Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates

James Brotherston
University College London, UK
J.Brotherston@ucl.ac.uk

Nikos Gorogiannis
Middlesex University, UK
nikos.gorogiannis@gmail.com

Max Kanovich Reuben Rowe
University College London, UK
M.Kanovich@ucl.ac.uk
R.Rowe@ucl.ac.uk

Abstract

We investigate the *model checking* problem for symbolic-heap separation logic with user-defined inductive predicates, i.e., the problem of checking that a given stack-heap memory state satisfies a given formula in this language, as arises e.g. in software testing or runtime verification approaches.

First, we show that the problem is *decidable*; specifically, we present a bottom-up fixed point algorithm that decides the problem and runs in exponential time in the size of the problem instance.

Second, we show that, while model checking for the full language is EXPTIME-complete, the problem becomes NP-complete or PTIME-solvable when we impose natural syntactic restrictions on the schemata defining the inductive predicates. We additionally present NP and PTIME algorithms for these restricted fragments.

Finally, we report on the experimental performance of our procedures on a variety of specifications extracted from programs, exercising multiple combinations of syntactic restrictions.

Categories and Subject Descriptors D.2.4 [*Software / Program Verification*]: Model checking; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs, Assertions

Keywords Separation logic, model checking, inductive definitions, complexity, runtime verification, program testing.

1. Introduction

In modern computer science, *model checking* is most commonly considered to be the problem of deciding whether a given Kripke structure or transition system S — typically representing a program or system — satisfies, or is a *model* of, a given formula A of modal or temporal logic [18]; this property is usually written as $S \models A$. More generally, in mathematical logic, S might be a mathematical structure of virtually any kind and A a formula in some appropriate logic for such structures (see e.g. [20] for the cases of first-order and monadic second-order logic).

In this paper, we investigate the model checking problem as it arises in the setting of *separation logic* with user-defined inductive predicates. Separation logic is an established formalism for the verification of imperative pointer programs, comprising both an assertion language of formulas based on *bunched logic* and a Hoare-style system of triples manipulating the pre- and postconditions of programs [23, 29]. Given a program annotated with separation logic assertions, one can try to prove *statically* that each assertion holds at the appropriate program point; a long line of research in this area has resulted in a number of tools that are capable of doing this automatically at least *some* of the time for industrial code (see e.g. [7, 8, 14, 16, 19, 24, 28]). Alternatively, one might also try to test *dynamically* whether properties hold: simply execute the program and check whether each assertion is satisfied by the actual memory state of the program at that point (this is sometimes known as *run-time verification*). Such an approach obviously necessitates a method for deciding, for any memory state S and separation logic formula A, whether or not $S \models A$: a model checking problem. While this is straightforward for simple formulas, it becomes much more complicated when arbitrary user-defined inductive predicates, describing complex shape properties of the memory, are permitted.

Our first contribution is a general model checking procedure (in the sense above) for the most commonly considered *symbolic heap* fragment of separation logic, extended with a general schema for user-defined inductive predicates. Since our definition schema allows inductive predicates to denote possibly-empty heap memories, and any heap trivially decomposes into itself combined with the empty heap, a naive top-down approach based on backtracking search will generally fail to terminate. Instead, we employ a bottom-up approach based on computing the fixed point of all "submodels" of the original memory that satisfy one of the defined inductive predicates. The crucial insight is that, for any given model checking query, the witnesses for the existentially quantified variables can be chosen from a *fixed* set of values given in advance. Our algorithm decides the model checking problem for our logic, in (worst-case) exponential time in the query size. Indeed, we show that this problem is EXPTIME-complete.

In practice, however, it is often the case that the inductive predicate definitions encountered in verification practice fall within much more well-behaved fragments of our general inductive schemata. Our second main contribution is an analysis of the model checking problem in cases where the syntactic form of inductive definitions is restricted in various ways (e.g., when recursion is forbidden in cases where the heap might be empty). We show that, for different combinations of these restrictions, the model checking problem can become NP-complete or even PTIME-solvable; and, in such cases, we give concrete model checking algorithms that fall within the appropriate complexity bound.

Finally, we provide an implementation of our general model checking algorithm, and of our specialised algorithm for the polynomial-time fragment, within the CYCLIST theorem proving framework; this implementation is available online [1]. We evaluate their performance on a range of examples gathered from the separation logic community, as well as some hand-crafted examples.

Our experimental results seem to bear out that our model checking methods are practical for runtime verification applications when suitable syntactic restrictions are present, and for offline testing (such as in unit test suites) in the general case.

Related work. Runtime verification for separation logic was addressed first in [26], and then more recently in the Verifast tool [3]. In both cases, model checking works only for classes of recursive predicates that are restricted in various ways, and comes without any formal correctness claims or complexity bounds. As far as we know, the present paper is the first to specifically address model checking for symbolic-heap separation logic with general inductive predicates from a fully formal perspective. However, the logic itself has attracted considerable recent interest amongst the verification community. The aforementioned automated program verification tools based on separation logic [7, 8, 14, 16, 19, 24, 28] are all based on symbolic heaps, and increasingly targeted at verifying specifications involving user-defined rather than hard-coded predicates. Indeed, there are now even tools capable of *automatically* generating the definitions of inductive predicates needed for analysis [11, 25]. On the theoretical side, the satisfiability problem for our logic was recently shown decidable [10] and its entailment problem undecidable [4], although decidability results have been obtained for restricted classes of entailments [5, 22]. Alongside these theoretical developments, there are automated tools geared towards the proof [13, 17] and disproof [12] of entailments, as needed to support program verification.

The remainder of this paper is structured as follows. Section 2 introduces our fragment of separation logic, and Section 3 develops our general model checking procedure for it. This model checking problem is then shown to be EXPTIME-complete in Section 4. We present our restricted fragments in Section 5, and establish their various complexities in Section 6. Section 7 presents details of our implementation and experiments, and Section 8 concludes.

2. $\mathsf{SL}^{\mathsf{SH}}_{\mathsf{ID}}$: Symbolic-heap separation logic with inductively defined predicates

In this section we present our fragment $\mathsf{SL}^{\mathsf{SH}}_{\mathsf{ID}}$ of separation logic, which restricts the syntax of formulas to *symbolic heaps* as introduced in [5, 6], but allows arbitrary user-defined inductive predicates over these, as considered e.g. in [9, 10, 11, 22].

We often write vector notation to abbreviate tuples, e.g. $\mathbf{x}$ for $(x_1, \dots, x_m)$. We write proj_i for the ith projection function on tuples, and we often abuse notation by treating a tuple $\mathbf{x}$ as the set containing exactly the elements occurring in $\mathbf{x}$. If X and Y are sets, we write $X \# Y$ as a shorthand for $X \cap Y = \emptyset$.

2.1 Syntax

A *term* is either a *variable* in the infinite set Var, or the constant nil. We write x, y, z, etc. to range over variables, and t, u, etc. to range over terms. We assume a finite set $\mathbf{P} = \{P_1, \dots, P_n\}$ of *predicate symbols*, each with associated arity.

Definition 2.1 (Symbolic heap)**.** *Spatial formulas* F and *pure formulas* π are given by the following grammar:

$$F ::= \mathsf{emp} \mid x \mapsto \mathbf{t} \mid P_i\mathbf{t} \mid F * F \qquad \pi ::= t = t \mid t \neq t$$

where x ranges over variables, t over terms, P_i over predicate symbols and $\mathbf{t}$ over tuples of terms (matching the arity of P_i in $P_i\mathbf{t}$).

A *symbolic heap* is given by $\exists\mathbf{z}.\ \Pi : F$, where $\mathbf{z}$ is a tuple of (distinct) variables, F is a spatial formula and Π is a finite set of pure formulas. Whenever one of Π, F is empty, we omit the colon. We write $FV(A)$ for the set of free variables occurring in a symbolic heap A; by convention, the bound variable names in A are chosen disjoint from the free variables $FV(A)$.

Definition 2.2. An *inductive rule set* is a finite set of *inductive rules*, each of the form $A \Rightarrow P_i\mathbf{x}$, where A is a symbolic heap (called the *body* of the rule), $P_i\mathbf{x}$ a formula (called its *head*), $\mathbf{x}$ is a tuple of distinct variables and $FV(A) \subseteq \mathbf{x}$.

For convenience, we sometimes drop existential quantifiers from inductive rules $A \Rightarrow P_i\mathbf{x}$: in that case, any variables occurring in A but not in $\mathbf{x}$ are implicitly existentially quantified.

As usual, the inductive rules with P_i in their head should be read as exhaustive, disjunctive clauses of an inductive definition of P_i. The formal semantics appears below.

2.2 Semantics

We use a RAM model employing heaps of records. We assume a countably infinite set Val of *values* of which an infinite subset $\mathsf{Loc} \subset \mathsf{Val}$ are addressable *locations*; we insist on at least one non-addressable value $nil \in \mathsf{Val} \setminus \mathsf{Loc}$.

A *stack* is a function $s\colon \mathsf{Var} \to \mathsf{Val}$; we extend stacks to terms by setting $s(\mathsf{nil}) =_{\text{def}} nil$, and write $s[z \mapsto a]$ for the stack defined as s except that $s[z \mapsto a](z) = a$. We extend stacks pointwise to act on tuples of terms.

A *heap* is a partial function $h\colon \mathsf{Loc} \rightharpoonup_{\text{fin}} (\mathsf{Val}\ \mathsf{List})$ mapping finitely many locations to *records*, i.e. arbitrary-length tuples of values; we set $\mathrm{dom}(h)$ to be the set of locations on which h is defined, and e to be the empty heap that is undefined everywhere. We write $\circ$ for *composition* of domain-disjoint heaps: if h_1 and h_2 are heaps, then $h_1 \circ h_2$ is the union of h_1 and h_2 when $\mathrm{dom}(h_1) \# \mathrm{dom}(h_2)$, and undefined otherwise. Finally, we define the *cover* of a heap h as

$$\mathrm{cover}(h) =_{\text{def}} \mathrm{dom}(h) \cup \{b \in \mathsf{Val} \mid b \in h(a), a \in \mathrm{dom}(h)\},$$

i.e., the set of all values mentioned anywhere in h.

Definition 2.3. Let Φ be a fixed inductive rule set. Then we say that a stack-heap pair (s, h) is a *model* of a symbolic heap A if the relation $s, h \models_\Phi A$ holds, defined by structural induction on A:

$$\begin{array}{lcl}
s, h \models_\Phi t_1 = t_2 & \Leftrightarrow & s(t_1) = s(t_2) \\
s, h \models_\Phi t_1 \neq t_2 & \Leftrightarrow & s(t_1) \neq s(t_2) \\
s, h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models_\Phi x \mapsto \mathbf{t} & \Leftrightarrow & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{t}) \\
s, h \models_\Phi P_i\mathbf{t} & \Leftrightarrow & (s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi \\
s, h \models_\Phi F_1 * F_2 & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \circ h_2 \text{ and } s, h_1 \models_\Phi F_1 \\
 & & \quad \text{and } s, h_2 \models_\Phi F_2 \\
s, h \models_\Phi \exists\mathbf{z}.\ \Pi : F & \Leftrightarrow & \exists\mathbf{a} \in \mathsf{Val}^{|\mathbf{z}|}.\ s[\mathbf{z} \mapsto \mathbf{a}], h \models_\Phi \pi \text{ for all} \\
 & & \quad \pi \in \Pi \text{ and } s[\mathbf{z} \mapsto \mathbf{a}], h \models_\Phi F
\end{array}$$

where the semantics $[\![P_i]\!]^\Phi$ of the inductive predicate P_i under Φ is defined below.

If A contains no inductive predicates, then its satisfaction relation does not depend on the inductive rules Φ, and we typically write $s, h \models A$ to mean that $s, h \models_\Phi A$, for *any* Φ. Similarly, if Π is a set of pure formulas, we write $s \models \Pi$ to mean that $s, h \models_\Phi \Pi$ for any heap h and inductive rule set Φ.

The following definition gives the standard semantics of the inductive predicate symbols $\mathbf{P}$ according to a fixed inductive rule set Φ, i.e., as the least fixed point of an n-ary monotone operator constructed from Φ:

Definition 2.4. First, for each predicate $P_i \in \mathbf{P}$ with arity α_i say, we define $\tau_i = \mathrm{Pow}(\mathsf{Val}^{\alpha_i} \times \mathsf{Heap})$ (where $\mathrm{Pow}(-)$ is powerset). We also partition the rule set Φ into $\Phi_1, \dots, \Phi_n$, where Φ_i is the set of all inductive rules in Φ of the form $A \Rightarrow P_i\mathbf{x}$.

Now let each Φ_i be indexed by j (i.e., $\Phi_{i,j}$ is the j-th rule defining P_i), and for each inductive rule $\Phi_{i,j}$ of the form $\exists \mathbf{z}.\ \Pi : F \Rightarrow P_i\mathbf{x}$, we define the operator $\varphi_{i,j} : \tau_1 \times \ldots \times \tau_n \to \tau_i$ by:

$$\varphi_{i,j}(\mathbf{Y}) =_{\text{def}} \{(s(\mathbf{x}), h) \mid s, h \models_{\mathbf{Y}} \Pi : F\}$$

where $\mathbf{Y} \in \tau_1 \times \ldots \times \tau_n$ and $\models_{\mathbf{Y}}$ is the satisfaction relation defined above, except that $[\![P_i]\!]^{\mathbf{Y}} =_{\text{def}} \mathsf{proj}_i(\mathbf{Y})$. We then finally define the tuple $[\![\mathbf{P}]\!]^{\Phi} \in \tau_1 \times \ldots \times \tau_n$ by:

$$[\![\mathbf{P}]\!]^{\Phi} =_{\text{def}} \mu\mathbf{Y}.\ (\textstyle\bigcup_j \varphi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{Y}))$$

We write $[\![P_i]\!]^{\Phi}$ as an abbreviation for $\mathsf{proj}_i([\![\mathbf{P}]\!]^{\Phi})$.

Note that in computing $\varphi_{i,j}(\mathbf{Y})$ above, we strip the existential quantifiers $\exists\mathbf{z}$ from the body of the inductive rule $\Phi_{i,j}$, taking advantage of the convention that the existentially bound variables $\mathbf{z}$ are disjoint from the free variables $\mathbf{x}$ in $\Phi_{i,j}$.

3. A model checking algorithm for $\mathsf{SL}^{\mathsf{SH}}_{\mathsf{ID}}$

In this section we develop a decision procedure for the *model checking problem* in our logic $\mathsf{SL}^{\mathsf{SH}}_{\mathsf{ID}}$. Formally, this problem is stated as follows:

Model checking problem (MC). *Given an inductive rule set Φ, stack s, heap h and symbolic heap A, decide whether $s, h \models_{\Phi} A$.*

We observe that whether $s, h \models_{\Phi} A$ depends not on the entire (infinite) valuation of s but only on the values of s on $FV(A)$, which is finite; thus an instance of MC can be also viewed as finite. In fact, the problem can be simplified further by noting that, if we can solve the case when $A = P\mathbf{x}$, for P an inductive predicate, then the general case follows almost immediately:

Restricted model checking problem (RMC). *Given an inductive rule set Φ, tuple of values $\mathbf{a} \in \mathsf{Val}$, heap h and predicate symbol P, decide whether $(\mathbf{a}, h) \in [\![P]\!]^{\Phi}$.*

Proposition 3.1. MC *and* RMC *are (polynomially) equivalent.*

Proof. Given an instance $(\Phi, \mathbf{a}, h, P)$ of RMC, where $m = |\mathbf{a}|$ is the arity of P, we define the corresponding instance of MC to be $(\Phi, s, h, P\mathbf{x})$, where $\mathbf{x}$ is an m-tuple of distinct variables and s is any stack satisfying $s(\mathbf{x}) = \mathbf{a}$. Then, clearly,

$$s, h \models_{\Phi} P\mathbf{x} \ \Leftrightarrow\ (s(\mathbf{x}), h) \in [\![P]\!]^{\Phi} \ \Leftrightarrow\ (\mathbf{a}, h) \in [\![P]\!]^{\Phi} .$$

Conversely, let (Φ, s, h, A) be an instance of MC. Let $FV(A) = \mathbf{x}$, let Q be a predicate symbol of arity $|\mathbf{x}|$ not occurring in Φ, and define $\Phi' = \Phi \cup \{A \Rightarrow Q\mathbf{x}\}$. We then define the corresponding instance of RMC to be $(\Phi', s(\mathbf{x}), h, Q)$. By construction,

$$s, h \models_{\Phi} A \ \Leftrightarrow\ s, h \models_{\Phi'} Q\mathbf{x} \ \Leftrightarrow (s(\mathbf{x}), h) \in [\![Q]\!]^{\Phi'} .$$

Both reductions are trivially computable in polynomial time. □

Thus it suffices to formulate a decision procedure for the restricted problem RMC. Before diving into the details of our decision procedure, let us motivate its development by making two main observations about this problem.

1. One might be tempted to adopt a top-down approach to the problem by applying inductive rules backwards to $P\mathbf{x}$, obtaining smaller model-checking problems (in the size of the heap h) as recursive instances. Unfortunately, our general schema for inductive rules does not guarantee that the models of subformulas of the body of an inductive rule are strictly smaller than the models of the entire body, and so such an approach might fail to terminate. For example, suppose $((a, b), h) \in [\![P]\!]^{\Phi}$, and is generated by the inductive rule

   $$\exists z.\ Pxz * Pzy \Rightarrow Pxy.$$

   Then we know that, for some $c \in \mathsf{Val}$, we should have both $((a, c), h_1) \in [\![P]\!]^{\Phi}$ and $((c, b), h_2) \in [\![P]\!]^{\Phi}$, where $h = h_1 \circ h_2$; but we do *not* know that h_1, h_2 are smaller than h; either might be the empty heap e. (Indeed, it is quite possible that *all* of h, h_1, h_2 are empty.)

   Therefore, we adopt a bottom-up approach: we attempt to compute *all* tuples in $[\![P_i]\!]^{\Phi}$ that are "sub-models" of $(\mathbf{a}, h)$, by iteratively applying the inductive rules until we reach a fixed point. (In fact, we have to do this for all inductive predicates $\mathbf{P}$ simultaneously, in order to account for possible mutual dependency among them.) This process is guaranteed to terminate provided that there are only finitely many such sub-models.

2. The principal remaining difficulty is one of completeness: i.e., can we guarantee that *any* $(\mathbf{a}, h) \in [\![P]\!]^{\Phi}$ can be generated by applying the inductive rules in Φ to sub-models of $(\mathbf{a}, h)$? In fact this point is quite delicate, due to the presence of unrestricted existential quantification in our inductive rules. For example, suppose $(a, e) \in [\![P]\!]^{\Phi}$ is generated by the rule

   $$\exists z.\ z \neq x : Qxz \Rightarrow Px.$$

   Then we know that for some $b \in \mathsf{Val}$, we have $((a, b), e) \in [\![Q]\!]^{\Phi}$, where $b \neq a$ and b (trivially) does not appear in the empty heap e. Thus we must allow our sub-models to mention fresh, or "spare", values not mentioned in $\mathbf{a}$ or h.

   Fortunately, as we show (Lemma 3.7), only finitely many such spare values are needed for any given rule set Φ; these can be "recycled" as needed at each fresh application of an inductive rule in our fixed point computation.

We now formally define our fixed point construction computing all tuples $(\mathbf{b}, h') \in [\![\mathbf{P}]\!]^{\Phi}$ that are sub-models of a given $(\mathbf{a}, h)$, where "sub-model" means that $h' \subseteq h$ and $\mathbf{b}$ consists of values from $\mathbf{a}$, $\mathrm{cover}(h)$, the null value nil and a suitably chosen set of "spare" values.

Definition 3.2. Let Φ be an inductive rule set, $\mathbf{a} \subset \mathsf{Val}$ a tuple of values and h a heap. We define the set $\mathrm{Good}(\mathbf{a}, h)$ (of "good sub-model values for $(\mathbf{a}, h)$") by

$$\mathrm{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \mathrm{cover}(h).$$

Now let β be the maximum number of (free and bound) variable names appearing in any inductive rule in Φ. We define $\mathrm{Spare}_{\Phi}(\mathbf{a}, h)$ to be a set of β fresh values (from Val) that do not occur in $\mathrm{Good}(\mathbf{a}, h)$.

Definition 3.3. Let Φ be an inductive rule set, $\mathbf{a}$ a tuple of values and h a heap. For each inductive rule $\Phi_{i,j} \in \Phi$ of the form $\exists\mathbf{z}.\ \Pi : F \Rightarrow P_i\mathbf{x}$, we define an operator $\psi_{i,j} : \tau_1 \times \ldots \times \tau_n \to \tau_i$ as follows:

$$\psi_{i,j}(\mathbf{Y}) =_{\text{def}}$$
$$\left\{ (s(\mathbf{x}), h') \;\middle|\; \begin{array}{l} s, h' \models_{\mathbf{Y}} \Pi : F \text{ and } h' \subseteq h \text{ and} \\ s(\mathbf{x} \cup \mathbf{z}) \subseteq \mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h) \end{array} \right\}$$

where $\mathrm{Good}(\mathbf{a}, h)$ and $\mathrm{Spare}_{\Phi}(\mathbf{a}, h)$ are the sets of values given by Definition 3.2. It should be clear that each $\psi_{i,j}$ is a monotone operator. Thus we define the tuple $\mathbf{MC}^{\Phi}(\mathbf{a}, h) \in \tau_1 \times \ldots \times \tau_n$ by

$$\mathbf{MC}^{\Phi}(\mathbf{a}, h) =_{\text{def}} \mu\mathbf{Y}.\ (\textstyle\bigcup_j \psi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \psi_{n,j}(\mathbf{Y}))$$

We write $MC^{\Phi}_i(\mathbf{a}, h)$ as an abbreviation for $\mathsf{proj}_i(\mathbf{MC}^{\Phi}(\mathbf{a}, h))$.

For the remainder of this section, we shall assume a fixed instance of $\mathbf{MC}^\Phi(\mathbf{a}, h)$, given by choosing inductive rule set Φ, tuple of values $\mathbf{a}$ and heap h.

It should be fairly obvious by comparing the constructions in Definitions 2.4 and 3.3 that $MC_i^\Phi(\mathbf{a}, h)$ can only contain tuples that are already elements of $[\![P_i]\!]^\Phi$. The following lemma formalises that claim.

Lemma 3.4 (Soundness). $\mathbf{MC}^\Phi(\mathbf{a}, h) \subseteq [\![\mathbf{P}]\!]^\Phi$.

Proof. We proceed by fixed point induction on the tuple of sets $\mathbf{MC}^\Phi(\mathbf{a}, h)$. That is, we assume the inclusion $\mathbf{Y} \subseteq [\![\mathbf{P}]\!]^\Phi$ holds for some tuple of sets $\mathbf{Y} = (Y_1, \ldots, Y_n) \in \tau_1 \times \ldots \times \tau_n$, and must show it holds for $(\bigcup_j \psi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \psi_{n,j}(\mathbf{Y}))$. This means, assuming that $(\mathbf{b}, h') \in \psi_{i,j}(\mathbf{Y})$ for some inductive rule $\Phi_{i,j}$, we must show that $(\mathbf{b}, h') \in [\![P_i]\!]^\Phi$. Without loss of generality, we can consider $\Phi_{i,j}$ to be written in the form:

$$\exists \mathbf{z}.\, \Pi : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m} \Rightarrow P_i\mathbf{x}.$$

By construction of $\psi_{i,j}(\mathbf{Y})$, there is a stack s with $s(\mathbf{x}) = \mathbf{b}$ and

$$s, h' \models_{\mathbf{Y}} \Pi : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m}\ .$$

This means that $s \models \Pi$ and $h' = h_1 \circ \ldots \circ h_{k+m}$, where

$$\begin{array}{rll} & s, h_i \models_{\mathbf{Y}} y_i \mapsto \mathbf{u_i} & \text{for all } 1 \le i \le k, \\ \text{and} & s, h_{k+i} \models_{\mathbf{Y}} P_{j_i}\mathbf{x_i} & \text{for all } 1 \le i \le m. \end{array}$$

In particular, for any $1 \le i \le m$ we have $(s(\mathbf{x_i}), h_{k+i}) \in Y_{j_i}$ and thus, by the induction hypothesis, $(s(\mathbf{x_i}), h_{k+i}) \in [\![P_{j_i}]\!]^\Phi$. That is,

$$\begin{array}{rll} & s, h_i \models_\Phi y_i \mapsto \mathbf{u_i} & \text{for all } 1 \le i \le k, \\ \text{and} & s, h_{k+i} \models_\Phi P_{j_i}\mathbf{x_i} & \text{for all } 1 \le i \le m. \end{array}$$

Putting everything together, we have

$$s, h' \models_\Phi \Pi : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m}\ .$$

Therefore, $s, h' \models_\Phi P_i\mathbf{x}$, i.e., $(\mathbf{b}, h') \in [\![P_i]\!]^\Phi$ as required. □

Next, we must show that $MC_i^\Phi(\mathbf{a}, h)$ contains *all* $(\mathbf{b}, h') \in [\![P_i]\!]^\Phi$ that are "sub-models" of $(\mathbf{a}, h)$. To do this, we need to argue that for any element $(\mathbf{b}, h') \in [\![P_i]\!]^\Phi$ that is "almost a sub-model" of $(\mathbf{a}, h)$ in that $h' \subseteq h$ but $\mathbf{b}$ contains "bad" values (not in $\mathrm{Good}(\mathbf{a}, h)$ or $\mathrm{Spare}_\Phi(\mathbf{a}, h)$), there are corresponding sub-models in $MC_i^\Phi(\mathbf{a}, h)$, obtained by substituting "spare" values for "bad" ones. The following definition and technical lemma capture the relevant notion of substitution.

Definition 3.5. A finite partial function $\theta : \mathsf{Val} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$ is called a *substitution for* $\mathbf{MC}^\Phi(\mathbf{a}, h)$ if it is injective, and, for all $b \in \mathrm{dom}(\theta)$,

$$\begin{array}{ll} \theta(b) = b & \text{if } b \in \mathrm{Good}(\mathbf{a}, h), \text{ and} \\ \theta(b) \in \mathrm{Spare}_\Phi(\mathbf{a}, h) & \text{if } b \notin \mathrm{Good}(\mathbf{a}, h). \end{array}$$

Lemma 3.6. *Let θ be a substitution for $\mathbf{MC}^\Phi(\mathbf{a}, h)$ such that $\mathrm{dom}(\theta) \supseteq \mathbf{b}$, and $V \subset \mathsf{Val}$ a (finite) set of values with $\mathbf{b} \subseteq V$ and $|V| \le |\mathrm{Spare}_\Phi(\mathbf{a}, h)|$. Let $\mathrm{Spare}_\Phi(\mathbf{a}, h) \setminus \theta(\mathbf{b}) = \{d_1, \ldots, d_m\}$ and let $V \setminus (\mathbf{b} \cup \mathrm{Good}(\mathbf{a}, h)) = \{e_1, \ldots, e_k\}$. Then the function $\theta' : V \to \mathsf{Val}$, defined by*

$$\theta'(c) =_{def} \begin{cases} c & \text{if } c \in \mathrm{Good}(\mathbf{a}, h) \\ \theta(c) & \text{if } c \in \mathbf{b} \setminus \mathrm{Good}(\mathbf{a}, h) \\ d_i & \text{if } c = e_i \text{ for some } 1 \le i \le k, \end{cases}$$

is also a substitution for $\mathbf{MC}^\Phi(\mathbf{a}, h)$, with $\theta'(\mathbf{b}) = \theta(\mathbf{b})$.

Proof. For convenience, we abbreviate $\mathrm{Good}(\mathbf{a}, h)$ by $\mathcal{G}$ and $\mathrm{Spare}_\Phi(\mathbf{a}, h)$ by $\mathcal{S}$ in this proof.

First, since $V \subset \mathsf{Val}$ is finite, θ' is indeed a finite partial function $\mathsf{Val} \rightharpoonup_{\mathsf{fin}} \mathsf{Val}$. We argue that θ' is well-defined. The three cases of its definition above are non-overlapping by construction, the first case is trivially well-defined and the second case is well-defined since $\mathrm{dom}(\theta) \supseteq \mathbf{b}$. Thus we just need to show that the third case is well-defined, which means showing that $k \le m$, i.e.,

$$|V \setminus (\mathbf{b} \cup \mathcal{G})| \le |\mathcal{S} \setminus \theta(\mathbf{b})|\ .$$

Since θ is injective by assumption, $|\theta(\mathbf{b})| = |\mathbf{b}|$. Thus, as $|V| \le |\mathcal{S}|$, we have $|V| - |\mathbf{b}| \le |\mathcal{S}| - |\theta(\mathbf{b})|$. Then, using standard set theory, we have as required

$$\begin{array}{rcll} |V \setminus (\mathbf{b} \cup \mathcal{G})| & \le & |V \setminus \mathbf{b}| & \\ & = & |V| - |\mathbf{b}| & (\text{since } \mathbf{b} \subseteq V) \\ & \le & |\mathcal{S}| - |\theta(\mathbf{b})| & (\text{by the above}) \\ & \le & |\mathcal{S} \setminus \theta(\mathbf{b})|. & \end{array}$$

Next we argue that θ' is indeed a substitution for $\mathbf{MC}^\Phi(\mathbf{a}, h)$. It is easy to see that $\theta'(c) = c$ if $c \in \mathrm{Good}(\mathbf{a}, h)$ and $\theta'(c) \in \mathrm{Spare}_\Phi(\mathbf{a}, h)$ otherwise. We just need to show θ' is injective. This follows from the fact that the three definitional cases of θ' are given by three injective functions with pairwise disjoint ranges: $\mathcal{G}$, $\theta(\mathbf{b}) (\subseteq \mathcal{S})$ and $\mathcal{S} \setminus \theta(\mathbf{b})$, respectively. Hence if $\theta'(c_1) = \theta'(c_2)$ then both c_1 and c_2 fall into the same definitional case of θ', and so $c_1 = c_2$ by injectivity of the corresponding function. Thus indeed θ' is a substitution for $\mathbf{MC}^\Phi(\mathbf{a}, h)$ as required.

Finally, to see that $\theta'(\mathbf{b}) = \theta(\mathbf{b})$, observe that $\theta'(c) = \theta(c)$ immediately if $c \in \mathbf{b} \setminus \mathcal{G}$, and if $c \in \mathbf{b} \cap \mathcal{G}$ then $\theta'(c) = c = \theta(c)$, using the fact that θ is a substitution for $\mathbf{MC}^\Phi(\mathbf{a}, h)$. □

Lemma 3.7 (Completeness). *Let $(\mathbf{b}, h') \in [\![P_i]\!]^\Phi$ and $h' \subseteq h$, and let θ be a substitution for $\mathbf{MC}^\Phi(\mathbf{a}, h)$ with $\mathrm{dom}(\theta) \supseteq \mathbf{b}$. Then $(\theta(\mathbf{b}), h') \in MC_i^\Phi(\mathbf{a}, h)$.*

Proof. We proceed by fixed point induction on $[\![\mathbf{P}]\!]^\Phi$. That is, we assume the lemma holds for $\mathbf{Y} = (Y_1, \ldots, Y_n) \in \tau_1 \times \ldots \times \tau_n$, and must show it holds for $(\bigcup_j \varphi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{Y}))$. This means, assuming that $(\mathbf{b}, h') \in \varphi_{i,j}(\mathbf{Y})$ for some inductive rule $\Phi_{i,j}$, where $h' \subseteq h$ and we have a θ satisfying the conditions of the lemma, we must show that $(\theta(\mathbf{b}), h') \in MC_i^\Phi(\mathbf{a}, h)$.

Without loss of generality, we may consider $\Phi_{i,j}$ to be written in the form:

$$\exists \mathbf{z}.\, \Pi : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m} \Rightarrow P_i\mathbf{x}.$$

By construction of $\varphi_{i,j}$, we have a stack s such that $s(\mathbf{x}) = \mathbf{b}$ and

$$s, h' \models_{\mathbf{Y}} \Pi : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} * P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m}\ .$$

This means that $s \models \Pi$ and $h' = h_1 \circ \ldots \circ h_{k+m}$, where

$$\begin{array}{rll} & s, h_i \models y_i \mapsto \mathbf{u_i} & \text{for all } 1 \le i \le k, \\ \text{and} & s, h_{k+i} \models_{\mathbf{Y}} P_{j_i}\mathbf{x_i} & \text{for all } 1 \le i \le m. \end{array}$$

The latter two statements can be rewritten as follows:

$$\begin{array}{rl} \mathrm{dom}(h_i) = \{s(y_i)\} \text{ and } h_i(s(y_i)) = s(\mathbf{u_i}) & \text{for all } 1 \le i \le k, \\ \text{and } (s(\mathbf{x_i}), h_{k+i}) \in Y_{j_i} & \text{for all } 1 \le i \le m. \end{array}$$

Recall that $\mathbf{x}$ and $\mathbf{z}$ describe respectively the sets of all free and bound variables appearing in the inductive rule $\Phi_{i,j}$. We have that $s(\mathbf{x} \cup \mathbf{z}) \subset \mathsf{Val}$ is finite, and $\mathbf{b} = s(\mathbf{x}) \subseteq s(\mathbf{x} \cup \mathbf{z})$ and $|s(\mathbf{x} \cup \mathbf{z})| \le |\mathrm{Spare}_\Phi(\mathbf{a}, h)|$ by construction. Therefore, by taking $V = s(\mathbf{x} \cup \mathbf{z})$ in Lemma 3.6, and noting $\mathrm{dom}(\theta) \supseteq \mathbf{b}$ by assumption, we can obtain a substitution θ' for $\mathbf{MC}^\Phi(\mathbf{a}, h)$ with $\mathrm{dom}(\theta') = s(\mathbf{x} \cup \mathbf{z})$ and $\theta'(\mathbf{b}) = \theta(\mathbf{b})$.

Now, since θ' is injective, it is easy to see that $s \circ \theta' \models \Pi$ (where $\circ$ here denotes function composition). Additionally, since $s(y_i), s(\mathbf{u_i}) \subseteq \mathrm{cover}(h) \subseteq \mathrm{Good}(\mathbf{a}, h)$, we have by construction, for all $1 \le i \le k$,

$$\begin{array}{ll} & \mathrm{dom}(h_i) = \{\theta'(s(y_i))\} \text{ and } h_i(\theta'(s(y_i))) = \theta'(s(\mathbf{u_i})) \\ \text{i.e.} & s \circ \theta', h_i \models y_i \mapsto \mathbf{u_i}. \end{array}$$

Notice that, for any $1 \le i \le m$, we have both $h_{k+i} \subseteq h' \subseteq h$ and $\mathrm{dom}(\theta') \supseteq s(\mathbf{x_i})$. Therefore, by the induction hypothesis,

$$(\theta'(s(\mathbf{x_i})), h_{k+i})) \in MC_i^{\Phi}(\mathbf{a}, h) \quad \text{for all } 1 \le i \le m.$$

Putting everything together, we obtain

$$s \circ \theta', h' \models_{\mathbf{MC}^{\Phi}(\mathbf{a},h)} \begin{array}{l} \Pi : y_1 \mapsto \mathbf{u_1} * \ldots * y_k \mapsto \mathbf{u_k} \\ * \, P_{j_1}\mathbf{x_1} * \ldots * P_{j_m}\mathbf{x_m} \ . \end{array}$$

As $(s \circ \theta')(\mathbf{x} \cup \mathbf{z}) \subseteq \mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h)$ by construction, we obtain by the definition of $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ (Definition 3.3):

$$((s \circ \theta')(\mathbf{x}), h') \in MC_i^{\Phi}(\mathbf{a}, h) \ .$$

Finally, as $s(\mathbf{x}) = \mathbf{b}$ and θ' coincides with θ on $\mathbf{b}$, we have $(s \circ \theta')(\mathbf{x}) = \theta'(s(\mathbf{x})) = \theta(\mathbf{b})$. Thus we obtain as required

$$(\theta(\mathbf{b}), h') \in MC_i^{\Phi}(\mathbf{a}, h) \ . \qquad \square$$

Lemma 3.8. *For each* $1 \le i \le n$,

$$(\mathbf{a}, h) \in [\![P_i]\!]^{\Phi} \;\Leftrightarrow\; (\mathbf{a}, h) \in MC_i^{\Phi}(\mathbf{a}, h) \ .$$

Proof. The ($\Leftarrow$) direction follows directly from Lemma 3.4. The ($\Rightarrow$) direction follows from Lemma 3.7 by taking $(\mathbf{b}, h')$ there to be $(\mathbf{a}, h)$, and θ to be the identity function on $\mathbf{a}$ (noting that $\mathbf{a} \subseteq \mathrm{Good}(\mathbf{a}, h)$, so this is trivially a substitution in the sense of Definition 3.5). □

Lemma 3.9. $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ *is finite and computable.*

Proof. By construction (Definition 3.3), $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ can only contain tuples of the form $(\mathbf{b}, h')$, where $h' \subseteq h$ and $\mathbf{b}$ is a finite tuple of values, drawn from the finite set $\mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h)$. As the heap h is also finite, each such $(\mathbf{b}, h')$ is a finite object and there can be only finitely many of them. Hence $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ is finite.

To see that $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ is computable, observe that it is defined as the least fixed point of a monotone operator. It is well known that this least fixed point can be approached iteratively in *approximant* stages, starting from the n-tuple $(\emptyset, \ldots, \emptyset)$. Since $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ is finite, there can be only finitely many such approximants. To see that each one is computable, it suffices to show that any $\psi_{i,j}(\mathbf{Y})$ is computable, given that $\mathbf{Y} \in \tau_1 \times \ldots \times \tau_n$ is computable and the inductive rule $\Phi_{i,j}$ is of the form $\exists \mathbf{z}.\ \Pi : F \Rightarrow P_i\mathbf{x}$, say. This is quite clear: First, there are only finitely many membership candidates $(\mathbf{b}, h')$ with $h' \subseteq h$ and $\mathbf{b} \subseteq \mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h)$. Second, since whether $s, h' \models_{\mathbf{Y}} \Pi : F$ depends only on the values s assigns to the variables appearing in $\Pi : F$, for each candidate $(\mathbf{b}, h')$ it suffices to pick an *arbitrary* stack s with $s(\mathbf{x}) = \mathbf{b}$ and $s(\mathbf{z}) \subseteq \mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h)$. Finally, for any such s, h' and computable $\mathbf{Y}$ it is straightforward to decide whether $s, h' \models_{\mathbf{Y}} \Pi : F$. □

Theorem 3.10. *The model checking problem* MC *is decidable. That is, for any stack s, heap h, inductive rule set Φ and symbolic heap A, it is decidable whether $s, h \models_{\Phi} A$.*

Proof. By Proposition 3.1, it suffices to show that RMC is decidable. Let $(\Phi, \mathbf{a}, h, P_i)$ be an instance of RMC. By Lemma 3.8, deciding whether $(\mathbf{a}, h) \in [\![P_i]\!]^{\Phi}$ is equivalent to deciding whether $(\mathbf{a}, h) \in MC_i^{\Phi}(\mathbf{a}, h)$. By Lemma 3.9, we have that $MC_i^{\Phi}(\mathbf{a}, h) = \mathsf{proj}_i(\mathbf{MC}^{\Phi}(\mathbf{a}, h))$ is a finite and computable set. Hence it is decidable whether $(\mathbf{a}, h) \in MC_i^{\Phi}(\mathbf{a}, h)$. □

We conclude this section by deducing some immediate consequences of Theorem 3.10 for the *entailment* problem in $\mathsf{SL}_{\mathsf{ID}}^{\mathsf{SH}}$.

Definition 3.11. Given an inductive rule set Φ and symbolic heaps A, B, we say the entailment $A \vdash_{\Phi} B$ is *valid* if $s, h \models_{\Phi} A$ implies $s, h \models_{\Phi} B$ for all stacks s and heaps h, and *invalid* otherwise.

It was shown in [4] that the set of valid sequents is not recursively enumerable (and, therefore, validity is, in general, undecidable). However, it does turn out to be *co*-recursively enumerable.

Corollary 3.12. *For any entailment $A \vdash_{\Phi} B$, the set of its* countermodels, *$\{(s, h) \mid s, h \models_{\Phi} A$ and $s, h \not\models_{\Phi} B\}$, is recursively enumerable.*

Corollary 3.13. *For any inductive rule set Φ, the set of invalid entailments over Φ is recursively enumerable.*

Remark 3.14. Since *satisfiability* for our logic is known to be decidable [10], one might imagine that we can simply reduce model checking to satisfiability: encode the state (s, h) as a formula $\gamma(s, h)$, so that $s, h \models_{\Phi} A$ iff $\gamma(s, h) \wedge A$ is satisfiable. Unfortunately, this does not work for our logic since standard conjunction $\wedge$ between arbitrary symbolic heaps is not permitted.

4. Complexity of general model checking

In this section we investigate the computational complexity of the general model checking problem MC, as described in the previous section. Specifically, we show that MC is EXPTIME-complete, and is still NP-hard in the size of the heap when the underlying inductive rule set is fixed in advance.

In the following, we write $\|o\|$ to denote the length of (some reasonable) encoding of a finite mathematical object o.

Lemma 4.1. MC *is* EXPTIME-*hard.*

Proof. By Proposition 3.1, it suffices to show that the restricted model checking problem, RMC, is EXPTIME-hard. This is by reduction from the following special case of the *satisfiability* problem for $\mathsf{SL}_{\mathsf{ID}}^{\mathsf{SH}}$, which was shown to be EXPTIME-hard in [10]: given an inductive rule set Φ containing no occurrences of $\mapsto$, and a predicate symbol P from Φ of arity 0, decide whether there exists a model s, h such that $s, h \models_{\Phi} P$. Since Φ contains no occurrences of $\mapsto$ and P no free variables, this means deciding whether $e \in [\![P]\!]^{\Phi}$ (recall e is the empty heap). Thus, given any instance (Φ, P) of the above problem, the corresponding instance of RMC is simply given by $(\Phi, (), e, P)$. □

Lemma 4.2. MC $\in$ EXPTIME.

Proof. By Proposition 3.1, it suffices to show RMC $\in$ EXPTIME. By Lemma 3.8, deciding an given instance $I = (\Phi, \mathbf{a}, h, P_i)$ of RMC can be done by computing $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ and checking whether $(\mathbf{a}, h) \in \mathsf{proj}_i(\mathbf{MC}^{\Phi}(\mathbf{a}, h))$. Thus it suffices to show that $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ can be computed in time exponential in $m =_{\mathrm{def}} \|I\|$.

Recall that $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ is obtained by a fixed point construction of a monotone operator (Definition 3.3):

$$\mathbf{MC}^{\Phi}(\mathbf{a}, h) =_{\mathrm{def}} \mu\mathbf{Y}.\ \left(\textstyle\bigcup_j \psi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \psi_{n,j}(\mathbf{Y})\right)$$

This least fixed point can be approached iteratively from below, starting from $(\emptyset, \ldots, \emptyset)$. Writing $N = |\mathbf{MC}^{\Phi}(\mathbf{a}, h)|$, this process will reach a fixed point in at most N iterations. Let T be the maximum number of polynomial-time steps required to compute any $\psi_{i,j}(\mathbf{Y})$, given the earlier fixed point approximant $\mathbf{Y}$. Since each iteration involves the computation of $\psi_{i,j}(\mathbf{Y})$ for every inductive rule $\Phi_{i,j} \in \Phi$, it is clear that computing $\mathbf{MC}^{\Phi}(\mathbf{a}, h)$ requires $N \cdot |\Phi| \cdot T$ polynomial-time steps.

Now, to obtain an upper bound for N, observe that by construction $|\mathbf{MC}^{\Phi}(\mathbf{a},h)|$ contains only pairs of the form $(\mathbf{b},h')$ such that $h' \subseteq h$, the length of $\mathbf{b}$ is bounded by the maximum arity of any predicate, which is bounded by $\|\Phi\|$, and $\mathbf{b} \subseteq \mathrm{Good}(\mathbf{a},h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a},h)$, which is bounded by $\|a\|+\|h\|+1+\|\Phi\| = \|I\|+1$ (the extra 1 comes from *nil*). Therefore, we obtain

$$N \leq (\|I\|+1)^{\|\Phi\|} \cdot 2^{\|h\|} = O\left(2^{\mathrm{poly}(m)}\right)$$

Next, we obtain an upper bound for T. Given $\mathbf{Y}$ and the inductive rule $\Phi_{i,j}$ of the form $\exists \mathbf{z}.\Pi : F \Rightarrow P_i\mathbf{x}$, say, it clearly suffices, for any "candidate" $(\mathbf{b},h')$ of the above form, to decide whether or not $(\mathbf{b},h') \in \psi_{i,j}(\mathbf{Y})$. This means checking whether $h' \subseteq h$ and, for every valuation of the variables $\mathbf{x} \cup \mathbf{z}$ into $\mathrm{Good}(\mathbf{a},h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a},h)$, checking whether $s(\mathbf{x}) = \mathbf{b}$ and $s,h' \models_{\mathbf{Y}} \Pi : F$ (where s is any stack obtained by extending the chosen valuation). The heap inclusion check can be done in polynomial time, and the number of possible valuations is (easily) bounded by N. To check $s,h' \models_{\mathbf{Y}} \Pi : F$ we might need to consider every possible division of h' into a number of "subheaps" bounded by the maximum number of $*$s in any rule, in turn bounded by $\|\Phi\|$ (as per the proofs of Lemmas 3.4 and 3.7). There are at most $2^{\|h\|\cdot\|\Phi\|}$ such combinations. Finally, as $\mathbf{Y}$ might contain up to N elements, checking whether a chosen division of h' satisfies F with respect to $\mathbf{Y}$ might take up to N steps. All other checks are polynomial, so we obtain

$$T \leq N \cdot N \cdot N \cdot 2^{\|h\|\cdot\|\Phi\|} = O\left(2^{\mathrm{poly}(m)}\right)$$

Therefore, altogether, the computation of $\mathbf{MC}^{\Phi}(\mathbf{a},h)$ requires at most $N \cdot |\Phi| \cdot T = O(2^{\mathrm{poly}(m)})$ polynomial-time steps. □

Theorem 4.3. MC *is* EXPTIME*-complete.*

Proof. Immediate by Lemmas 4.1 and 4.2. □

Typically, in program verification applications, the definitions of the inductive predicates are fixed in advance. Thus, it is also of interest to know how the complexity of MC varies in the size of the heap h over a fixed inductive rule set Φ.

Proposition 4.4. MC *is* NP*-hard in* $\|h\|$.

Proof. By Proposition 3.1, it suffices to show RMC is NP-hard in $\|h\|$. We exhibit a polynomial-time reduction from the following *triangle partition problem*, known to be NP-complete [21]: given a graph $G = (V,E)$ with $|V| = 3q$ for some $q > 0$, decide whether there is a partition of G into triangles.

First, we fix the following inductive rule set Φ_{PT}:

$$x \mapsto nil \Rightarrow V(x)$$

$$e \mapsto (x,y) * e' \mapsto (y,x) \Rightarrow E(x,y)$$

$$V(x) * V(y) * V(z) * E(x,y) * E(y,z) * E(z,x) \Rightarrow T$$

$$E(x,y) * J \Rightarrow J \qquad \mathsf{emp} \Rightarrow J \qquad T * P \Rightarrow P \qquad J \Rightarrow P$$

Now we give the reduction. For any instance $G = (V,E)$ of the above triangle partition problem, first write $V = \{v_1,\ldots,v_n\}$ and $E = \{e_1,\ldots,e_m\}$. We let $a_1,\ldots,a_n$ and $b_1,\ldots,b_{2m}$ be distinct addresses in Loc, and define a heap h_G, with $\mathrm{dom}(h_G) = \{a_1,\ldots,a_m,b_1,\ldots,b_{2m}\}$, as follows:

$$\begin{array}{rll} h_G(a_i) = nil & \text{for } 1 \leq i \leq n, \\ h_G(b_{2k}) = (a_i,a_j) & \text{for } 1 \leq k \leq 2m \text{ and } e_k = \{v_i,v_j\}, \\ h_G(b_{2k+1}) = (a_j,a_i) & \text{for } 1 \leq k \leq 2m \text{ and } e_k = \{v_i,v_j\}. \end{array}$$

The required instance of RMC is then given by $(\Phi_{\mathrm{PT}},(),h_G,P)$ (note that Φ_{PT} and P are *fixed* for any G). Clearly it is polynomial-time computable. For correctness, we need to show that G has a partition into triangles if and only if $h_G \in [\![P]\!]^{\Phi_{\mathrm{PT}}}$. This follows from the following easy observations, for any subheap h of h_G and values c,d (the formal details are easily reconstructed):

- $(c,h) \in [\![V]\!]^{\Phi_{\mathrm{PT}}}$ iff (c,h) exactly represents a vertex in G;
- $((c,d),h) \in [\![E]\!]^{\Phi_{\mathrm{PT}}}$ iff $((c,d),h)$ exactly represents an (undirected) edge in G;
- $h \in [\![T]\!]^{\Phi_{\mathrm{PT}}}$ iff h exactly represents a triangle in G;
- $h \in [\![J]\!]^{\Phi_{\mathrm{PT}}}$ iff h exactly represents some collection of edges in G;
- $h \in [\![P]\!]^{\Phi_{\mathrm{PT}}}$ iff h exactly represents a collection of non-overlapping triangles in G covering all vertices in G, plus a collection of "leftover" edges from G, i.e. iff G has a partition into triangles. □

5. Restricted fragments

According to Theorem 4.3, the general model checking problem is EXPTIME-complete. In practice, however, one frequently encounters definition schemas that are more restrictive than our general schema. Here and in the next section, we investigate the computational complexity of model checking when various natural syntactic restrictions are imposed on predicate definitions. Informally, the restrictions we consider are the following:

MEM**:** "*Memory-consuming*" rules, which only permit recursion in the presence of explicit non-empty memory.

CV**:** "*Constructively valued*" rules, in which the values of all variables occurring in a rule body are uniquely determined by the values of variables occurring in its head, together with the heap.

DET**:** "*Deterministic*" rules, in which the pure (dis)equality conditions in the rules for a predicate P are mutually exclusive.

Arity**:** The maximum arity of any predicate is fixed in advance.

Importantly, each of the above restrictions can be described in a clear syntactical way. The restrictions DET and CV have appeared previously in the literature, in various guises (e.g. [16, 3]). Together, as we will show in Section 6.5, they imply *precision*, the notion that a formula unambiguously circumscribes the part of the heap on which it is true [27]. The restriction MEM, as far as we know, is novel, but is seemingly crucial in reducing the complexity of model checking from EXPTIME down to NP or PTIME.

In Section 6 we show that the general EXPTIME can be reduced to PSPACE to NP or even PTIME for the fragments defined by different combinations of the above restrictions. The following table summarises our results:

		CV	DET	CV + DET
non-MEM	EXPTIME	EXPTIME	EXPTIME	≥ PSPACE
MEM	NP	NP	NP	PTIME

Remark 5.1. For each of the combinations MEM, MEM + CV, MEM + DET, their NP-completeness holds even when the arity of the predicates involved is *fixed in advance*.

In contrast, notwithstanding EXPTIME-hardness, for the fragment defined only by *non-memory-consuming rules*, model checking can be resolved in PTIME, but the degree of the polynomial is proportional to the maximal arity of the predicates involved.

We now formally introduce the restrictions MEM, CV and DET.

Definition 5.2 (MEM)**.** An inductive rule set is said to be *memory-consuming* (a.k.a. "in MEM") if every rule in it is of the form

$$\begin{array}{rl} & \Pi : \mathsf{emp} \;\Rightarrow\; P\mathbf{x}, \\ \text{or} & \exists \mathbf{z}.\,\Pi : F * x \mapsto \mathbf{t} \;\Rightarrow\; P\mathbf{x}\,. \end{array}$$

In practice, most predicate definitions in the literature fall into MEM: one or more pointers are "consumed" when recursing.

Example 5.3. The following definitions of binary predicates ls, defining possibly-cyclic list segments by head recursion, and rls, defining possibly-cyclic list segments by tail recursion, are both in MEM. Both definitions "consume" a pointer when recursing.

$$\begin{array}{rl} x = y \colon \mathsf{emp} \;\Rightarrow\; ls(x,y) & \\ \exists z.\, x \mapsto z \,*\, ls(z,y) \;\Rightarrow\; ls(x,y) & \end{array} \qquad (1)$$

and

$$\begin{array}{rl} x = y \colon \mathsf{emp} \;\Rightarrow\; rls(x,y) & \\ \exists z.\, x \neq y \colon\; rls(x,z) \,*\, z \mapsto y \;\Rightarrow\; rls(x,y) & \end{array} \qquad (2)$$

Definition 5.4 (CV)**.** A variable z occurring in an inductive rule $\exists \mathbf{y}.\, \Pi : F \Rightarrow P_i\mathbf{x}$ is said to be *constructively valued* in that rule if: (a) $z \in \mathbf{x}$, or (b) there is a variable w also occurring in the rule such that w is constructively valued, and either

- $\Pi \models z = w$, or,
- $w \mapsto \mathbf{u}$ is a subformula of F and $z \in \mathbf{u}$.

An inductive rule is constructively valued if all its variables are, and an inductive rule set is constructively valued (a.k.a. "in CV") if all its rules are.

Example 5.5. The existentially quantified variable z is constructively valued in the definition of ls in Example 5.3 (1), but not in the definition of rls (2).

Definition 5.6 (DET)**.** A predicate P_i is said to be *deterministic* (in an inductive rule set Φ) if for any two distinct rules of the form

$$\exists \mathbf{z}.\, \Pi : F \Rightarrow P_i\mathbf{x} \quad \text{and} \quad \exists \mathbf{z}'.\, \Pi' : F' \Rightarrow P_i\mathbf{x}\ ,$$

there exists no stack s such that $s, e \models \exists \mathbf{z}.\, (\Pi : \mathsf{emp})$ and $s, e \models \exists \mathbf{z}'.\, (\Pi' : \mathsf{emp})$. An inductive rule set Φ is deterministic (a.k.a. "in DET") if all predicates defined in Φ are deterministic.

We note that whether Φ is deterministic is decidable in polynomial time via a simple procedure that eliminates pure sub-formulas employing quantified variables [10].

Example 5.7. In Example 5.3, the definition of rls (2) is deterministic, but the definition of ls (1) is not.

6. Complexity of naturally restricted fragments

In this section, we investigate the computational complexity of model checking for different combinations of the restrictions MEM, CV and DET introduced in the previous section.

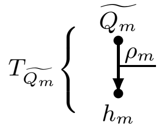
The main technical tool underpinning the following complexity results is the notion of an *unfolded inductive tree*. The idea is simple: in order to show that $(\mathbf{a}, h) \in [\![P]\!]^{\Phi}$, we repeatedly unfold the rules from Φ backwards (from head to body), instantiating variables with values and matching $\mapsto$ assertions with pointers in h as we go according to the rule constraints.

Definition 6.1. For the sake of brevity, given an elementary formula $Q(x_1, .., x_n)$, we write $\widetilde{Q}$ to denote a statement $Q(a_1, .., a_n)$ obtained by instantiating the variables in Q with values and heap pointers in the obvious way. For example, $\widetilde{x} \mapsto \widetilde{y}$ represents the one-cell heap that contains $\widetilde{y}$ at location $\widetilde{x}$. Then we specify the unfolded inductive tree by induction (see Figure 1):

(a) Let $\widetilde{Q_m}$ be generated by an instantiated rule ρ_m of the form

$$\widetilde{\Pi_m} : h_m \;\Rightarrow\; \widetilde{Q_m}$$

Then we make a tree, $T_{\widetilde{Q_m}}$, consisting of one edge labelled by ρ_m; the root is labelled by $\widetilde{Q_m}$, the leaf by h_m.



(b) Suppose that an instantiated rule ρ of the form

$$\widetilde{\Pi} : \widetilde{Q_0} \,*\, \widetilde{Q_1} \,*\, \cdots \,*\, \widetilde{Q_m} \;\Rightarrow\; \widetilde{R}$$

generates $\widetilde{R}$, and $T_{\widetilde{Q_0}}, T_{\widetilde{Q_1}}, \ldots, T_{\widetilde{Q_m}}$ are inductive trees having been already constructed for $\widetilde{Q_0}, \widetilde{Q_1}, \ldots, \widetilde{Q_m}$, resp.
Then we make a tree, $T_{\widetilde{R}}$, by taking a new root with $m{+}1$ outgoing edges *all* labelled by ρ, labelling the root by $\widetilde{R}$, and connecting our root with the roots of $T_{\widetilde{Q_0}}, T_{\widetilde{Q_1}}, \ldots, T_{\widetilde{Q_m}}$, resp.

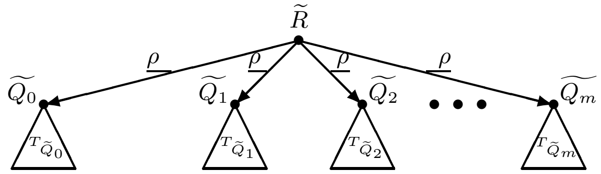


Figure 1. An unfolded inductive tree for $\widetilde{R}$.

Proposition 6.2. *The restricted model checking problem* RMC *can be solved by an exhaustive search for an unfolded inductive tree for the query* $(\mathbf{a}, h) \in [\![P]\!]^{\Phi}$*, where the values for instantiated existential variables are drawn from the set* $\mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h)$*, as per Definition 3.2.*

Proof. (Sketch) The soundness of the approach is obvious. *Termination* follows from the fact that the range of permissible values is finite, and *completeness* from the results in Section 3 which show that it suffices to confine our attention to values drawn from the polynomial-size set $\mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_{\Phi}(\mathbf{a}, h)$. □

However, the above procedure might still generate an exponential number of leaves labelled by the empty heap e:

Example 6.3. Let Φ_n be the set of inductive rules (for $1 \leq j \leq n$):

$$\begin{array}{rl} P_1 \,*\, P_2 \Rightarrow P_0, & \\ P_{2j+1} \,*\, P_{2j+2} \Rightarrow P_{2j-1}, & P_{2j+1} \,*\, P_{2j+2} \Rightarrow P_{2j}, \\ \mathsf{emp} \Rightarrow P_{2n+1}, & \mathsf{emp} \Rightarrow P_{2n+2}. \end{array}$$

Then any unfolded inductive tree for the query $s, e \models_{\Phi_n} P_0$ has 2^{n+1} leaves labelled by e. □

Nevertheless, in the case of MEM, we are able to reduce EXPTIME to NP by proving that the number of leaves labelled by e can be bounded by $|\mathrm{dom}(h)|$.

6.1 An upper NP-bound for MEM

Here, we show that, when we restrict to memory-consuming rules (MEM), model checking becomes an NP problem.

Theorem 6.4. *We can design an* NP *procedure to determine, given a set of* memory-consuming *inductive rules* Φ*, tuple of values* $\mathbf{a}$*, heap* h *and predicate symbol* P*, whether* $(\mathbf{a}, h) \in [\![P]\!]^{\Phi}$.

Proof. (Sketch) Taking into account the bounds provided by Section 3, we look for an unfolded inductive tree such that within each rule instance ρ from Φ used in the tree all *values* are taken only from a set of polynomial size, which is *fixed in advance*.

To provide an NP procedure, it suffices to prove Lemma 6.5 and Lemma 6.6 below. The crucial issue is that, in contrast to Example 6.3, the number of leaves labelled by e can be bounded by the size of $\mathrm{dom}(h)$ when all rules are memory-consuming.

Lemma 6.5. *According to Section 3, $(\mathbf{a}, h) \in [\![P]\!]^{\Phi}$ iff there is an unfolded tree for the appropriately instantiated $\widetilde{P}$ such that h is the $*$-composition of all heaps labelling the leaves of the tree.*

Lemma 6.6. *The number of nodes in the above inductive trees for $\widetilde{P}$ is bounded by $2(m+1) \cdot |\mathrm{dom}(h)|$, where m is the maximal number of predicate symbols in the body of the rules.*

Proof. It is clear that the number of leaves labelled by non-empty heaplets is bounded by $|\mathrm{dom}(h)|$. Let v be a leaf labelled by e. Then either its parent w is the root of the tree, or the edge of the form (v_0, w) incoming to w is labelled by some ρ, providing thereby a leaf v' with its incoming edge (v_0, v') labelled by the same ρ, such that v' is labelled by a *non-empty* $\widetilde{x} \mapsto \widetilde{\mathbf{t}}$ (Figure 2 shows such a ρ in MEM). Since no more than m leaves labelled by e can be associated with one and the same v' specified in such a way, *the total number of leaves* is bounded by $(m+1) \cdot |\mathrm{dom}(h)|$.

It remains to apply an induction to conclude the proof. □

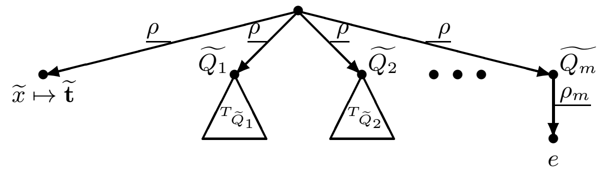


Figure 2. $\rho \;=\; \widetilde{\Pi}\colon \widetilde{x} \mapsto \widetilde{\mathbf{t}} * \widetilde{Q_1} * \widetilde{Q_2} * \cdots * \widetilde{Q_m} \Rightarrow \widetilde{R}$

6.2 NP-hardness for MEM + CV

Here we show NP-hardness for the restricted fragment MEM+ CV. The proof is by reduction from the *3-partition problem* [21].

Definition 6.7. By means of the length of circular lists, we resolve a key issue: representing *integers* as *logic formulas*.

By a *ring-formula of length ℓ, with leading variable x,* we mean a formula of the form ($x'_1, x'_2, \ldots, x'_\ell$ are fresh):

$$x \mapsto x'_1 * x'_1 \mapsto x'_2 * x'_2 \mapsto x'_3 * \cdots x'_{\ell-1} \mapsto x'_\ell * x'_\ell \mapsto x'_1$$

Given a 3-partition problem instance, i.e., a bound B and a multiset $\mathcal{S} = \{ s_1, s_2, \ldots, s_{3m} \}$, we introduce a linear-ordered list $\mathbf{X}$ of distinct variables: $\mathbf{X} = x_1, x_2, \ldots, x_i, \ldots, x_{3m}$.

Then we encode each of the numbers s_i by a ring-formula, $S_i(x_i)$, of length s_i, with the leading variable x_i.

The whole $\mathcal{S}$ is encoded as a concrete heap $h_{\mathcal{S}}$, which is a collection of $3m$ disjoint "circular" lists of the form $S_i(a_i)$. With an appropriate stack $s_{\mathcal{S}}$, $s_{\mathcal{S}}, h_{\mathcal{S}} \models \Pi(\mathbf{X})\colon \varphi_{\mathcal{S}}(\mathbf{X})$, where

$$\varphi_{\mathcal{S}}(\mathbf{X}) \;=\; S_1(x_1) * S_2(x_2) * \cdots * S_{3m}(x_{3m})$$

and $\Pi(\mathbf{X})$ says $(x \neq y)$ for all distinct variable names x and y mentioned, explicitly or implicitly, in $\varphi_{\mathcal{S}}(\mathbf{X})$.

Define a set of inductive rules $\Phi_{\mathcal{S}}$ as follows. To keep the predicate arities *fixed* and, at the same time, maintain i-th position inside the tuple $\mathbf{X}$, we define predicates $Q_i(x)$ by the rules

$$x \neq \mathsf{nil} : \mathsf{emp} \;\Rightarrow\; Q_i(x) \tag{3}$$

For $i<j<k$, we use "goal" predicates $R_{ijk}(x, y, z)$ with the rules

$$x \neq y, y \neq z, z \neq x : \mathsf{emp} \;\Rightarrow\; R_{ijk}(x, y, z) \tag{4}$$

In the case of $i<j<k$ and $s_i+s_j+s_k = B$, we add the rule:

$$S_i(x) * S_j(y) * S_k(z) * R_{ijk}(x, y, z) \;\Rightarrow\; R_{ijk}(x, y, z) \tag{5}$$

Lemma 6.8. *Let $h_{\mathcal{S}}$ and $s_{\mathcal{S}}$ be defined above. Then*

$$s_{\mathcal{S}}, h_{\mathcal{S}} \models_{\Phi_{\mathcal{S}}} \Pi(\mathbf{X})\colon \mathop{\LARGE *}_{i=1}^{3m} Q_i(x_i) * \mathop{\LARGE *}_{i<j<k} R_{ijk}(x_i, x_j, x_k)$$

iff there is a complete 3-partition on $\mathcal{S}$ - i.e., $\mathcal{S}$ can be partitioned in groups of three, say s_i, s_j, and s_k, so that $s_i+s_j+s_k = B$.

Proof. Each $R_{ijk}(a_i, a_j, a_k)$ at the top is generated either by (4), with emp, or by (5), with $S_i(a_i)$, $S_j(a_j)$, $S_k(a_k)$ being 'consumed'. The latter provides the corresponding group of s_i, s_j, s_k.

Corollary 6.9. *(a) In the case of the memory-consuming rules, the model checking problem is* NP*-complete (even if the arity of the predicates involved is at most* 3*).*

(a) For the memory-consuming rules with constructively valued variables, model checking is still NP*-complete (even if the arity of the predicates involved is at most* 3*).*

Proof. This follows from Sections 6.1 and 6.2. □

6.3 NP-hardness for MEM + DET

The challenge - to simulate *intrinsically non-deterministic* 3-SAT by *deterministic* memory-consuming rules, with, moreover, keeping the arity of predicates *fixed* - is solved using generalised versions of the linked list segments inductively defined in Example 5.3.

Namely, within a list fragment leading from x to y, by means of $RLS(x, u, y)$, defined below, we will keep the information about *the exact predecessor u of the final y* in the list.

A linked list $RLS(x, u, y)$ is formed by attaching a new tail:

$$x = y :\; \mathsf{emp} \Rightarrow RLS(x, x, y)$$
$$\exists u.\, x \neq z :\; RLS(x, u, y) * y \mapsto (y, z) \Rightarrow RLS(x, y, z)$$

Notice that the rules for $RLS(x, u, y)$ are both memory-consuming and deterministic. The fact that u is not constructively valued is a key ingredient in our reduction, which allows us to cope with the *non-deterministic* problem 3-SAT.

Definition 6.10. By means of the following heaplets $h_{ab}^{(0)}$ and $h_{ab}^{(1)}$ (here $a \neq b$) we represent the truth values, "false" and "true", resp.:

$$h_{ab}^{(\delta)} = \begin{cases} a \!\rightarrow\! \boxed{a}\boxed{b} \text{ i.e. } a \mapsto (a, b), & \text{for } \delta = 0, \\ a \!\rightarrow\! \boxed{a}\boxed{b} \!\rightarrow\! \boxed{b}\boxed{b} \text{ i.e } a \mapsto (a, b) * b \mapsto (b, b), & \text{for } \delta = 1. \end{cases}$$

Lemma 6.11. *Assuming $a \neq b$, let $h_{ab}^{(\delta)} \models_{\mathrm{RLS}} RLS(a, c, b)$ for some c. Then: $\big((\delta = 0) \wedge (a = c \neq b)\big) \vee \big((\delta = 1) \wedge (a \neq c = b)\big)$.*

Proof. $h_{ab}^{(0)} \models_{\mathrm{RLS}} RLS(a, a, b)$, and $h_{ab}^{(1)} \models_{\mathrm{RLS}} RLS(a, b, b)$. □

Let $\varphi \equiv (\widehat{C}_1 \wedge \cdots \wedge \widehat{C}_m)$ be a formula of m clauses over linear ordered n propositional variables p_1,p_2,..,p_n, and each $\widehat{C}_j$ is of the form (here, for any q, we denote $q^1 = q$ and $q^0 = \neg q$):

$$\widehat{C}_j(q_1, q_2, q_3) \equiv (q_1^{\varepsilon_{j,1}} \vee q_2^{\varepsilon_{j,2}} \vee q_3^{\varepsilon_{j,3}})$$

Each $\widehat{C_j}$ is encoded by a predicate $C_j(\alpha_1, \gamma_1, \alpha_2, \gamma_2, \alpha_3, \gamma_3)$ with the following rule ξ_j (for the sake of readability, we squeeze three deterministic rules into one but with disjunction):

$$((\alpha_1 \neq \gamma_1)^{\varepsilon_{j,1}} \vee (\alpha_2 \neq \gamma_2)^{\varepsilon_{j,2}} \vee (\alpha_3 \neq \gamma_3)^{\varepsilon_{j,3}}) \wedge \textstyle\bigwedge_{k \neq \ell} (\alpha_k \neq \alpha_\ell)\colon$$
$$\mathsf{emp} \Rightarrow C_j(\alpha_1, \gamma_1, \alpha_2, \gamma_2, \alpha_3, \gamma_3)$$

Example 6.12. Here we represent p_i as "$x_i \neq u_i$". So satisfiability of $\widehat{C}(p_1, p_2, p_3)$ of the form $(p_1 \vee \neg p_2 \vee p_3)$ is reformulated as

$$(x_1 \neq u_1) \vee (x_2 = u_2) \vee (x_3 \neq u_3)\colon C(x_1, u_1, x_2, u_2, x_3, u_3)$$

We take the following linear-ordered variables: $\mathbf{W} = w_0$, $\mathbf{X} = x_1, x_2, .., x_{2n}$, $\mathbf{U} = u_1, u_2, .., u_{2n}$, $\mathbf{Y} = y_1, y_2, .., y_{2n}$.

The challenge - to maintain the arity *fixed* and, at the same time, to "*keep i-th position*" inside the long $\mathbf{X}, \mathbf{U}, \mathbf{Y}$ - is solved by taking predicates $Q_i(x, u, y)$ with the rules κ_i:

$$x \neq y,\ u = x\colon \mathsf{emp} \Rightarrow Q_i(x, u, y), \quad x \neq y,\ u = y\colon \mathsf{emp} \Rightarrow Q_i(x, u, y).$$

Here we abbreviate $\mathbf{X} = x_0, x_1, .., x_n$, $\mathbf{Q} = q_0, q_1, .., q_\ell$, $\mathbf{\Xi} = \xi_\mathbf{0}, \xi_\mathbf{1}, .., \xi_\mathbf{m}$, $\Pi_i = \{\, x_j \neq q \mid j \neq i,\ q \in \mathbf{Q} \,\} \cup \{\, z \neq z' \mid z, z' \in \mathbf{Q} \cup \mathbf{\Xi} \,\}$

The *final* "empty configuration" is generated by the "backward" rule (recall that $\widehat{\xi}_0$ is the blank symbol, $\widehat{\xi}_1$ and $\widehat{\xi}_2$ are end markers)

$$\Pi_0,\ x_0 = q_0,\ x_1 = \xi_1,\ x_2 = x_3 = \cdots = x_{n-1} = \xi_0,\ x_n = \xi_2 :\ \mathsf{emp}\ \Rightarrow\ T(\mathbf{X}, \mathbf{Q}, \mathbf{\Xi})$$

An instruction *"if in state $\widehat{q}_k$ looking at $\widehat{\xi}_s$, replace it by $\widehat{\xi}_{s'}$, move to the right, and go into $\widehat{q}_{k'}$"*, is simulated by n rules (here $0 \leq i < n$):

$$\Pi_i,\ x_i = q_k,\ x_{i+1} = \xi_s,\ y_i = \xi_{s'},\ y_{i+1} = q_{k'} :\ T(x_0, x_1, .., x_{i-1}, y_i, y_{i+1}, x_{i+2}, .., x_n, \mathbf{Q}, \mathbf{\Xi})\ \Rightarrow\ T(\mathbf{X}, \mathbf{Q}, \mathbf{\Xi})$$

An instruction *"if in state $\widehat{q}_k$ looking at $\widehat{\xi}_s$, replace it by $\widehat{\xi}_{s'}$, move to the left, and go into $\widehat{q}_{k'}$"*, is simulated in a similar "backward" way.
An *alternating* instruction *"if in state $\widehat{q}_k$, run two copies of the configuration in parallel but with states $\widehat{q}_{k'}$ and $\widehat{q}_{k''}$, resp."*, is simulated by $(0 \leq i < n)$:

$$\Pi_i,\ x_i = q_k,\ y_i = q_{k'},\ z_i = q_{k''} :\ T(x_0, .., x_{i-1}, y_i, x_{i+1}, .., x_n, \mathbf{Q}, \mathbf{\Xi}) * T(x_0, .., x_{i-1}, z_i, x_{i+1}, .., x_n, \mathbf{Q}, \mathbf{\Xi}) \Rightarrow T(\mathbf{X}, \mathbf{Q}, \mathbf{\Xi})$$

Figure 3. Simulating a Turing machine M running in space n, in a backward manner - "from outputs to inputs"

The key points of our reduction are encapsulated in the rule ω_{ok}:

$$\exists\, \mathbf{X}, \mathbf{U}, \mathbf{Y}, \mathbf{W}\ \Big(\Pi(\mathbf{X}, \mathbf{Y}, \mathbf{W}) :\ w_0 \mapsto w_0\ *$$
$$\mathop{\ast}\nolimits_{i=1}^{2n} Q_i(x_i, u_i, y_i) * \mathop{\ast}\nolimits_{i=1}^{2n} RLS(x_i, u_i, y_i)\ *$$
$$\mathop{\ast}\nolimits_{j=1}^{m} C_j(x_{i_{j,1}}, u_{i_{j,1}}, x_{i_{j,2}}, u_{i_{j,2}}, x_{i_{j,3}}, u_{i_{j,3}})\Big)\ \Rightarrow\ ok$$

where $\Pi(\mathbf{X}, \mathbf{Y}, \mathbf{W})$ says $(x \neq y)$ for all distinct variable names x and y mentioned either in $\mathbf{X}$ or in $\mathbf{Y}$ or in $\mathbf{W}$.

Definition 6.13. Define a heap $H_{\mathtt{RLS}}$ as a collection of n disjoint heaplets of the form $h^{(0)}_{ab}$, and n disjoint heaplets of the form $h^{(1)}_{ab}$ (we assume $a \neq b$), and a heap H_φ as a loop of the form $d_0 \mapsto d_0$.

Lemma 6.14. *With the empty input tuple of values,*

$$((), H_{\mathtt{RLS}} \circ H_\varphi) \in [\![ok]\!]^{\,\omega_{ok}\ \cup\ \mathtt{RLS}\ \cup\ \bigcup_{j=1}^{m} \xi_j\ \cup\ \bigcup_i \kappa_{i=1}^{2n}} \tag{6}$$

if and only if $(\widehat{C}_1 \wedge \cdots \wedge \widehat{C}_m)$ is satisfiable.

Proof. (Sketch) The $(\Rightarrow)$ direction is the hardest. Suppose that (6) is valid. Then *ok* is generated by ω_{ok} with an unfolded inductive tree for *ok*. We can show that, for some values $a_1, a_2, .., a_{2n}$, $c_1, c_2, .., c_{2n}$, $b_1, b_2, .., b_{2n}$, the heap $H_{\mathtt{RLS}}$ can be partitioned into $2n$ disjoint heaplets of the form $h^{(\delta_i)}_{a_i b_i}$, so that we get the following: $h^{(\delta_i)}_{a_i b_i} \models_{\mathtt{RLS}} RLS(a_i, c_i, b_i)$. Now, using the rules ξ_j and κ_i and Lemma 6.11, we can prove the desired satisfiability (see also Example 6.12): $\varphi(\delta_1, \delta_2, \ldots, \delta_{n-1}, \delta_n) = 1$.

The $(\Leftarrow)$ direction follows essentially by reading the above line of reasoning "bottom-up". □

Corollary 6.15. *For deterministic and memory-consuming rules, model checking is still* NP*-complete (even if the arity of the predicates involved is at most* 6*).*

6.4 PSPACE- and EXPTIME-hardness for non-MEM rules.

Unexpectedly, with *non-memory-consuming rules*, such as

$$\exists \mathbf{z}.\, \Pi :\ Q_1\mathbf{u_1}\ *\ Q_2\mathbf{u_2}\ * \cdots *\ Q_m\mathbf{u_m}\ \Rightarrow\ P\mathbf{x}.$$

model checking becomes more complex. Namely:

Theorem 6.16. (a) *For inductive rule sets in* CV*, model checking is* EXPTIME*-complete.*

(b) *For inductive rule sets in* CV + DET*, model checking is* PSPACE*-hard.*

(c) *For inductive rule sets in* DET*, model checking is* EXPTIME*-complete.*

Proof. (Sketch) We prove all three lower bounds by simulating Turing machines in a backward manner - "from outputs to inputs".

Let M be a Turing machine that accepts in space n, with states $\widehat{q}_0$, $\widehat{q}_1$,.., $\widehat{q}_\ell$, and tape symbols $\widehat{\xi}_0$, $\widehat{\xi}_1$,.., $\widehat{\xi}_m$. Here $\widehat{q}_1$ is the initial state, $\widehat{q}_0$ is an accept state, $\widehat{\xi}_0$ is the blank symbol, and M acts in the space n between two *unerased* markers, say $\widehat{\xi}_1$ and $\widehat{\xi}_2$.
Let M always jump, and no M's instruction starts with $\widehat{q}_0$.
By $(\eta_1, \eta_2, .., \eta_{i-1},\ q_k,\ \eta_i, .., \eta_n)$ we formalize that *"in state $\widehat{q}_k$, M scans i-th square, when $\widehat{\eta}_1, \widehat{\eta}_2, .., \widehat{\eta}_{i-1}, \widehat{\eta}_i, .., \widehat{\eta_n}$ is printed on its tape"*. We encode M by means of the rules Φ_M given in Figure 3, where the "tape" predicate T depicts M's configurations:

$$T(\underbrace{x_0, x_1, .., x_n}_{\text{configuration}}, \underbrace{q_0, q_1, .., q_\ell}_{\text{states}}, \underbrace{\xi_0, \xi_1, .., \xi_m}_{\text{tape symbols}})$$

Lemma 6.17. *Let M be a deterministic TM or an alternating TM [15]. Then $s, e \models_{\Phi_M} T(q_1, \xi_1, \xi_0, \xi_0, \xi_0 .., \xi_0, \xi_2, \mathbf{Q}, \mathbf{\Xi})$ if and only if M can go from the initial "empty configuration" to the final "empty configuration".* □

All variables in Φ_M occur in $\mathbf{X} \cup \mathbf{Q} \cup \mathbf{\Xi}$, hence, they are *constructively valued*, which provides item (a) in Theorem 6.16.

Whenever M is deterministic, Φ_M is deterministic, which provides item (b) in Theorem 6.16.

To answer the challenging (c), M's *non-deterministic* instruction *"if in state $\widehat{q}_k$ looking at $\widehat{\xi}_s$, go either into $\widehat{q}_{k_1}$ or into $\widehat{q}_{k_2}$"* is simulated by the *deterministic* rules (here $0 \leq i < n,\ \ell = 1, 2$):

$$\exists y.\ \Pi_i,\ x_i = q_k,\ x_{i+1} = \xi_s,\ y \notin \mathbf{\Xi},\ y_{i+1} = \xi_{d(s,k)} :$$
$$T(x_0, .., x_{i-1}, y, y_{i+1}, x_{i+2}, .., x_n, \mathbf{Q}, \mathbf{\Xi})\ \Rightarrow\ T(\mathbf{X}, \mathbf{Q}, \mathbf{\Xi}),$$
$$\Pi_i,\ x_i = q_{d'(s,k,k_\ell)},\ x_{i+1} = \xi_{d(s.k)},\ y_i = q_{k_\ell},\ y_{i+1} = \xi_s :$$
$$T(x_0, .., x_{i-1}, y_i, y_{i+1}, x_{i+2}, .., x_n, \mathbf{Q}, \mathbf{\Xi})\ \Rightarrow\ T(\mathbf{X}, \mathbf{Q}, \mathbf{\Xi})$$

where, however, y is *not* constructively valued.

The idea behind the encoding is as follows. First, M goes non-deterministically into *any state* y, with encrypting the situation by a special 'double' $\widehat{\xi}_{d(s.k)}$. To continue a computation, the lucky guess should be one of our specially introduced states $\widehat{q}_{d'(s,k,k_1)}$ and $\widehat{q}_{d'(s,k,k_2)}$. As a result, M finishes either in $\widehat{q}_{k_1}$ or in $\widehat{q}_{k_2}$. □

Remark 6.18. The above EXPTIME-hardness necessarily employs predicates of unbounded arity (cf. Remark 5.1).

6.5 Polynomial time for MEM+CV+DET

We next show that the model checking problem is in PTIME whenever Φ is in MEM+CV+DET. Essentially, predicates in CV+DET make a top-down procedure fully deterministic; and, the size of any possible proof is linear in the size of the heap, if Φ is in MEM.

Where h is a heap and v is a value such that $v \in \mathrm{dom}(h)$, we write $h \dot{-} v$ to denote the heap h' that has domain $\mathrm{dom}(h) \setminus v$ and agrees with h on its domain. This operation is lifted in the obvious way to sets of values, i.e., $h \dot{-} V$ where $V \subseteq \mathrm{dom}(h)$.

Definition 6.19. The construct $s, h \Vdash_\Phi A \ \ \rightsquigarrow\ \ h'$ is called a reduction, where s is a stack, h, h' are heaps and A is a symbolic heap with inductive predicate occurrences defined in Φ. We say that the above reduction is *valid* if $h' \subseteq h$ and $s, h \dot{-} \mathrm{dom}(h') \models_\Phi A$.

Figure 4 presents a proof system for reductions. A proof is, as usual, a tree whose leaves are labelled by axioms and internal nodes

$$\frac{s(x) = s(y) \qquad s, h \Vdash_\Phi \Pi : F \;\rightsquigarrow\; h'}{s, h \Vdash_\Phi x = y, \Pi : F \;\rightsquigarrow\; h'}\,(=) \qquad \frac{s(x) \neq s(y) \qquad s, h \Vdash_\Phi \Pi : F \;\rightsquigarrow\; h'}{s, h \Vdash_\Phi x \neq y, \Pi : F \;\rightsquigarrow\; h'}\,(\neq) \qquad \frac{s(x) \in \mathrm{dom}(h) \qquad h(s(x)) = s(\mathbf{t})}{s, h \Vdash_\Phi x \mapsto \mathbf{t} \;\rightsquigarrow\; h \mathbin{\dot{-}} s(x)}\,(\mapsto)$$

$$\frac{s, h \Vdash_\Phi \sigma \;\rightsquigarrow\; h' \qquad s, h' \Vdash_\Phi F \;\rightsquigarrow\; h''}{s, h \Vdash_\Phi \sigma * F \;\rightsquigarrow\; h''}\,(*) \qquad \frac{x \in \mathbf{v} \qquad y \notin \mathbf{v} \qquad s[x \mapsto s(y)], h \Vdash_\Phi \exists\,(\mathbf{v} \setminus \{x\})\,.\,(x = y, \Pi : F) \;\rightsquigarrow\; h'}{s, h \Vdash_\Phi \exists \mathbf{v}.\,(x = y, \Pi : F) \;\rightsquigarrow\; h'}\,(\exists_=)$$

$$\frac{x \in \mathbf{v} \qquad y \notin \mathbf{v} \qquad s(y) \in \mathrm{dom}(h) \qquad \exists i.\, t_i \equiv x \qquad s[x \mapsto h(s(y))_i], h \Vdash_\Phi \exists\,(\mathbf{v} \setminus \{x\})\,.\,(\Pi : F * y \mapsto \mathbf{t}) \;\rightsquigarrow\; h'}{s, h \Vdash_\Phi \exists \mathbf{v}.\,(\Pi : F * y \mapsto \mathbf{t}) \;\rightsquigarrow\; h'}\,(\exists_\mapsto)$$

$$\frac{}{s, h \Vdash_\Phi \mathsf{emp} \;\rightsquigarrow\; h}\,(\mathsf{emp}) \qquad \frac{(\exists \mathbf{v}.\,(\Pi : F) \Rightarrow P_i\mathbf{x}) \in \Phi \qquad s, e \models \exists \mathbf{v}.(\Pi : \mathsf{emp}) \qquad s[\mathbf{x} \mapsto s(\mathbf{t})], h \Vdash_\Phi \exists \mathbf{v}.\,(\Pi : F) \;\rightsquigarrow\; h'}{s, h \Vdash_\Phi P_i\mathbf{t} \;\rightsquigarrow\; h'}\,(P_i)$$

Figure 4. Proof rules for reductions. The formula σ in the rule $(*)$ is atomic.

are labelled by inference rules accordingly. We say that a reduction R is provable if there exists a proof whose root is labelled by R.

Lemma 6.20. *(Soundness) For any set of rules Φ, formula A, stack s and heap h, if $s, h \Vdash_\Phi A \;\rightsquigarrow\; h'$ is provable then it is valid.*

Proof. Follows by induction over the structure of the proof. □

Let $\bar{\phi}(\mathbf{Y}) =_{\mathrm{def}} (\bigcup_j \varphi_{1,j}(\mathbf{Y}), \ldots, \bigcup_j \varphi_{n,j}(\mathbf{Y}))$ (cf. Def. 3.3). Set $\bar{\phi}^0 = (\emptyset, \ldots, \emptyset)$ and $\bar{\phi}^{\alpha+1} = \bar{\phi}(\bar{\phi}^\alpha)$ for any ordinal α. Clearly, $(\mathbf{a}, h) \in [\![P_i]\!]^\Phi$ iff there is an ordinal α such that $(\mathbf{a}, h) \in \mathsf{proj}_i(\bar{\phi}^\alpha)$. Next, we write $s, h \models^\alpha_\Phi \Pi : F$ if it is the case that $s, h \models_\Phi \Pi : F$ and for every $P_i\mathbf{t}$ in F, $(s(\mathbf{t}), h') \in \mathsf{proj}_i(\bar{\phi}^\alpha)$ for the appropriate subheap $h' \subseteq h$. We extend this to quantified formulas using the same ordinal, and to valid reductions in the obvious manner. Finally, we say that $s, h \Vdash_\Phi A \;\rightsquigarrow\; h'$ is α-supported if α is the *least ordinal* such that $s, h \mathbin{\dot{-}} \mathrm{dom}(h') \models^\alpha_\Phi A$.

Let $\mathcal{R}$ be the set of valid, constructively valued reductions. We define an ordering $\prec$ over $\mathcal{R}$. Let $R_i \equiv s_i, h_i \Vdash_\Phi A_i \;\rightsquigarrow\; h'_i$ be an α_i-supported reduction in $\mathcal{R}$, for $i \in \{1, 2\}$. Then $R_1 \prec R_2$ iff either: (a) $\alpha_1 < \alpha_2$; or (b) $\alpha_1 = \alpha_2$ and

1. $A_1 \equiv \Pi_1 : F$ and $A_2 \equiv \Pi_2 : F$ and $\Pi_1 \subset \Pi_2$, or,
2. $A_1 \equiv F$ and $A_2 \equiv \sigma * F$ for some atomic σ, or,
3. $A_1 \equiv \exists \mathbf{x}.\, B$ and $A_2 \equiv \exists \mathbf{y}.\, B$ and $\mathbf{x} \subset \mathbf{y}$.

The ordering $\prec$ is easily seen to be well-founded.

Lemma 6.21. *(Completeness) For any set of rules Φ and formula A that are in* CV*, and any stack s and heap h, if $s, h \Vdash_\Phi A \;\rightsquigarrow\; h'$ is valid then it is provable.*

Proof. By well-founded induction over $(\mathcal{R}, \prec)$. Consult the appendix for a complete proof. □

Next we show that proof search is complete for CV predicates, *without* requiring DET or MEM rules.

Lemma 6.22. *Let R be a reduction and Φ a set of rules, both in* CV*. A backwards-chaining, back-tracking, depth-first proof search starting with R, always terminates and decides the validity of R.*

Proof. First, observe that the structure of a given reduction dictates that there is at most one applicable rule from Fig. 4. Rules $(\exists_=)$ and $(\exists_\mapsto)$ are an exception, but the order of their applications, as well as the choice of quantified variable to eliminate next, is immaterial to the provability of a CV reduction, thus a fixed order can be used.

The rule $(*)$ resembles a cut in that the intermediate heap h' does not appear in the conclusion. However, as $h' \subseteq h$ must hold it suffices to back-track over all subheaps of h, meaning that the non-determinism associated with $(*)$ is bounded.

The non-determinism in (P_i) is also bounded as Φ is finite.

Thus, this procedure is terminating as every application of a rule during proof search produces subgoal reductions that are $\prec$-less than the conclusion, and the $\prec$-ordering is well-founded.

Lemmas 6.20 and 6.21 yield soundness and completeness. □

We recall the notion of precision: a formula A is precise iff for every stack s and heap h, there is at most one $h' \subseteq h$ such that $s, h' \models_\Phi A$. Precision entails that if $s, h \Vdash_\Phi A \;\rightsquigarrow\; h'$ is valid then there is no other $h'' \neq h'$ such that $s, h \Vdash_\Phi A \;\rightsquigarrow\; h''$ is valid. Thus precision allows the deterministic application of $(*)$.

Lemma 6.23. *Let Φ be a set of rules in* CV+DET*. Then any formula in* CV *using predicates defined in Φ is precise.*

Proof. Observe the following points: if Σ and Σ' are precise then so is $\Sigma * \Sigma'$; if Σ is precise then so is $\Pi : \Sigma$, for any Π; if $\Pi : \Sigma$ is precise then so is $A \equiv \exists \mathbf{v}.\,(\Pi : \Sigma)$, provided all variables in $\mathbf{v}$ are constructively valued in A. Finally, note that the problem reduces to guaranteeing that every formula of the form $P_i\mathbf{t}$ is precise.

Thus we need to show that for every tuple of values $\mathbf{a}$ and heap h there is at most one $h' \subseteq h$ such that $(\mathbf{a}, h') \in [\![P_i]\!]^\Phi$. This follows by fixpoint induction. Suppose there are values $\mathbf{a}$ and heaps $h_1, h_2 \subseteq h$ such that $h_1 \neq h_2$ and $(\mathbf{a}, h_1), (\mathbf{a}, h_1) \in \bigcup_j \varphi_{i,j}(\mathbf{Y})$. As the rules are precise there must be $k \neq l$ such that $(\mathbf{a}, h_1) \in \varphi_{i,k}(\mathbf{Y})$ and $(\mathbf{a}, h_2) \in \varphi_{i,l}(\mathbf{Y})$. This, however, directly contradicts determinism. □

Finally, we establish that proof search becomes fully deterministic when all predicates all constructively valued and deterministic; and, its runtime is naturally bounded by a polynomial in the input size, if in addition, all rules are memory consuming.

Theorem 6.24. *Let A be a formula in* CV *and Φ a set of rules in* MEM+CV+DET*. Then, for any stack s and heap h, checking whether $s, h \models_\Phi A$ can be performed in polynomial time.*

Proof. First, note that $s, h \models_\Phi A$ iff $s, h \Vdash_\Phi A \;\rightsquigarrow\; e$ is provable.

As Φ is in DET, (P_i) can be used with at most one rule and this rule can be found in polytime by evaluating the side condition of (P_i) for all rules. This means no back-tracking is required over Φ.

The only remaining sourse of non-determinism is in $(*)$ where many choices for h' may have to be checked. However, as all formulas involved are precise (Lemma 6.23) if there is such a heap h' it is unique. Observe that only axioms impose constraints on the RHS heap. Thus, we avoid back-tracking by using meta-variables for the heap h'. If the left subgoal of $(*)$ is proven, then h' is instantiated by axioms and the search continues in the right subgoal of $(*)$. This makes the proof search fully deterministic.

Now, for MEM rules each application of (P_i) leads to at least one subgoal that requires $(\mapsto)$, and there cannot be more instances of $(\mapsto)$ in a proof than the size of the heap h in the root reduction.

			(MEM+CV+DET)			
# Heap	stack.c	iter.c		queue.c		schorr
Cells	push	add	dequeue	dispose		-waite.c
	[pre]	[pre]	[pre]	[pre]	[loop inv]	[pre]
0-5	4	5	6	8	5	7
6-10	6	6	6	5	8	10
11-15	5	6	8	8	9	14
16-20	8	7	10	10	14	19
21-25	9	9	16	13	22	31
26-30	10	11	16	19	38	41
31-40	12	16	34	32	93	81
41-50	22	20	55	58	222	148
51-60	28	30	92	95	477	224
61-70	40	45	153	153	859	335
71-80	56	62	224	220	1463	491
81-90	74	83	331	327	2480	747
91-100	110	114	467	470	3863	967

# Heap Cells	(MEM+DET) schorr-waite.c [post]
0-5	7
6-10	14
11-15	28
16	32
17	34
18	42
19	51
20	72
21	107
22	160
23	249
24	380
25	590
26	1021
27	1887
28	3597
29	7218
30	10579

	(EXPTIME)	(MEM+CV)	(MEM)
# Heap	composite.c		set.c
Cells	add/disp	add	set(x)
	tree(x)	[pre]	
3	12	11	5
4	14	8	7
5	21	18	6
6	21	17	13
7	18	20	35
8	35	38	120
9	45	46	378
10	56	61	1237
11	71	94	3778
12	107	143	10438
13	155	210	-
14	231	279	-
15	359	388	-
16	474	524	-
17	690	985	-
18	1056	1384	-
19	1467	1533	-

(a) Tests using single model per heap size

	(MEM+CV+DET)		(MEM)				(CV)		(EXPTIME)	
# Heap	queue.c		iter.c				schorr-waite.c			
Cells	dequeue[loop]		iter(it, list)		next[pre]		[loop inv](1)		[loop inv](2)	
	Min. /	Max.	Min. /	Max.	Min. /	Max.	Min. /	Max.	Min. /	Max.
7	2 /	8	9 /	12	8 /	13	6 /	22	6 /	21
8	3 /	7	12 /	15	12 /	17	9 /	30	10 /	28
9	4 /	10	11 /	18	16 /	20	9 /	43	12 /	45
10	5 /	10	17 /	23	20 /	27	13 /	62	13 /	59
11	6 /	13	21 /	27	25 /	31	19 /	89	15 /	103
12	7 /	15	29 /	34	32 /	38	21 /	122	18 /	133
13	7 /	25	35 /	40	39 /	47	18 /	183	18 /	233
14	9 /	20	43 /	48	52 /	62	27 /	254	22 /	371
15	11 /	25	52 /	59	62 /	70	30 /	491	26 /	813
16	14 /	25	64 /	74	79 /	88	34 /	701	31 /	1151
17	19 /	32	87 /	92	101 /	110	37 /	1275	34 /	1923
18	21 /	32	104 /	112	124 /	133	40 /	2077	39 /	2850
19	24 /	52	128 /	140	153 /	163	42 /	2983	41 /	3980
20	26 /	66	153 /	164	187 /	199	49 /	5403	46 /	6872
21	35 /	76	189 /	202	237 /	250	-		-	
22	14 /	89	224 /	243	274 /	286	-		-	
23	53 /	118	288 /	298	348 /	358	-		-	

(b) Tests using multiple models per heap size

	(# Heap Cells)																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Min.	1	2	2	1	1	1	2	2	3	5	15	30	87	208	536	1416	3476	8121	18833	41442
Max.	3	7	7	5	8	6	8	8	12	15	24	41	113	284	669	1641	4065	9120	20729	44114

(c) Spatial True Encoding

Figure 5. Test results of fixed point model checking algorithm (all times in milliseconds)

Thus the size of the proof is $O(\|h\|)$, and as the search is deterministic, runtime is linear as well. □

7. Implementation and evaluation

We implemented the general model checking algorithm described in Section 3 as well as the CV+DET algorithm described in Section 6.5, in about 1400 lines of OCaml code. Our implementation is part of the CYCLIST theorem proving framework [13] which provides support for inductive definitions, and in particular for our logic $\mathsf{SL}^{\mathsf{SH}}_{\mathsf{ID}}$. Both model checking algorithms are parameterised over the datatypes for heap locations and ground values (e.g. integers, booleans, strings, etc.) and thus may be instantiated to handle models where heap locations have arbitrary representations (e.g. hex strings) and heap cells contain arbitrary data. We employed a number of techniques to improve the efficiency of the implementation, including pre-computing the models for points-to subformulas, using hashsets to store submodels, and using bit vectors to represent heaps. We also extended both algorithms so that we can model-check assertions of the form $A * \top$, since the runtime heap will, in general, contain more cells than those corresponding to the local assertion being checked. The code and test suite for our tool are available online [1].

We tested the performance of our implementation across a range of 'typical' predicate definitions gathered from the verification community, and a number of hand-crafted definitions designed to elicit the worst-case, exponential performance. We extracted models from a number of example programs at runtime using an extension of GDB which supports scripting using Python [2]. All tests were carried out on a 2.93GHz Intel Core i7-870 processor with 8GB RAM.

'Typical' performance Testing our implementation against typical, real-world data requires sourcing programs annotated with separation logic assertions. We identified 6 programs from the suite of examples in the Verifast distribution [24] containing non-trivial inductive predicates which translate into our assertion language:

(i) **stack.c**: a stack data-structure implemented using linked lists.
(ii) **queue.c**: a lock-free concurrent queue based on list segments.
(iii) **set.c**: a concurrent set data-structure based on linked lists.
(iv) **schorr-waite.c**: an implementation of the Schorr-Waite graph marking algorithm over binary trees.
(v) **iter.c**: a list data-structure with an *iterator* pointing into the list.
(vi) **composite.c**: an example of the *composite* design pattern, where each node of a tree must maintain local data consistent with a global property.

The assertions we collected cover most combinations of the syntactic restrictions defined in Section 5. The first four programs use standard list segment and tree predicates, all of which are in the PTIME fragment apart from iter.c which uses a non-DET predicate (describing a possibly cyclic list). The predicate describing the set data-structure in set.c is in MEM+DET, but is not in CV. The models we tested for the Schorr-Waite algorithm were taken from the entry and exit points and the start of each loop iteration. The precondition is MEM+CV+DET, although the postcondition is not CV. The loop condition given in the Verifast code is CV but not MEM or DET, and we also checked against a natural 'simplified' version, obtained by a predicate fold, which falls outside all the restrictions. For the iterator example one of the three formulas we model checked against is in the PTIME fragment, while the others are only MEM. The composite pattern example uses 'non-standard' predicates, making essential use of non-DET and non-CV rules: we checked against one formula MEM+CV, and the other in the general EXPTIME fragment.

From each program we extracted models with heaps containing 0–100 cells, with each cell containing 1–4 data values. We extracted over 2150 models in total, covering 15 different assertions across various program points. The results of our model checking tests of the general algorithm can be seen in Figures 5(a) and 5(b). Where we were able to collect a number of non-isomorphic models per heap size, we show both the minimum and maximum times observed for model checking. In Figure 5(a), where we give a result for a range of heap sizes, this represents the maximum time observed over that range. All times are given in milliseconds, and averaged over 3 test runs.

The results, e.g. for the set.c example and the schorr-waite.c postcondition, clearly show the exponential nature of the general algorithm. The maximum times observed for the schorr-waite.c loop invariants display a similar profile, although the large variation between minimum and maximum times shows that the particular shape of the model being checked also has a large influence. The results for those formulas in the PTIME fragment display less pronounced growth, increasing by a factor of roughly between 1.5–2 for each 10 heap cells added. We suspect the large variation in times is due to the specific shapes of the particular models being checked. The stack.c and iter.c models are simple list segments checked against formulas containing only single predicate instances. The queue.c and schorr-waite.c tests, in contrast, consist of formulas with multiple predicate instances and so consequently many more submodels must be calculated. We suggest that the general algorithm might not be practical for runtime model checking, except possibly as a fallback for faster algorithms, but can be useful for offline testing.

We also tested the algorithm for the CV+DET fragment in Section 6.5 on the formulas falling within it (formulas appearing in the left-hand side tables of Fig. 5(a) and 5(b)). The runtimes were excellent, never exceeding 12 milliseconds in the worst case; we do not report these times as they are very low and close to our experimental error margin. This suggests that this algorithm should be suitable for dynamic checking of assertions.

Worst-case performance We tested the general algorithm in two situations designed to elicit worst-case behaviour. Firstly, we took all of the models up to size 20 extracted from the Verifast example programs, and checked them against a "universal" predicate (i.e., $\top$) satisfied by every model. The definition of this predicate is in MEM, but not CV or DET. Thus, these instances are in an NP-complete fragment. The results of these tests, in Figure 5(c), clearly demonstrate the worst-case performance.

Our second test is on a family of predicates encoding an n-bit binary counter (as also used to elicit worst-case performance for satisfiability in [10]). The predicate definitions in this example satisfy *none* of MEM, CV, and DET (in fact, they contain no spatial formulas at all), and thus fall into the EXPTIME fragment. The 2-bit counter checks in 15ms, the 3-bit counter in about 2.95s, and the 4-bit counter in just over 6 minutes; the 5-bit counter had not returned a value after 20 hours.

8. Conclusions

Our main contribution in this paper is a general fixed point algorithm that decides model checking in symbolic-heap separation logic with user-defined inductive predicates. Furthermore, we show that while model checking for this logic is EXPTIME-complete in the general case, it can be made NP-complete and even polynomial by restricting the admissible definitions of inductive predicates in various natural (syntactic) ways. Finally, we implement two algorithms, the general one described in Section 3 and the algorithm for the CV + DET fragment, described in Section 6.5.

The performance of the CV + DET algorithm is good enough for runtime verification, when additionally the rules employed are memory-consuming. This, however, significantly restricts the kinds of properties that can be checked during program execution. As evidenced by the complex properties found in the Verifast examples, the general model checking algorithm can usefully extend the expressiveness available to the programmer; its performance may not be appropriate for verification during execution, but it can still be used productively in offline testing such as in unit tests.

The ability to model-check formulas also opens up the possibility of *disproving* entailments in our logic via the direct generation and testing of possible countermodels, in contrast e.g. to the approach based on overapproximation in [12]. We are uncertain as to the scalability of such an approach, but nevertheless consider it an interesting avenue for potential future work. Another possibility includes employing SAT solvers for deciding model-checking of rule sets that fall within the MEM + DET or MEM + CV fragments.

References

[1] CYCLIST: software distribution. `https://github.com/ngorogiannis/cyclist/`.

[2] Project Archer GDB development branch. `https://sourceware.org/gdb/wiki/ProjectArcher`.

[3] P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *Proc. POPL-42*. ACM, 2015.

[4] T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *Proc. FoSSaCS-17*. Springer, 2014.

[5] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Proc. FSTTCS-24*. Springer, 2004.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proc. APLAS-3*. Springer, 2005.

[7] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: memory safety for systems-level code. In *Proc. CAV-23*. Springer, 2011.

[8] M. Botincan, D. Distefano, M. Dodds, R. Grigore, D. Naudziuniene, and M. J. Parkinson. coreStar: The core of jStar. In *Proc. 1st BOOGIE*, 2011.

[9] J. Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *Proc. SAS-14*. Springer, 2007.

[10] J. Brotherston, C. Fuhs, N. Gorogiannis, and J. Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proc. CSL-LICS*. ACM, 2014.

[11] J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *Proc. SAS-21*. Springer, 2014.

[12] J. Brotherston and N. Gorogiannis. Disproving inductive entailments in separation logic via base pair approximation. In *Proc. TABLEAUX-24*, 2015. To appear.

[13] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proc. APLAS-10*, LNCS. Springer, 2012.

[14] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *Proc. NFM-3*. Springer, 2011.

[15] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1), 1981.

[16] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comp. Prog.*, 77(9), 2012.

[17] D.-H. Chu, J. Jaffar, and M.-T. Trinh. Automatic induction proofs of data-structures in imperative programs. In *Proc. PLDI-36*. ACM, 2015.

[18] E. M. Clarke. The birth of model checking. In *25 Years of Model Checking*. Springer, 2008.

[19] K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *Proc. CAV-23*. Springer, 2011.

[20] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130, 2004.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[22] R. Iosif, A. Rogalewicz, and J. Simacek. The tree width of separation logic with recursive definitions. In *Proc. CADE-24*. Springer, 2013.

[23] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. POPL-28*. ACM, 2001.

[24] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proc. NFM-3*. Springer, 2011.

[25] Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape analysis via second-order bi-abduction. In *Proc. CAV-26*. Springer, 2014.

[26] H. H. Nguyen, V. Kuncak, and W.-N. Chin. Runtime checking for separation logic. In *Proc. VMCAI-9*. Springer, 2008.

[27] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. POPL-31*. ACM, 2004.

[28] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *Proc. PLDI-35*. ACM, 2014.

[29] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS-17*. IEEE, 2002.

A. Proof of Lemma 6.21

Proof. We proceed by well-founded induction over $(\mathcal{R}, \prec)$, i.e., we show that if all $R' \in \mathcal{R}$ such that $R' \prec R$ are provable, then so is $R \in \mathcal{R}$.

Let $R \in \mathcal{R}$ be the reduction $s, h \Vdash_\Phi \exists \mathbf{v}.\, A \rightsquigarrow h'$. As R is constructively valued, there must be some variable $x \in \mathbf{v}$ such that (i) there is a free variable y such that $x = y$ is a subformula of A, or, (ii) x appears in the right-hand side of a subformula $y \mapsto \mathbf{t}$ of A. Let $\mathbf{z} =_{\text{def}} \mathbf{v} \setminus \{x\}$. As R is valid we have $h' \subseteq h$ and $s, h \mathbin{\dot{-}} \mathrm{dom}(h') \models^\alpha_\Phi \exists \mathbf{v}.\, A$, thus there is a stack s' such that $s'(FV(\exists \mathbf{z}.\, A)) = s(FV(\exists \mathbf{z}.\, A))$ and $s', h \mathbin{\dot{-}} \mathrm{dom}(h') \models^\alpha_\Phi \exists \mathbf{z}.\, A$.

If clause (i) is true then clearly $s'(x) = s'(y)$ by the semantics of $=$, and $s'(y) = s(y)$ as y is free. Thus without loss of generality we can set $s' = s[x \mapsto s(y)]$. Therefore the reduction $R' \equiv s[x \mapsto s(y)], h \Vdash_\Phi \exists \mathbf{z}.\, A \rightsquigarrow h'$ is valid and constructively valued. It is easy to see that R' must be α-supported (otherwise we can derive a contradiction with the assumption that α is the least such ordinal), thus $R' \prec R$ by clause (2c) of the definition of $\prec$. By the inductive hypothesis, R' is provable and the rule $(\exists_=)$ can be applied, thus proving R. Clause (ii) is similar and uses $(\exists_\mapsto)$.

Next, let $R \in \mathcal{R}$ be of the form $s, h \Vdash_\Phi x = y, \Pi : F \rightsquigarrow h'$, thus, $h' \subseteq h$ and $s, h \mathbin{\dot{-}} \mathrm{dom}(h') \models^\alpha_\Phi x = y, \Pi : F$. It is easy to see that the reduction $R' \equiv s, h \Vdash_\Phi \Pi : F \rightsquigarrow h'$ is valid, constructively valued and α-supported. By clause (2a) of the definition of $\prec$ it follows that $R' \prec R$. Thus by the inductive hypothesis, R' is provable and the rule $(=)$ applies, therefore R is also provable. Disequalities are treated similarly via the rule $(\neq)$.

Now, let $R \in \mathcal{R}$ be of the form $s, h \Vdash_\Phi \sigma * F \rightsquigarrow h'$. Thus, $h' \subseteq h$ and $s, h \mathbin{\dot{-}} \mathrm{dom}(h') \models^\alpha_\Phi \sigma * F$. Therefore there are two disjoint heaps h_σ, h_F such that $h \mathbin{\dot{-}} \mathrm{dom}(h') = h_\sigma \circ h_F$, and $s, h_\sigma \models^{\alpha_\sigma}_\Phi \sigma$ and $s, h_F \models^{\alpha_F}_\Phi F$, where $\alpha = \max\{\alpha_\sigma, \alpha_F\}$. Thus $h = h_\sigma \circ h_F \circ h'$, and $s, h \mathbin{\dot{-}} \mathrm{dom}(h_F \circ h') \models^{\alpha_\sigma}_\Phi \sigma$ and $s, h_F \circ h' \mathbin{\dot{-}} \mathrm{dom}(h') \models^{\alpha_F}_\Phi F$. Therefore the reductions $R_\sigma \equiv s, h \Vdash_\Phi \sigma \rightsquigarrow h_F \circ h'$ and $R_F \equiv s, h_F \circ h' \Vdash_\Phi F \rightsquigarrow h'$ are both valid and constructively valued. In addition, $R_\sigma, R_F \prec R$: either $\alpha_\sigma < \alpha$ (resp., $\alpha_F < \alpha$) where clause (1) of the definition of $\prec$ applies, or $\alpha_\sigma = \alpha$ ($\alpha_F = \alpha$) and clause (2b) applies. By the inductive hypothesis, R_σ and R_F are provable, and so is R via $(*)$.

Finally, suppose $R \in \mathcal{R}$ has the form $s, h \Vdash_\Phi P_i\mathbf{t} \rightsquigarrow h'$, meaning that $h' \subseteq h$ and $s, h \mathbin{\dot{-}} \mathrm{dom}(h') \models^\alpha_\Phi P_i\mathbf{t}$. By Def. 3.3, this means that $(s(\mathbf{t}), h \mathbin{\dot{-}} \mathrm{dom}(h')) \in [\![P_i]\!]^\Phi$ which in turn means that there is a rule $\exists \mathbf{v}.\, \Pi : F \Rightarrow P_i\mathbf{x}$ in Φ and some stack s' such that $s', h \mathbin{\dot{-}} \mathrm{dom}(h') \models^{\alpha'}_\Phi \Pi : F$, and $s'(\mathbf{x}) = s(\mathbf{t})$ (equality of tuples), and $\alpha' < \alpha$. Trivially, $s', h \mathbin{\dot{-}} \mathrm{dom}(h') \models^{\alpha'}_\Phi \exists \mathbf{v}.\, \Pi : F$. Thus the query $R' \equiv s', h \Vdash_\Phi \exists \mathbf{v}.\, \Pi : F \rightsquigarrow h'$ is valid, constructively valued and α'-supported. By the inductive hypothesis, R' is provable and therefore so is R by applying (P_i).

The cases for $\mapsto$, emp easily treated with rule $(\mapsto)$, (emp). □