \mapsto nil\}\big)$ with $i', .., i' + i \notin \mathtt{dom}(h)$ |
| $[\![\ \mathtt{free}\ \mathtt{x}, i\ ]\!]\ (s, h * \{i' + i \mapsto f\})$ | $\stackrel{\text{def}}{\equiv}$ | $(s,h)$ with $s(\mathtt{x}) = i'$ |

**Table 3.** Semantics for instructions

of $\mathtt{p}$ such that $\rho, 0 \models \phi$ (which we write $\mathtt{p} \models \phi$)? The variant problem in which we require that the program is without destructive update [resp. that the initial memory state is fixed, say $(s,h)$] is denoted by $\mathrm{MC}^{ct}(\mathrm{F})$ [resp. $\mathrm{MC}_{init}(\mathrm{F})$]. The problem $\mathrm{MC}^{ct}_{init}(\mathrm{F})$ is defined analogously. We may write $\mathtt{p}, (s,h) \models \phi$ to emphasize what is the initial memory state.

All the model-checking and satisfiability problems defined above can be placed in $\Sigma^1_1$ in the analytical hierarchy. Additionally, all the above problems can easily be shown PSPACE-hard since they all generalize LTL satisfiability and model-checking [SC85].

## 3 Complexity results

Figure 1 contains a summary of the complexity results about fragments of $\mathrm{LTL}^{\mathrm{mem}}$, more details can be found in [BDL07]. However, let us present some ideas below. The decidability results for SAT(CL) and SAT(RF) are obtained thanks to a polynomial size abstraction of the memory states similar to symbolic heaps. A notable difference is that values of variables at successive states can be compared. This abstraction is used to define a finite alphabet on which is based a Büchi automaton recognizing a language of abstract runs. Unlike the standard

| | MC | $\mathrm{MC}^{ct}$ | $\mathrm{MC}^{ct}_{init}$ | $\mathrm{MC}_{init}$ | SAT | $\mathrm{SAT}^{ct}$ | $\mathrm{SAT}^{ct}_{init}$ |
|---|---|---|---|---|---|---|---|
| LF | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c. | $\Sigma^1_1$-c. | PSPACE-c. | $\Sigma^0_1$-c. | PSPACE-c. |
| CL and RF | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c. | $\Sigma^1_1$-c. | PSPACE-c. | $\Sigma^0_1$-c. | PSPACE-c. |
| SL$\backslash\{-\!\!*\}$ | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c |
| SL | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. |

**Fig. 1.** Complexity of reasoning about program with pointer variables

construction for LTL, we need to recognize sequences of abstract memory states that admit a concrete sequence of memory states. Decidability is shown when the set of abstract sequences is $\omega$-regular.

By way of example, let us also consider the undecidability proof for SAT(SL). The recurrence problem for non-deterministic Minsky machines can be reduced to it. The main difficulty is to be able to encode incrementations and decrementations of a variable $\mathtt{x}$. Observe that expressions of the form $\mathtt{x} = \mathtt{y} + 1$ do not belong to the logical language. The encoding will use in some essential way the interplay between the separation connectives and the temporal operators. We found two different ways to encode increment and decrement: using non-aliasing expressed by the separating conjunction, and using the precise pointing assertion $\mathtt{x} \overset{next}{\mapsto} \eta$ stating that the heap contains only one cell, in conjunction with the $-\!\!*$ operator.

$$\phi^{*}_{\mathtt{x}++} = (\mathsf{X}\mathtt{x} \overset{next}{\hookrightarrow} \mathtt{null} \wedge \mathtt{x}+1 \overset{next}{\hookrightarrow} \mathtt{null}) \wedge \neg(\mathsf{X}\mathtt{x} \overset{next}{\hookrightarrow} \mathtt{null} * \mathtt{x}+1 \overset{next}{\hookrightarrow} \mathtt{null})$$
$$\phi^{*}_{\mathtt{x}--} = (\mathsf{X}\mathtt{x}+1 \overset{next}{\hookrightarrow} \mathtt{null} \wedge \mathtt{x} \overset{next}{\hookrightarrow} \mathtt{null}) \wedge \neg(\mathsf{X}\mathtt{x}+1 \overset{next}{\hookrightarrow} \mathtt{null} * \mathtt{x} \overset{next}{\hookrightarrow} \mathtt{null})$$
$$\phi^{-\!\!*}_{\mathtt{x}++} = \mathtt{emp} \quad \wedge \quad \big((\mathsf{X}\mathtt{x} \overset{next}{\mapsto} \mathtt{null}) -\!\!* \mathtt{x}+1 \overset{next}{\mapsto} \mathtt{null}\big)$$
$$\phi^{-\!\!*}_{\mathtt{x}--} = \mathtt{emp} \quad \wedge \quad \big((\mathtt{x} \overset{next}{\mapsto} \mathtt{null}) -\!\!* \mathsf{X}\mathtt{x}+1 \overset{next}{\mapsto} \mathtt{null}\big)$$

The formulae based on the separating conjunction correctly express incrementation and decrementation when the cells at index $\mathtt{x}, \mathtt{x}+1, \mathtt{x}-1$ are allocated, whereas formulae based on the operator $-\!\!*$ do not need the same assumption.

So, when the heap is constant, only the second way to encode increment applies. Moreover, in the absence of the operator $-\!\!*$, we can show that the problem $\mathrm{SAT}^{ct}_{init}(\mathrm{SL}\backslash\{-\!\!*\})$ is in PSPACE, which contrasts with the $\Sigma^1_1$-hardness of $\mathrm{SAT}^{ct}_{init}(\mathrm{SL})$. This PSPACE result is obtained by reduction into $\mathrm{SAT}^{ct}_{init}(\mathrm{RF})$. In some essential way, we take into account that the heap is constant and that only subheaps can be considered thanks to the absence of the operator $-\!\!*$. This decidability result also implies that $\mathrm{SAT}^{ct}(\mathrm{SL})$ is in $\Sigma^0_1$.

As far as model-checking problems are concerned, the complexity results do not depend on fragments. For the problems of the form $\mathrm{MC}^{ct}_{init}$, the program can be abstracted as a finite-state automaton. Using standard results for LTL and Proposition 1, we get the PSPACE upper bound. For the problems of the form $\mathrm{MC}^{ct}$, halting problem for Minsky machines can be encoded by guessing the maximal value of counters for reaching the halting state, whence the $\Sigma^0_1$-hardness results. Finally, the problems of the form MC can encode infinite runs

of non-deterministic Minsky machines since the memory is not bounded, whence the $\Sigma_1^1$-hardness results.

## 4 Discussion

We provided above complexity results for reasoning tasks about $\mathrm{LTL}^{\mathrm{mem}}$. We are currently investigating issues about the expressive power of this logical formalism. Let us discuss few issues below.

First, the interest of model-checking programs with heap updates stems from early works on automata-based verification. Decision procedures are obtained at the cost of limitations: to define approximations as done in [YRSW03,DKR04] or to restrict the programming language, see e.g. [BFLS06]. However, with this approach, compositionality principles are lost which is a pity since they made the success of separation logic, as frame rule and composition rule.

Second, assuming that the heap is constant is subject to promising development. Indeed, it is then possible to define spatial operators at the same syntactic level as temporal operators, and write formulae as e.g. $(\mathtt{x} \hookrightarrow \mathsf{X}\mathtt{x}\mathsf{U}\mathtt{x} \hookrightarrow \mathtt{null}) * y \mapsto \mathtt{null}$. This might be a way to model modularity in model-checking programs without destructive updates, but there are other points of interest we will try to advocate now.

### 4.1 Recursion with local parameters

The constant heap semantics provides an original viewpoint for recursion with local parameters and local quantification. The problem of decision procedures in presence of recursive predicates has not yet completely satisfactory answers, as particular axiomatizations have been proposed for some standard recursive structures [CBO05], or incomplete, though apparently good in practice, methods of inference.

In order to be a bit more precise, let us consider the fragment of recursive separation logic where all recursive formulae are of the form:

$$(1) \quad \mu X(x_1,..,x_k).\ A(x_1,..,x_k) \vee \exists x'_1..x'_k.\ B(x_1,..,x_k,x'_1,..,x'_k) \wedge X(x'_1,..,x'_k)$$

This fragment is rich enough to express single lists, cyclic lists, and doubly-linked lists. However, we conjecture that it is not expressive enough for trees and DAGs.

We conjecture that deciding satisfiability in the fragment of recursive separation logic mentioned above reduces to $\mathrm{SAT}^{ct}(\mathrm{SL})$, and the model-checking problem reduces to $\mathrm{SAT}^{ct}_{init}$, considering that (1) can be rewritten as:

$$\big(B(x_1,..,x_k,\mathsf{X}x_1,..,\mathsf{X}x_k)\big)\ \mathsf{U}\ A(x_1,..,x_k).$$

In this perspective, our results could rise interesting decidability results for model-checking some of the recursive separation logic with local quantifiers. For satisfiability, we expect to define decidable fragments for $\mathrm{SAT}^{ct}(\mathrm{SL})$, for

instance considering the techniques for checking temporal properties of flat programs without destructive updates introduced in [FLS07]. Another interesting fragment of recursive separation logic is probably the one where recursion is guarded by the separation operator $*$, but we do not currently see how to treat it in the temporal logic perspective.

### 4.2 Programs as formulae

Let us speculate some more. We may take advantage of expressing programs as formulae in order to reduce model-checking to satisfiability, a known approach from [SC85]. For programs without destructive update, we have the following result.

**Proposition 2.** *Let* F *be a fragment of* SL *among* SL, CL, RF, *or* LF. *Then there is a logspace reduction from* $\mathrm{MC}^{ct}(\mathrm{F})$ *to* $\mathrm{SAT}^{ct}(\mathrm{F})$.

Intuitively, we translate instructions of the form $\mathtt{x} := \mathtt{y}$ into $\mathsf{X}\mathtt{x} = \mathtt{y}$, $\mathtt{x} := \mathtt{y} \to l$ into $\mathtt{y} \overset{l}{\hookrightarrow} \mathsf{X}\mathtt{x}$, and $\mathtt{x} := \mathtt{y}[i]$ into $\mathtt{y} + i \hookrightarrow \mathsf{X}\mathtt{x}$. Guards are translated accordingly. To translate the control of the program, we use special variables to encode the control state and define a formula that expresses the transition relation.

Moreover, we believe we can extend this result to programs with updates, but with a slightly different perspective. The constant heap semantics can be helpful to define the input-output relation of programs, even with destructive updates, provided some conditions on the way the program read and write over the memory are satisfied. To do so, we consider the extension of $\mathrm{LTL}^{\mathrm{mem}}$ with two predicates $\hookrightarrow_0$ and $\hookrightarrow_1$ instead of $\hookrightarrow$, and models are couples of state sequences with constant heap, that is tuples $\langle\ (s_i)_{i\geq 0}\ ,\ h_0\ ,\ h_1\rangle$. Let us define the input-output relation $R_P$ of a program $P$ as : for all $(s_0, h_0), (s_1, h_1)$, $(s_0, h_0) R_P (s_1, h_1)$ if there is a run of $P$ that starts with $(s_0, h_0)$ and ends with $(s_1, h_1)$. Then we conjecture that for an interesting class of programs, this relation is definable in $\mathrm{LTL}^{\mathrm{mem}}$ extended with $\hookrightarrow_0$ and $\hookrightarrow_1$. Basically, the encoding of the control of the program will be the same as for programs without destructive updates, but the encoding of the instructions will be different. For instance, $\mathtt{x} \to l := \mathtt{y}$ would be encoded by $\mathtt{y} \overset{l}{\hookrightarrow}_1 \mathsf{X}\mathtt{x}$, whereas $\mathtt{x} := \mathtt{y} \to l$ would be encoded as $(\mathsf{X}\mathtt{x}) \overset{l}{\hookrightarrow}_0 \mathtt{y}$

## 5 Conclusion

We have introduced a logic that combines both aspects of temporal logic and separation logic, and permits to express constraints between values at different instants. We defined several decision problems, and studied their decidability and complexity. One of the most important complexity results states the decidability of the satisfiability problem without pointer arithmetic or with pointer arithmetic but without separation operators. In those cases, the problems are in PSPACE, which is really tight considering that both SL and LTL satisfiability problems are PSPACE-complete. This is not completely expected since LTL with

simple Presburger constraints is undecidable even though both LTL and expressive fragments of Presburger arithmetic can be solved in PSPACE. Our result strongly relies on a faithful abstraction of memory states. As a future work, we plan to investigate decidable fragments of separation logic with recursion.

## References


[BCO05] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. *APLAS'05*, 3780:52–68, 2005.

[BDL07] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. In *LFCS'07*, LNCS. Springer, 2007. to appear.

[BFLS06] S. Bardin, A. Finkel, E. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. *5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, 2006.

[BIL04] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *SAS'04*, volume 3148 of *LNCS*, pages 344–360. Springer, 2004.

[CBO05] C. Calcagno, J. Berdine, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. 4111:115–137, 2005.

[CC00] H. Comon and V. Cortier. Flatness is not a weakness. *CSL'00*, 1862:262–276, 2000.

[CYO01a] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language. In *APLAS'01*, pages 289–300, 2001.

[CYO01b] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST&TCS'01*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.

[DD07] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

[DKR04] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 250–262. Springer, 2004.

[FLS07] A. Finkel, E. Lozes, and A. Sangnier. Towards model-checking pointer systems without destructive update. 2007. Under submission.

[GM05] D. Galmiche and D. Mery. Characterizing provability in BI's pointer logic through resource graphs. In *LPAR'05*, volume 3835 of *LNCS*, pages 459–473. Springer, 2005.

[JJKS97] J. Jensen, M. Jorgensen, N. Klarlund, and M. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI'97*, pages 226–236. ACM, 1997.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS'00*, pages 280–301, 2000.

[Loz04] E. Lozes. Separation logic preserves the expressive power of classical logic. In *2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'04)*, 2004.

[Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57. IEEE Computer Society Press, 1977.

[Rey02] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.

[SC85] A. Sistla and E. Clarke. The complexity of propositional linear temporal logic. *JACM*, 32(3):733–749, 1985.

[VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[YRSW03] E. Yahav, Th. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'03*, volume 2618 of *LNCS*, pages 204–22. Springer, 2003.