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
University College London, UK
and National Research University Higher School of Economics, Russia
M.Kanovich@ucl.ac.uk

Reuben Rowe
University College London, 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.

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 “sub-models” 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 CYCLOIDS 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. Symbolic-heap Separation Logic with Inductively Defined Predicates

In this section we present our fragment $\text{SL}_{\text{ID}}^\text{SH}$ 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. $x$ for $(x_1, \ldots, x_n)$. We write $\text{proj}_j$ for the $j$th projection function on tuples, and we often abuse notation by treating a tuple $x$ as the set containing exactly the elements occurring in $x$. If $X$ and $Y$ are sets, we write $X \neq Y$ as a shorthand for $X \cap Y = \emptyset$.

2.1 Syntax

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

Definition 2.1 (Symbolic heap). Spatial formulas $F$ and pure formulas $\pi$ are given by the following grammar:

$$F ::= \text{emp} \mid x \mapsto t \mid \Pi \mid F \circ F \quad \pi ::= t = t \mid t \neq t$$

where $x$ ranges over variables, $t$ over terms, $P$ over predicate symbols and $\pi$ over tuples of terms (matching the arity of $P$ in $\Pi$).

A symbolic heap is given by $\exists x. \Pi : F$, where $x$ is a tuple of (distinct) variables, $F$ is a spatial formula and $\Pi$ is a finite set of pure formulas. Whenever one of $\Pi, F$ is empty, we omit the colon. We write $\text{FV}(A)$ 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 $\text{FV}(A)$.

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

For convenience, we sometimes drop existential quantifiers from inductive rules $A = Px$; in that case, any variables occurring in $A$ but not in $x$ are implicitly existentially quantified.

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

2.2 Semantics

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

A stack is a function $s : \text{Var} \to \text{Val}$; we extend stacks to terms by setting $s(\text{nil}) = \text{def} \text{nil}$, 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 $h : \text{Loc} \to (\text{Val} \text{List})$ mapping finitely many locations to records, i.e. arbitrary-length tuples of values; we set $\text{dom}(h)$ to be the set of locations on which $h$ is defined, and $e$ to be the empty heap that is undefined on all locations. We write $s$ 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 $\text{dom}(h_1) \neq \text{dom}(h_2)$, and undefined otherwise. Finally, we define the cover of a heap $h$ as

$$\text{cover}(h) = \text{def} \text{dom}(h) \setminus \{b \in \text{Val} \mid b \in h(a), a \in \text{dom}(h)\},$$

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

Definition 2.3. Let $\Phi$ 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 $A$:

$$s, h \models_{\Phi} H_1 \circ H_2 \iff s(\pi) = s(t)$$

$$s, h \models_{\Phi} t = t \iff s(t_1) = s(t_2)$$

$$s, h \models_{\Phi} x \mapsto t \iff s(t_1) \neq s(t_2)$$

$$s, h \models_{\Phi} \text{emp} \iff s(t_1) = t$$

$$s, h \models_{\Phi} \Pi \iff s(t_1, t_2) \subseteq \Pi$$

$$s, h \models_{\Phi} F_1 \circ F_2 \iff \exists h_1, h_2. h_1 = h_1 \circ h_2$$

$$s, h \models_{\Phi} \text{emp} \iff h \models_{\Phi} F_1$$

$$s, h \models_{\Phi} \exists x. \Pi : F \iff \exists (\text{Val}^*) \cdot \Pi[s \mapsto a], h \models_{\Phi} F$$

where the semantics $[\Pi]^F_{\Phi}$ of the inductive predicate $P$ under $\Phi$ is defined below.

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

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

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

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

$$\varphi_{ij}(Y) = \{ (s(x), h) | s, h \models \Pi : F \}$$

where $Y \in \tau_1 \times \ldots \times \tau_n$ and $\models Y$ is the satisfaction relation defined above, except that $[P_i]^Y = \text{proj}_i(Y)$. Then we finally define the tuple $[P]^\Phi = \tau_1 \times \ldots \times \tau_n$ by:

$$[P]^\Phi = \mu Y. (U_j \varphi_{ij}(Y), \ldots, U_j \varphi_{in,j}(Y))$$

where $\mu$ is the least fixed point constructor. We write $[P]^\Phi$ as an abbreviation for $\text{proj}_i([P]^\Phi)$.

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

### 3. A Model Checking Algorithm for SL$^{SH}_{ID}$

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

**Model checking problem (MC).** Given an inductive rule set $\Phi$, 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 $A = Px$, for $P$ an inductive predicate, then the general case follows almost immediately:

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

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

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

$$s, h \models \Phi Px \iff (s(x), h) \in [P]^\Phi \iff (a, h) \in [P]^\Phi$$

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

$$s, h \models \Phi A \iff s, h \models \Phi Px \iff (s(x), h) \in [Q]^\Phi'$$

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 $P, x$, 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 $e \in \text{Val}$, we should have both $(a, c, h_1) \in [P]^\Phi$ and $(c, b, h_2) \in [P]^\Phi$, where $h = h_1 \cup h_2$; 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 $h_1, h_2$ are empty.)

Therefore, we adopt a bottom-up approach: we attempt to compute all tuples in $[P]^\Phi$ that are “sub-models” of $(a, h)$, by iteratively applying the inductive rules until we reach a fixed point. (In fact, we have to do this for all inductive predicates $P$ 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 $(a, h) \in [P]^\Phi$ can be generated by applying the inductive rules in $\Phi$ to sub-models of $(a, h)$? 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 $b \in \text{Val}$, 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 $a$ or $h$.

Fortunately, as we show (Lemma 3.7), only finitely many such spare values are needed for any given rule set $\Phi$; 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 $b, h' \in [P]^\Phi$ that are sub-models of a given $(a, h)$, where “sub-model” means that $h' \subseteq h$ and $b$ consists of values from $a, \text{cover}(h)$, the null value $nil$ and a suitably chosen set of “spare” values.

**Definition 3.2.** Let $\Phi$ be an inductive rule set, $a$ a tuple of values (from $\text{Val}$) and $h$ a heap. We define the set $\text{Good}(a, h)$ (of “good sub-model values for $(a, h)$”) by:

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

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

**Definition 3.3.** Let $\Phi$ be an inductive rule set, $a$ a tuple of values and $h$ a heap. For each inductive rule $\Phi_{ij} \in \Phi$ of the form

$$\exists z. \Pi : F \Rightarrow P_i x,$$

we define an operator $\psi_{ij} : \tau_1 \times \ldots \times \tau_n \rightarrow \tau_i$...
as follows:
\[
\psi_i,j(Y) := \text{def} \begin{cases} 
(s(x), h') \; | \; s, h' \models \Pi \; \text{and} \; \Phi \; \text{and} \\
\text{where Good}(a, h) \; \text{and} \; \text{Spare}_\Phi(a, h) \; \text{are} \; \text{the} \; \text{sets} \; \text{of} \; \text{values} \; \text{given} \; \text{by} \; \text{Definition} \; 3.2. \; \text{It} \; \text{should} \; \text{be} \; \text{clear} \; \text{that} \; \text{each} \; \psi_i,j \; \text{is} \; \text{a} \; \text{monotone} \; \text{operator}.
\end{cases}
\]

We proceed by fixed point induction on the tuple of sets MC*\((a, h)\) in \(\tau_1 \times \ldots \times \tau_n\).

MC*\((a, h)\) := \mu Y. \(\bigcup_j \psi_j(Y)\) \cup \psi_n(Y)

We write MC*\((a, h)\) as an abbreviation for projj(MC*\((a, h)\)).

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

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

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

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

\[
\exists z. \; \Pi : y_1 \rightarrow u_1 \ldots \cdot y_k \rightarrow u_k \cdot P_1 x_1 \ldots \cdot P_{m} x_m \Rightarrow P(x).
\]

By construction of \(\psi_i,j(Y)\), there is a stack \(s\) with \(s(x) = b\) and \(s, h' \models \Pi \; \text{and} \; \Phi \; \text{and} \; h_i \models \psi_i,j(Y) \; \forall y_k \in \Phi\). Thus, by the induction hypothesis, \(s, h_{i+1} \models \psi_i,j(Y) \; \forall y_k \in \Phi\). That is, in particular, for any \(1 \leq i \leq m\) we have \(s(x_i), h_{i+1} \in Y_i\) and thus, by the induction hypothesis, \(s(x_i), h_{i+1} \in [P]^\Phi\). That is, in particular, for any \(1 \leq i \leq m\) we have \(s(x_i), h_{i+1} \models \psi_i,j(Y) \; \forall y_k \in \Phi\). Therefore, \(s, h' \models \psi_i,j(Y) \; \forall y_k \in \Phi\).

Next, we must show that MC*\((a, h)\) contains all \((b, h') \in [P]^{\Phi}\) that are “sub-models” of \((a, h)\) in \([P]^{\Phi}\). To do this, we need to argue that for any element \((b, h') \in [P]^{\Phi}\) that is “almost a sub-model” of \((a, h)\), in that \(h' \subseteq \text{Spare}_\Phi(a, h)\), there are corresponding submodels in MC*\((a, h)\), obtained by substituting “spare” values for “bad” ones. The following definition captures the relevant notion of substitution.

**Definition 3.5.** A finite partial function \(\theta\) : Val \rightharpoonup Val is called a **substitution for MC*\((a, h)\)** if it is injective, and, for all \(b \in \text{dom}(\theta)\),

\[
\text{if } b \in \text{Good}(a, h) \; \text{and} \; \theta(b) \in \text{Spare}_\Phi(a, h) \; \text{if} \; b \notin \text{Good}(a, h).
\]

Next, the following technical lemma, which will be crucial to completeness, captures the fact that we can “recycle” values as needed. Roughly speaking, it says that we can extend a substitution on the values \(b\) instantiating the head of an inductive rule to a substitution on the values \(V \supseteq b\) instantiating the head of the rule

**Lemma 3.6.** Let \(\theta\) be a substitution for MC*\((a, h)\) such that \(\text{dom}(\theta) \supseteq b\) and \(V \subseteq \text{Val} a\) (finite) set of values \(b \subseteq V\) and \(|V| \leq |\text{Spare}_\Phi(a, h)|\). Let \(\text{Spare}_\Phi(a, h) \setminus \theta(b) = \{d_1, \ldots, d_n\}\) and let \(V \setminus \text{Good}(a, h) = \{e_1, \ldots, e_k\}\). Then the function \(\theta' : V \rightarrow \text{Val}\) defined by

\[
\theta'(c) := \begin{cases} 
\theta(c) & \text{if } c \in \text{Good}(a, h) \\
d_i & \text{if } c = e_i \; \text{for some } 1 \leq i \leq k,
\end{cases}
\]

is also a substitution for MC*\((a, h)\), with \(\theta'(b) = \theta(b)\).

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

First, since \(V \subseteq \text{Val}\) is finite, \(\theta'\) is indeed a finite partial function \(\forall \theta\). We argue that \(\theta'\) 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 \(\text{dom}(\theta) \supseteq b\). Thus we just need to show that the third case is well-defined, which means showing that \(k \leq m\), i.e.,

\[
|V \setminus (b \cup G)| \leq |S \setminus \theta(b)|.
\]

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

\[
|V \setminus (b \cup G)| \leq |V| - |b| \leq |S| - |\theta(b)| \leq |S\setminus \theta(b)|
\]

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

Finally, to see that \(\theta'(b) = \theta(b)\), observe that \(\theta'(c) = \theta(c)\) immediately if \(c \in b \cup G\), and if \(c \in b \cup \neg G\) then \(\theta'(c) = c = \theta(c)\), using the fact that \(\theta\) is a substitution for MC*\((a, h)\).

**Lemma 3.7 (Completeness).** Let \((b, h') \in [P]^{\Phi}\) and \(h' \subseteq h\), and let \(\theta\) be a substitution for MC*\((a, h)\) with \(\text{dom}(\theta) \supseteq b\).

Then \((\theta(b), h') \in MC^\Phi(a, h)\).

**Proof.** We proceed by fixed point induction on \([P]^{\Phi}\). That is, we assume the lemma holds for \(Y = (Y_1, \ldots, Y_n) \in \tau_1 \times \ldots \times \tau_n\), and must show it holds for \(\bigcup_j \varphi_j(Y)\). This means, assuming that \((b, h') \in \varphi_i,j(Y)\) for some inductive rule \(\Phi_i,j\), \(h' \subseteq h\) and we have a \(\theta\) satisfying the conditions of the lemma, we must show that \((\theta(b), h') \in MC^\Phi(a, h)\).

Without loss of generality, we may consider \(\Phi_i,j\) to be written in the form:

\[
\exists z. \; \Pi : y_1 \rightarrow u_1 \ldots \cdot y_k \rightarrow u_k \cdot P_1 x_1 \ldots \cdot P_{m} x_m \Rightarrow P(x).
\]

By construction of \(\varphi_i,j\), we have a stack \(s\) such that \(s(x) = b\) and \(s, h' \models \Pi : y_1 \rightarrow u_1 \ldots \cdot y_k \rightarrow u_k \cdot P_1 x_1 \ldots \cdot P_{m} x_m \).
This means that $s \models h' = h_1 \circ \ldots \circ h_{k+m}$, where $s, h_i \models y_i \rightarrow u_i$ for all $1 \leq i \leq k$, and $s, h_{k+i} \models \psi_i, x_i$ for all $1 \leq i \leq m$.

The latter two statements can be rewritten as:

$$\text{dom}(h_i) = \{s(y_i)\} \quad \text{and} \quad h_i(s(y_i)) = s(u_i) \quad \text{for all} \quad 1 \leq i \leq k,$$

and

$$\text{dom}(h_{k+i}) = \{x_i\} \quad \text{for all} \quad 1 \leq i \leq m.$$

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

Now, since $\theta'$ is injective, it is easy to see that $s \circ \theta' \models h' = h_1 \circ \ldots \circ h_{k+m}$ (where $\circ$ here denotes function composition). Additionally, since $s(y_i), s(u_i) \subseteq \text{cover}(h_i) \subseteq \text{Good}(a, h)$, we have by construction, for all $1 \leq i \leq k$,

$$\text{dom}(h_i) = \{\theta'(s(y_i))\} \quad \text{and} \quad h_i(\theta'(s(y_i))) = \theta'(s(u_i))$$

i.e. $s \circ \theta', h_i \models y_i \rightarrow u_i$.

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

$$(\theta'(s(x_i)), h_{k+i}) \in MC^b(a, h) \quad \text{for all} \quad 1 \leq i \leq m.$$ Putting everything together, we obtain

$$s \circ \theta', h' \models MC^b(a, h) \quad \text{for all} \quad 1 \leq i \leq m.$$ As $(s \circ \theta')(x \cup z) \subseteq \text{Good}(a, h) \cup \text{Spare}_k(a, h)$ by construction, we obtain by the definition of $MC^b(a, h)$ (Definition 3.3):

$$(s \circ \theta')(x, h') \in MC^b(a, h).$$

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

$$\theta(b), h' \in MC^b(a, h).$$

Lemma 3.8. For each $1 \leq i \leq n$,

$$(a, h) \in [P_i] \models MC^b(a, h).$$

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

Lemma 3.9. $MC^b(a, h)$ is finite and computable.

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

To see that $MC^b(a, h)$ 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 $MC^b(a, h)$ can be finite, there can be only finitely many such approximants. To see that each one is computable, it suffices to show that any $\psi_{i,j}(Y)$ is computable, given that $Y \in n_t \times \ldots \times n_t$ is computable and the inductive rule $\Phi_i, j$, is of the form $\exists x. \Pi : F \Rightarrow P_i, x$. say. This is quite clear: First, there are only finitely many membership candidates $(b, h')$ with $h' \subseteq h$ and $b \subseteq \text{Good}(a, h) \cup \text{Spare}_k(a, h)$. Second, since whether $s, h' \models \Pi : F$ depends only on the values $s$ assigns to the variables appearing in $\Pi : F$, for each candidate $(b, h')$ it suffices to pick an arbitrary stack $s$ with $s(x) = b$ and $s(z) \subseteq \text{Good}(a, h) \cup \text{Spare}_k(a, h)$. Finally, for any such $s, h'$ and computable $Y$ it is straightforward to decide whether $s, h' \models \Pi : F$.

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

Proof. By Proposition 3.1, it suffices to show that $MC$ is decidable. Let $(\Phi, a, h, P_i)$ be an instance of $MC$. By Lemma 3.8, deciding whether $(a, h) \in [P_i] \models MC^b(a, h)$. By Lemma 3.9, we have that $MC^b(a, h) = \psi_{i,j}(MC^b(a, h))$ is a finite and computable set. Hence it is decidable whether $(a, h) \in MC^b(a, h)$.

Remark 3.11. 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 $s, h \models \emptyset A$ 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.

Remark 3.12. In practice, we might sometimes want to consider "intuitionistic" model checking queries, of the following form: Given an inductive rule set $\Phi$, stack $s$, heap $h$ and formula $A$, decide whether there is an $h' \subseteq h$ such that $s, h' \models \emptyset A$. As in Proposition 3.1, we may assume without loss of generality that $A = P_i x$ for some predicate symbol $P_i$. This problem is clearly also decidable: letting $s(x) = a$, we simply check whether $(a, h') \in MC^b(a, h)$ for some $h'$. Correctness follows similarly to Lemma 3.8. Indeed, all of our correctness and complexity results in this paper adapt straightforwardly to intuitionistic queries.

We conclude this section by deducing some immediate consequences of Theorem 3.10 for the entailment problem in $\text{SL}_{\text{HD}}$.

Definition 3.13. Given an inductive rule set $\Phi$ and symbolic heaps $A, B$, we say the entailment $A \models B$ is valid if $s, h \models \emptyset A$ implies $s, h \models \emptyset B$ 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 3.14. For any entailment $A \models B$, the set of its countermodels, $\{(s, h) \mid s, h \models \emptyset A \land s, h \not\models \emptyset B\}$, is recursively enumerable.

Proof. First, the set of all heaps is recursively enumerable, since heaps are finite objects. Second, although stacks are not finite objects, it clearly suffices to enumerate only the values of $s$ on the finite set of variables $\text{FV}(A) \cup \text{FV}(B) = x$, say. Thus we can recursively enumerate all "representative candidates" of the form $(s(x), h)$. Finally, for any such candidate model, we can decide whether $s, h \models \emptyset A$ and $s, h \not\models \emptyset B$ by Theorem 3.10.

Corollary 3.15. For any inductive rule set $\Phi$, the set of invalid entailments over $\Phi$ is recursively enumerable.

Proof. The set of all symbolic heaps over $\Phi$ is recursively enumerable, so the set of all entailments $A \models B$ is also enumerable. Next, note that the set of countermodels of a given entailment is enumerable (Corollary 3.14). Thus the invalid entailments are recursively enumerable simply by enumerating all entailments and, for each of these, dovetailing the process of searching for a countermodel.
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 $\|\|$ 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 $\mathsf{L}_{\mathsf{H}0}$, which was shown to be EXPTIME-hard in [10]: given an inductive rule set $\Phi$ containing no occurrences of $\rightarrow$, and a predicate symbol $P$ from $\Phi$ of arity 0, decide whether there exists a model $s, h$ such that $s, h \models_P P$. Since $\Phi$ contains no occurrences of $\rightarrow$ and $P$ no free variables, this means deciding whether $e \in [P]^b$ (recall $e$ is the empty heap). Thus, given any instance $(\Phi, P)$ of the above problem, the corresponding instance of RMC is simply given by $(\Phi, (), e, P)$. $\square$

**Lemma 4.2.** MC $\in$ EXPTIME.

**Proof.** By Proposition 3.1, it suffices to show RMC $\in$ EXPTIME. By Lemma 3.8, deciding a given instance $I = (\Phi, a, h, P)$ of RMC can be done by computing $MC^\Phi(a, h)$ and checking whether $(a, h) \in \text{proj}_2(MC^\Phi(a, h))$. Thus, it suffices to show that $MC^\Phi(a, h)$ can be computed in time exponential in $m = \text{def} \|I\|$.

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

$$MC^\Phi(a, h) = \text{fix } \mu Y. (\bigcup_j \psi_j(Y), \ldots, \bigcup_j \psi_{n,j}(Y))$$

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

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

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

Next, we obtain an upper bound for $T$. Given $Y$ and the inductive rule $\Phi_{i,j}$ of the form $\exists x.J : F \Rightarrow P[x]$, say, it clearly suffices, for any “candidate” $(b, h')$ of the above form, to decide whether or not $(b, h') \models_{\psi_{i,j}(Y)}$. This means checking whether $h' \subseteq h$ and, for every valuation of the variables $x \cup z$ into $\text{Good}(a, h) \cup \text{Spare}_g(a, h)$, checking whether $s(x) = b$ and $s, h' \models_{\psi_{i,j}} J : F$ (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 $s, h' \models_{\psi_{i,j}} J : F$ 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 $2^{\|\Phi\|}$ such combinations. Finally, as $Y$ might contain up to $N$ elements, checking whether a chosen division of $h'$ satisfies $F$ with respect to $Y$ might take up to $N$ steps. All other checks are polynomial, so we obtain

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

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

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

**Proof.** Immediate by Lemmas 4.1 and 4.2. $\square$

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 $\Phi$.

**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 $\Phi_{PT}$:

$$x \mapsto \text{nil} \Rightarrow V(x)$$

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

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

$$E(x, y) \ast J \Rightarrow J \quad \text{emp} \Rightarrow J \quad T \ast J \Rightarrow J \quad P \Rightarrow J \Rightarrow P$$

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 $\text{dom}(h_G) = \{a_1, \ldots, a_n, b_1, \ldots, b_{2m}\}$, as follows:

$$h_G(a_i) = \text{nil} \quad \text{for } 1 \leq i \leq n,$$

$$h_G(b_{2k}) = \{a_i, a_j\} \quad \text{for } 1 \leq k \leq 2m \text{ and } c_k = \{v_i, v_j\},$$

$$h_G(b_{2k+1}) = \{a_i, a_j\} \quad \text{for } 1 \leq k \leq 2m \text{ and } c_k = \{v_i, v_j\}.$$
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)equation 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 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 summarizes our results:

<table>
<thead>
<tr>
<th></th>
<th>CV</th>
<th>DET</th>
<th>CV+DET</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-MEM</td>
<td>EXPTIME</td>
<td>EXPTIME</td>
<td>EXPTIME</td>
</tr>
<tr>
<td>MEM</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>≥ PSPACE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$\Pi : \text{emp} \Rightarrow P_1, x, \ldots, x \Rightarrow P_n.$$  

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.

$$x = y : \text{emp} \Rightarrow ls(x, y)$$

$$\exists z. x \Rightarrow z \ast ls(z, y) \Rightarrow ls(x, y)$$  \hspace{1cm} (1)

and

$$x = y : \text{emp} \Rightarrow rls(x, y)$$

$$\exists z. x \neq y : rls(x, z) \ast z \Rightarrow y \Rightarrow rls(x, y)$$  \hspace{1cm} (2)

Definition 5.4 (CV). A variable $z$ occurring in an inductive rule $\exists y. \Pi : F \Rightarrow P_x$ is said to be constructively valued in that rule if:

- (a) $z \in x$, 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
  - $w \hookrightarrow u$ is a subformula of $F$ and $z \in u$.

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 $P_1$, is said to be deterministic (in an inductive rule set $\Phi$) if for any two distinct rules of the form

$$\exists z. \Pi : F \Rightarrow P_1, x \quad \text{and} \quad \exists z'. \Pi' : F' \Rightarrow P_1, x,$$

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

We note that whether $\Phi$ 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 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 $(a, h) \in \lfloor \mathcal{P} \rfloor^\mathcal{P}$, we repeatedly unfold the rules from $\Phi$ backwards (from head to body), instantiating variables with values and matching 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 $Q(x_1, \ldots, x_n)$, we write $\overline{Q}$ to denote a statement $\overline{Q}(a_1, \ldots, a_n)$ obtained by instantiating the variables in $Q$ with values and heap pointers in the obvious way. For example, $\overline{x} \mapsto \overline{y}$ represents the one-cell heap that contains $\overline{y}$ at location $\overline{x}$. Then we specify the unfolded inductive tree by induction (see Figure 1):

(a) Let $\overline{Q}_m$ be generated by an instantiated rule $\rho_m$ of the form

$$\overline{\Pi}_m : h_m \Rightarrow \overline{Q}_m.$$ Then we make a tree, $\overline{T}_{\overline{Q}_m}$, consisting of one edge labelled by $\rho_m$; the root is labelled by $\overline{Q}_m$, the leaf by $h_m$.

(b) Suppose that an instantiated rule $\rho$ of the form

$$\overline{\Pi} : \overline{Q}_0 \ast \overline{Q}_1 \ast \ldots \ast \overline{Q}_m \Rightarrow \overline{R}$$

generates $\overline{R}$, and $\overline{T}_{\overline{Q}_0}, \overline{T}_{\overline{Q}_1}, \ldots, \overline{T}_{\overline{Q}_m}$ are inductive trees having been already constructed for $\overline{Q}_0, \overline{Q}_1, \ldots, \overline{Q}_m$, resp.

Then we make a tree, $\overline{T}_{\overline{R}}$, by taking a new root with $m+1$ outgoing edges $\overline{\rho}_0$ labelled by $\rho$, labelling the root by $\overline{R}$, and connecting our root with the roots of $\overline{T}_{\overline{Q}_0}, \overline{T}_{\overline{Q}_1}, \ldots, \overline{T}_{\overline{Q}_m}$, resp.

![Figure 1. An unfolded inductive tree for $\overline{R}$.](image-url)
Proposition 6.2. The restricted model checking problem RMC can be solved by an exhaustive search for an unfolded inductive tree for the query \((a, h) \in [P]^h\), where the values for instantiated existential variables are drawn from the set \(\text{Good}(a, h) \cup \text{Spare}_e(a, h)\), 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 \(\text{Good}(a, h) \cup \text{Spare}_e(a, h)\).

However, the above procedure might still generate an exponential number of leaves labelled by the empty heap \(e\).

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

\[
P_1 \ast P_2 \Rightarrow P_0, \\
P_{2j+1} \ast P_{2j+2} \Rightarrow P_{2j-1}, \quad P_{2j+1} \ast P_{2j+2} \Rightarrow P_{2j}, \\
\text{emp} \Rightarrow P_{2n+1}, \quad \text{emp} \Rightarrow P_{2n+2}.
\]

Then any unfolded inductive tree for the query \(s, e \models u, P_0\) 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 \(|\text{dom}(h)|\).

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 \(\Phi\), tuple of values \(a, h\), heap \(P\) and predicate symbol \(P\), whether \((a, h) \in [P]^h\).

Proof. (Sketch) Taking into account the bounds provided by Section 3, we look for an unfolded inductive tree such that within each rule instance \(\rho\) from \(\Phi\) 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 3.6, the number of leaves labelled by \(e\) can be bounded by the size of \(|\text{dom}(h)|\) when all rules are memory-consuming.

Lemma 6.5. According to Section 3, \((a, h) \in [P]^h\) if and only if there is an unfolded tree for the appropriately instantiated \(\Phi\) such that \(h\) is the \(\ast\)-composition of all heaps labelling the leaves of the tree.

Lemma 6.6. The number of nodes in the above inductive trees for \(P\) is bounded by \(2((m+1) \cdot |\text{dom}(h)|\), 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 \(|\text{dom}(h)|\). 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 \(\rho\), providing thereby a leaf \(v'\) with its incoming edge \((v_0, v')\) labelled by the same \(\rho\), such that \(v'\) is labelled by a non-empty \(\tilde{t} \Rightarrow t\) (Figure 2 shows such a \(\rho\) in MEM). Since no more than \(m\) leaves labelled by \(e\) can be associated with one and the same \(v'\) specified in a way, the total number of leaves is bounded by \((m+1) \cdot |\text{dom}(h)|\)!

It remains to apply an induction to conclude the proof.

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].

Figure 2. \(\rho = \tilde{t} \Rightarrow \tilde{Q}_1 \ast \tilde{Q}_2 \ast \cdots \ast \tilde{Q}_m \Rightarrow \tilde{R}\)

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 \(\ell\), with leading variable \(x\), we mean a formula of the form \((x_1', x_2', \ldots, x_{\ell}'\text{ are fresh}):\)

\[
x \Rightarrow x'_1 \ast x'_1 \Rightarrow x'_2 \ast x'_3 \Rightarrow \cdots \Rightarrow x'_{\ell-1} \Rightarrow x'_{\ell} \ast x'_{\ell} \Rightarrow x'_{\ell+1}
\]

Given a \(3\)-partition problem instance, i.e., a bound \(B\) and a multiset \(S = \{s_1, s_2, \ldots, s_{3m}\}\), we introduce a linear-ordered list \(X\) of distinct variables: \(X = x_1, x_2, \ldots, x_{3n}, x_{3n+1}, \ldots, x_{3m}\). 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 \(S\) is encoded as a concrete heap \(h_S\), which is a collection of \(3m\) disjoint “circular” lists of the form \(S_i(x_i)\).

With an appropriate stack \(s_s, s_S, h_S = \Pi(X)\) : \(\varphi_S(X)\), where \(\varphi_S(X) = S_1(x_1) \ast S_2(x_2) \ast \cdots \ast S_m(x_m)\) and \(\Pi(X)\) says \((x \neq y)\) for all distinct variable names \(x\) and \(y\) mentioned, explicitly or implicitly, in \(\varphi_S(X)\).

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

\[
x \neq \text{nil} : \text{emp} \Rightarrow Q_i(x)
\]

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

\[
x \neq y, y \neq z, z \neq x : \text{emp} \Rightarrow R_{ijk}(x, y, z)
\]

In the case of \(i < j < k\) and \(s_i + s_j + s_h = B\), we add the rule:

\[
S_i(x) \ast S_j(y) \ast S_k(z) \ast R_{ijk}(x, y, z) \Rightarrow R_{ijk}(x, y, z)
\]

Lemma 6.8. Let \(h_S, s_S, s_S\) be defined above. Then

\[
s_S, h_S \models \Pi(X) \ast \bigwedge_{i=1}^{3m} Q_i(x_i) \ast \bigwedge_{i<j<k} R_{ijk}(x_i, x_j, x_k)
\]

iff there is a complete \(3\)-partition on \(S\) - i.e., \(S\) can be partitioned in groups of size \(3\), say \(s_i, s_j, s_k\), so that \(s_i + s_j + s_k = B\).

Proof. Each \(R_{ijk}(a_1, a_2, a_3)\) at the top is generated either by (4), with \(\text{emp}\), or by (5), with \(S_i(a_1), S_j(a_2), S_k(a_3)\) 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).

(b) 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 \(\text{fixed} - \) 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.
Here we abbreviate $X = x_0, x_1, ..., x_n$, $Q = q_0, q_1, ..., q_{|Q|}$, $\Xi = \xi_0, \xi_1, ..., \xi_m$. $\Pi_i = \{ x_j \neq q_i \mid j \neq i, q_i \in Q \} \cup \{ z \neq z' \mid z, z' \in Q \cup \Xi \}$

The final “empty configuration” is generated by the “backward” rule (recall that $\xi_0$ is the blank symbol, $\xi_1$ and $\xi_2$ are end markers)

$F_0, x_0 = q_0, x_1 = \xi_1, x_2 = x_3 = \cdots = x_{n-1} = \xi_0, x_n = \xi_2; \text{ emp } \Rightarrow T(X, Q, \Xi)$

An instruction “if in state $q_i$ looking at $\xi$, replace by $\xi_j$, move to the right, and go into $\xi_k$”, is simulated by $n$ rules (here $0 \leq i < n$):

$F_i, x_i = q_i, x_{i+1} = \xi_i, y_i = \xi_j, y_{i+1} = q_k; \quad T(x_0, x_1, ..., x_{i-1}, y_i, y_{i+1}, x_{i+2}, ..., x_n, Q, \Xi) \Rightarrow T(X, Q, \Xi)$

An instruction “if in state $q_i$ looking at $\xi$, replace by $\xi_j$, move to the left, and go into $\xi_k$”, is simulated in a similar “backward” way.

An alternating instruction “if in state $q_i$ run two copies of the configuration in parallel but with states $\xi_k$ and $\xi_j$, resp.”, is simulated by $(0 \leq i < n)$:

$F_i, x_i = q_i, y_i = \xi_k, z_i = \xi_j; \quad T(x_0, x_1, ..., x_{i-1}, y_i, y_{i+1}, z_i, x_{i+2}, ..., x_n, Q, \Xi) \Rightarrow T(X, Q, \Xi)$

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

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

| $x = x_0 : \text{ emp } \Rightarrow RLS(x, u, y)$ |

| $\exists u \neq x : RLS(x, u, y) \Rightarrow RLS(x, y, z)$ |

Notice that the rules for $RLS(x, u, y)$ are both memory-consuming and deterministic. The fact that $u$ is not constructed 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.:

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

**Lemma 6.11.** Assuming $a \neq b$, let $h_{ab}(0) \models_{RLS} RLS(a, c, b)$ for some c. Then: $(\langle \delta = 0 \rangle \land (a = c \neq b))$.

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

Let $\varphi \equiv (C_1 \land \cdots \land C_m)$ be a formula of $m$ clauses over linear ordered $n$ propositional variables $p_1, p_2, ..., p_n$, and each $C_j$ is of the form (here, for any $a$, we denote $q^0 = q = q^1 = q$ and $q^0 \neq = q^1$):

$C_j(q_1, q_2, q_3) \equiv \langle q^1_1 \lor q^2_2 \lor q^3_3 \rangle$

Each $C_j$ is encoded by a predicate $C_j(a_1, \gamma_1, a_2, \gamma_2, a_3, \gamma_3)$ with the following rule $\xi_j$ (for the sake of readability, we squeeze three deterministic rules into one but with disjunction):

$((a_1 \neq \gamma_1) \lor (a_2 \neq \gamma_2) \lor (a_3 \neq \gamma_3)) \land \land_k a \neq \gamma_k \Rightarrow C_j(a_1, \gamma_1, a_2, \gamma_2, a_3, \gamma_3)$

**Example 6.12.** Here we represent $p_i$ as "$x_i \neq u_i$". So satisfiability of $C(p_1, p_2, p_3)$ of the form $(p_1 \lor \neg p_2 \lor p_3)$ is reformulated as $(x_1 \neq u_1) \lor (x_2 = u_2) \lor (x_3 \neq u_3)$.

We take the following linear-ordered variables: $W = w_{0n}$, $X = x_1, x_2, ..., x_n$, $U = u_1, u_2, ..., u_n$, $Y = y_1, y_2, ..., y_n$.

The challenge - to maintain the arity fixed and, at the same time, to "keep i-th position" inside the long $X, U, Y$ is solved by taking predicates $Q_i(x, u, y)$ with the rules $\kappa_i; x \neq y, u = x : \text{ emp } \Rightarrow Q_i(x, u, y), \quad x \neq y, u = y : \text{ emp } \Rightarrow Q_i(x, u, y)$.

The key points of our reduction are encapsulated in the rule $\omega_{ok}$:

$\exists X, U, Y, W \left( \Pi(X, Y, W) : w_0 \Rightarrow w_0 \ast \right.$

$\ast_{i=1}^{2n}Q_i(x_i, u_i, y_i) \ast_{j=1}^{2n}RLS(x_i, u_i, y_i) \ast_{j=1}^{m}C_j(x_{i_1}, u_{i_1}, x_{i_2}, u_{i_2}, x_{i_3}, u_{i_3}) \Rightarrow \omega_{ok}$

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

**Definition 6.13.** Define a heap $H_{RLS}$ as a collection of $n$ disjoint heaplets of the form $h_{ab}(0)$, and $n$ disjoint heaplets of the form $h_{ab}(1)$ (we assume $a \neq b$), and a heap $H_Q$ as a loop of the form $d_0 \Rightarrow d_0$.

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

$$(((), H_{RLS} \cup H_Q) \in [0k]_{\omega_{ab}} \cup RLS \cup \cup_{i=1}^{n}I_i \cup_{i}^{\infty} \kappa_0^{\infty}$$

if and only if $(C_1 \land \cdots \land C_m)$ is satisfiable.

**Proof.** (Sketch) The $(\Rightarrow)$ direction is the hardest. Suppose that (6) is valid. Then ok is generated by $\omega_{ab}$ 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 $H_{RLS}$ can be partitioned into $2n$ disjoint heaplets of the form $h_{ab}(1)$ so that we get the following:

$\langle a_1 \neq \gamma_1 \rangle \land \land_k a \neq \gamma_k \Rightarrow RLS(a_1, c_2, b_2)$

Now, using the rules $\xi_j$ and $\kappa_i$, and Lemma 6.11, we can prove the desired satisfiability (see also Example 6.12): $\varphi(\delta_1, \delta_2, ..., \delta_{n-1}, 1) = 1$.

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

$$\exists I_i : Q_i u_1 \ast Q_2 u_2 \ast \cdots \ast Q_m u_m \Rightarrow P x$$

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 $q_0, q_1, ..., q_{|Q|}$, and tape symbols $\xi_0, \xi_1, ..., \xi_m$. Here $\xi_0$ is the initial state, $\xi_i$ is an accept state, $\xi_0$ is the blank symbol, and $M$ acts in the space $n$ between two unused markers, say $\xi_1$ and $\xi_2$.

Let $M$ always jump, and no $M$’s instruction starts with $q_0$. By $(\eta_1, \eta_2, ..., \eta_{n-1}, \eta_0)$ we formalize that “in state $q_i$, M scans i-th square, when $\eta_i, \eta_{i-1}, \eta_{i-1}$, ..., $\eta_0$ is printed on its tape”. We encode $M$ by means of the rules $\Phi_M$ given in Figure 3, where the “tape” predicate $T$ depicts M’s configurations:

$T(x_0, x_1, ..., x_n, q_0, q_1, ..., q_{|Q|}, \xi_0, \xi_1, ..., \xi_m)$

configuration states tape symbols

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

□
All variables in \( \Phi_M \) occur in \( X \cup Q \cup \Xi \), hence, they are constructively valued, which provides item (a) in Theorem 6.16.

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

To answer the challenging (c), \( M \)'s non-deterministic instruction “if in state \( \hat{q}_a \), looking at \( \xi \), go either into \( \hat{q}_{a_1} \) or into \( \hat{q}_{a_2} \)” is simulated by the deterministic rules (here \( 0 \leq i < n, \ell = 1, 2 \)):

\[
\exists y. \Pi_i, x_i = x_{i+1} = \xi, y \notin \Xi, y_i = 1 = \xi_i, T(x_0, \ldots, x_{i+1}, y, y_i, \ldots, x_{n+1}, \Xi, \Xi) \Rightarrow T(X, \Xi, \Xi), \\
\Pi_i, x_i = \hat{q}_{a'_i}(s, k, k_1, \ldots, k_r, \Xi) \quad \Rightarrow \quad T(X, \Xi, \Xi),
\]

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’ \( \hat{q}_{a'_i} \). To continue a computation, the lucky guess should be one of our specially introduced states \( \hat{q}_{a''_i}(s, k, k_2, \Xi) \) and \( \hat{q}_{a''_i}(s, k, k_2, \Xi) \).

As a result, \( M \) finishes either in \( \hat{q}_{a_i} \) or in \( \hat{q}_{a_2} \).

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

### 6.5 Polynomial Time for MEM+DET

We next show that the model checking problem is in PTIME whenever \( \Phi \) is in MEM+DET. Essentially, predicates in \( \Phi \) are labelled by inference rules accordingly. We say that a reduction \( \alpha \) is valid then it is provable.

A ≺ 3.

Let \( R \) be the form of \( s, h \vdash \Phi \), \( \exists \mathbf{A} \sim h' \). As \( R \) is constructively valued, there must be some variable \( x \) in \( v \) 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 \( y \mapsto t \) of \( A \). Let \( v = (x) \). As \( R \) is valid we have \( h' \subseteq h \) and \( h \vdash \text{dom}(h') = \emptyset \wedge \forall A \). Thus there is a stack \( s' \) such that \( s'(\forall \exists \mathbf{A}) = s(\forall \exists \mathbf{A}) \) and \( s', h \vdash \text{dom}(h') \subseteq \emptyset \). 3.

If clause (i) is true then clearly \( s'(x) = (y) \) by the semantics of \( = \), and \( s'(y) = (y) \) as \( y \) is free. Thus without loss of generality we can set \( s' = s[x \mapsto s(y)] \). Therefore the reduction \( R' \equiv s[x \mapsto s(y)], h \vdash \exists \mathbf{A} A \sim h' \) is valid and constructively valued. It is easy to see that \( R' \) must be \( \alpha \)-supported (otherwise we can derive a contradiction with the assumption that \( \alpha \) is the least such ordinal), thus \( R' \sim R \) by clause (2c) of the definition of \( \prec \).

By the inductive hypothesis, \( R' \) is provable and the rule (\( \exists \mathbf{A} \)) can be applied, thus proving \( R \). Clause (ii) is similar and uses (\( \exists \mathbf{A} \)).

Next, let \( R \in \mathcal{R} \) be of the form \( s, h \vdash \Phi \), \( \alpha \vdash h' \). Thus \( s' \subseteq h \) and \( h, h \vdash \text{dom}(h') \subseteq \emptyset \). As \( \alpha \) is valid we have \( s \vdash h' \subseteq h \) and \( h, h \vdash \text{dom}(h') \subseteq \emptyset \). Thus \( h = h \cup h \vdash h' \), and \( s, h \vdash \text{dom}(h') \subseteq \emptyset \). Therefore it follows that the reduction \( R' \equiv s[h \vdash h], h \vdash \exists \mathbf{A} : \mathbf{F} \vdash h' \) is valid, constructively valued and \( \alpha \)-supported. By clause (2a) of the definition of \( \prec \) it follows that \( R' \sim R \). 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 \( s, h \vdash \Phi \), \( \alpha \vdash h' \). Thus \( h' \subseteq h \) and \( h, h \vdash \text{dom}(h') \subseteq \emptyset \). Therefore there are two disjoint heaps \( h_0, h_F \) such that \( h = h_0 \circ h_F \), and \( s, h_0 \vdash \alpha \circ \sigma \) and \( s, h_F \vdash \alpha \circ \sigma \), where \( \alpha = \max(\sigma, \alpha_F) \).

Thus \( h = h_0 \circ h_F \), and \( s, h \vdash \text{dom}(h') \subseteq \emptyset \). Therefore the reductions \( R_0 \equiv s, h \vdash \alpha \circ \sigma \circ h_F \) and \( R_F \equiv s, h_F \vdash \alpha \circ \sigma \circ h' \) are both valid and constructively valued. In addition, \( R_0 \vdash R_F \vdash R \); either \( \alpha_F < \alpha \) (resp., \( \alpha_F < \alpha \)) where clause (1) of the definition of \( \prec \) applies, or \( \alpha_F = \alpha \) (case (2b) applies. By the inductive hypothesis, \( R_0 \) and \( R_F \) are provable, and so it \( R \) via (\( \prec \)).

Finally, suppose \( R \in \mathcal{R} \) has the form \( s, h \vdash P_t \sim h' \), meaning that \( h' \subseteq h \) and \( h, h \vdash \text{dom}(h') \subseteq P_t \). By Def. 3.3, this means that \( (s, t)(h, h \vdash \text{dom}(h')) \in [P_t]^{\emptyset} \) which in turn means that there is a rule \( \exists \mathbf{A} : \mathbf{F} \vdash P_t \) in \( \Phi \) and some stack \( s' \) such that \( s', h \vdash \text{dom}(h') \subseteq \emptyset \). \( : \mathbf{F} \vdash s(t)(x) \in \text{range}(\text{dom}(h')) \), (equality of tuples), and \( \alpha' < \alpha \). Trivially, \( s', h \vdash \text{dom}(h') \subseteq \emptyset \). Therefore the query \( R' \equiv s[h \vdash h'], h \vdash \exists \mathbf{A} : \mathbf{F} \vdash h' \) is valid, constructively valued and \( \alpha' \)-supported. By the inductive hypothesis, \( R' \) is provable and therefore so is \( R \) by applying (\( P_t \)).

The cases for \( \rightarrow \), emp easily treated with rules (\( \rightarrow \)), (emp).
by noting that the proof search described in Theorem 6.23 actually
whether Theorem 6.23. Let
and, its runtime is naturally bounded by a polynomial in the input
First, note that as directly contradicts determinism.
and order the proof search to first prove the left-
The rule (∗) resembles a cut in that the intermediate heap
Thus the size of the proof is \( P \) in a proof
Thus, for any stack \( s \) and heap \( h \), checking whether \( s, h \models \Phi \) can be performed in polynomial time.
Proof. First, note that \( s, h \models \Phi \) iff \( s, h \models \Phi, A \) iff \( s, h \models \Phi, A \rarrow e \) is provable.
Observe that the structure of a given reduction dictates that there is at most one applicable rule from Fig. 4. Rules about quantifiers, (\( \exists_x \) and \( \exists_y \)), form 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.
As \( \Phi \) is in \( \DET \), (\( P_i \)) can be used with at most one rule and this rule can be found in polynomial time by evaluating the side condition of (\( P_i \)) for all rules. This means no back-tracking is required over \( \Phi \).
The rule (∗) resembles a cut in that the intermediate heap \( h' \) does not appear in the conclusion. This can be seen as a source of non-determinism as many choices for \( h' \) may have to be checked (due to the fact that \( h' \subseteq h \)). However, as all formulas involved are precise (Lemma 6.22) if there is such a heap \( h' \) it is unique.
In addition, observe that only axioms impose constraints on the \( \RHS \) heap. Thus, we avoid back-tracking by using meta-variables for the heap \( h' \) and order the proof search to first prove the left-hand subgoal of (\( \ast \)). If the left subgoal of (\( \ast \)) is proven, then \( h' \) is instantiated by axioms and the search continues in the right subgoal of (\( \ast \)). Otherwise, the goal reduction is clearly invalid.
These observations together guarantee that the proof search is fully deterministic.
Now, for \( \MEM \) rules each application of (\( P_i \)) leads to at least one subgoal that requires (\( \rarrow \)) and, there cannot be more instances of (\( \rarrow \)) in a proof than the size of the heap \( h \) in the root reduction. Thus the size of the proof is \( O(|h|) \), and as the search is deterministic, runtime is linear as well.

\[ \begin{align*}
\text{s(x) = s(y) & s, h \models \Pi : F \rarrow h' \quad (\ast)} \\
\text{s(x) \neq s(y) & s, h \models \Pi : F \rarrow h' \quad (\neq)} \\
\text{s(x) \in \dom(h) & h(s(x)) = s(t) \quad (\rightarrow)} \\
\text{s, h \models x = y, \Pi : F \rarrow h' \quad (=)} \\
\text{s, h \models x \neq y, \Pi : F \rarrow h' \quad (\neq)} \\
\text{s, h \models \Phi \rarrow h' \quad (\})}
\end{align*} \]

Figure 4. Proof rules for reductions. The formula \( \sigma \) in the rule (\( \ast \)) is atomic.

Remark 6.24. Deciding intuitionistic queries (cf. Remark 3.12) in \( \MEM + \CV + \DET \) can be done in \( \PTIME \). This follows easily by noting that the proof search described in Theorem 6.23 actually computes a heap for the \( \RHS \) of any reduction. Given the precision of the formulas involved (Lemma 6.22), this means that we can answer correctly intuitionistic queries using the same algorithm.

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 \( \SL2c \). 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 formulations, using hash sets to store submodels, and using bit vectors to represent heaps. We also implemented the intuitionistic version of our algorithms as per Remark 3.12 and Remark 6.24. 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.

We note that all tests carried out were on positive instances. This was decided for two reasons. First, the worst-case performance can be exhibited with positive instances as shown below. Second, when using runtime checks, for instance in code contracts or offline test suites, negative instances usually lead to program termination because they indicate that some pre- or postcondition, or invariant is no longer satisfied. Thus the runtime on positive instances is a much more important measure of performance.

‘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) ifte.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.

\[ \cup_{i \neq l} \varphi_{i,l}(Y) \]. As the rules are precise there must be \( k \neq l \) such
that \((a, h_1) \in \varphi_{i,k}(Y)\) and \((a, h_2) \in \varphi_{i,l}(Y)\). This, however, directly contradicts determinism.

Finally, we establish that proof search is a deterministic, terminating decision procedure when all predicates are in \( \CV + \DET \); and, its runtime is naturally bounded by a polynomial in the input size, if in addition, all rules are in \( \MEM \).

Theorem 6.23. Let \( \Phi \) be a formula in \( \CV \) and \( A \) a set of rules in \( \MEM + \CV + \DET \). Then, for any stack \( s \) and heap \( h \), checking whether \( s, h \models \Phi \) can be performed in polynomial time.

\[ \begin{align*}
\text{U}_i \varphi_{i,l}(Y). & \quad \text{As the rules are precise there must be } k \neq l \text{ such that } (a, h_1) \in \varphi_{i,k}(Y) \text{ and } (a, h_2) \in \varphi_{i,l}(Y). \quad \square
\end{align*} \]
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 MEM+CV+DET apart from \texttt{iter.c} which uses a non-DET predicate (describing a possibly cyclic list). The predicate describing the set data-structure in \texttt{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 MEM+CV+DET, while the others are only MEM. The \texttt{queue.c} 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. Where runtimes are not given, this indicates that we elected not to check models of the corresponding size due to the length of time required to obtain an answer; in these cases we felt that the data we collected was sufficient to analyse the performance of the algorithm.

The results, e.g. for \texttt{set.c} 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 \texttt{queue.c} and \texttt{schorr-waite.c} models are simple list segment checked against formulas containing only single predicate instances. The \texttt{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, e.g., unit test suites.

We also tested the algorithm for the CV+DET fragment in Section 6.5 on the formulas falling within it (i.e. those appearing in the left-hand side tables of Fig. 5(a) and 5(b)) and their corresponding models, whose sizes range from 0–100 heap cells. In our benchmark suite, all such formulas happen to also be in MEM, and thus fall within the PTIME fragment. 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 predicate encoding the spatial truth constant \(T\), 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. Note that while intuitionistic model checking does not incur complexity penalties (see Remark 3.12), encoding such queries does.

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

---

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

The table below shows the results of the model checking algorithm for various heap sizes and predicate encodings. The columns include the number of heap cells, the size of models, and the runtimes in milliseconds. The results are grouped by the type of encoding, with separate tables for single and multiple model per heap size, and spatial true encoding.

<table>
<thead>
<tr>
<th># Heap Cells</th>
<th>(MEM+CV+DET)</th>
<th>(MEM+DET)</th>
<th>(EXPTIME: (MEM+CV) (MEM+DET))</th>
<th># Heap Cells</th>
<th>(MEM+CV+DET)</th>
<th>(MEM+DET)</th>
<th>(EXPTIME: (MEM+CV) (MEM+DET))</th>
<th># Heap Cells</th>
<th>(MEM+CV+DET)</th>
<th>(MEM+DET)</th>
<th>(EXPTIME: (MEM+CV) (MEM+DET))</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-10</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6-10</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6-10</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

(a) Tests using single model per heap size

(b) Tests using multiple models per heap size

(c) Spatial True Encoding

---

### 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 predicate encoding the spatial truth constant \(T\), 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. Note that while intuitionistic model checking does not incur complexity penalties (see Remark 3.12), encoding such queries does.

Our second test is on a family of predicates encoding a \(n\)-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.

Acknowledgements. We wish to thank the anonymous referees for their comments, which have enabled us to improve the presentation of the paper. Thanks also to Matt Parkinson for pointing us towards the Verifast benchmarks.

Brotherston is supported by an EPSRC Career Acceleration Fellowship, and Kanovich and Rowe by EPSRC grant EP/K040049/1.

References