Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates

James Brotherston ¹  Max Kanovich ¹  Nikos Gorogiannis ²
Reuben N. S. Rowe ¹

POPL 2016, St Petersburg, Florida, USA,
Wednesday 20th January 2016

¹Programming Principles, Logic & Verification Group
Department of Computer Science, University College London

²Foundations of Computing Group
Department of Computer Science, Middlesex University
Model Checking in General

- Model checking is the problem of checking whether a structure $S$ satisfies, or is a model of, some formula $\varphi$: does $S \models \varphi$?
Model Checking in General

- Model checking is the problem of checking whether a structure $S$ satisfies, or is a model of, some formula $\phi$: does $S \models \phi$?

- Typically, $S$ is a Kripke structure representing a program, and $\phi$ a formula of modal or temporal logic describing its behaviour.
Model Checking in General

- **Model checking** is the problem of checking whether a structure $S$ satisfies, or is a model of, some formula $\varphi$: does $S \models \varphi$?

- Typically, $S$ is a Kripke structure representing a program, and $\varphi$ a formula of modal or temporal logic describing its behaviour.

- More generally, $S$ could be any kind of mathematical structure and $\varphi$ a formula in a language describing such structures.
Separation logic (SL) facilitates verification of imperative pointer programs by describing program memory.
Separation logic (SL) facilitates verification of imperative pointer programs by describing program memory.

Typically, we do static analysis: given an annotated program, prove that it meets its specification.
Separation logic (SL) facilitates verification of imperative pointer programs by describing program memory.

Typically, we do static analysis: given an annotated program, prove that it meets its specification.

When static analysis fails, we might try run-time verification: run the program and check that it does not violate the spec.
• **Separation logic** (SL) facilitates verification of imperative pointer programs by describing program memory.

• Typically, we do **static analysis**: given an annotated program, prove that it meets its specification.

• When static analysis fails, we might try **run-time verification**: run the program and check that it does not violate the spec.

• In that case, we need to compare memory states $S$ against a specification $\varphi$: does $S \models \varphi$?
• **Separation logic** (SL) facilitates verification of imperative pointer programs by describing program memory.

• Typically, we do static analysis: given an annotated program, prove that it meets its specification.

• When static analysis fails, we might try run-time verification: run the program and check that it does not violate the spec.

• In that case, we need to compare memory states $S$ against a specification $\varphi$: does $S \models \varphi$?

• We focus on the popular symbolic-heap fragment of SL, allowing arbitrary sets of inductive predicates.
Overview of our Results

For *symbolic-heap SL* with arbitrary inductive predicates:

- the model checking problem is **decidable**
For *symbolic-heap* SL with arbitrary inductive predicates:

- the model checking problem is **decidable**
- complexity is **EXPTIME**
Overview of our Results

For *symbolic-heap* SL with arbitrary inductive predicates:

- the model checking problem is **decidable**
  - complexity is **EXPTIME**

- We identify three natural syntactic criteria for restricting inductive definitions
For *symbolic-heap* SL with arbitrary inductive predicates:

- the model checking problem is **decidable**
  - complexity is **EXPTIME**

- We identify three natural syntactic criteria for restricting inductive definitions
  - These reduce the complexity to **NP** or **PTIME**
Overview of our Results

For *symbolic-heap* SL with arbitrary inductive predicates:

- the model checking problem is **decidable**
  - complexity is **EXPTIME**

- We identify three natural syntactic criteria for restricting inductive definitions
  - These reduce the complexity to **NP** or **PTIME**

- We provide a prototype tool implementation and experimental evaluation
Terms: $t ::= x | \text{nil}$
Symbolic Heaps with Inductive Predicates

Terms: 
\[ t ::= x | \text{nil} \]

Pure Formulas: 
\[ \pi ::= t = t | t \neq t \]
Symbolic Heaps with Inductive Predicates

Terms: \[ t ::= x \mid \text{nil} \]

Pure Formulas: \[ \pi ::= t = t \mid t \neq t \]

Spatial Formulas: \[ \Sigma ::= \text{emp} \mid x \mapsto t \mid P \cdot t \mid \Sigma \cdot \Sigma \]

(P a predicate symbol, \( t \) a tuple of terms)
Symbolic Heaps with Inductive Predicates

Terms: \[ t ::= x \mid \text{nil} \]

Pure Formulas: \[ \pi ::= t = t \mid t \neq t \]

Spatial Formulas: \[ \Sigma ::= \text{emp} \mid x \mapsto t \mid P t \mid \Sigma \times \Sigma \]

(P a predicate symbol, \( t \) a tuple of terms)

- \( \text{emp} \) is the empty heap
Symbolic Heaps with Inductive Predicates

Terms: \( t ::= x \mid \text{nil} \)

Pure Formulas: \( \pi ::= t = t \mid t \neq t \)

Spatial Formulas: \( \Sigma ::= \text{emp} \mid x \mapsto t \mid Pt \mid \Sigma \ast \Sigma \)

(P a predicate symbol, \( t \) a tuple of terms)

- \textbf{emp} is the empty heap
- \( \mapsto \) ("points to") denotes a \textit{pointer} to a \textit{single heap record}
Symbolic Heaps with Inductive Predicates

Terms: \[ t ::= x \mid \text{nil} \]

Pure Formulas: \[ \pi ::= t = t \mid t \not= t \]

Spatial Formulas: \[ \Sigma ::= \text{emp} \mid x \mapsto t \mid Pt \mid \Sigma \star \Sigma \]

(P a predicate symbol, \( t \) a tuple of terms)

- \texttt{emp} is the empty heap
- \( \mapsto \) ("points to") denotes a pointer to a single heap record
- \( \star \) ("separating conjunction") describes the combining of two domain-disjoint heaps
Symbolic Heaps with Inductive Predicates

Terms: $t ::= x \mid \text{nil}$

Pure Formulas: $\pi ::= t = t \mid t \neq t$

Spatial Formulas: $\Sigma ::= \text{emp} \mid x \mapsto t \mid Pt \mid \Sigma \ast \Sigma$

(P a predicate symbol, $t$ a tuple of terms)

- $\text{emp}$ is the empty heap
- $\mapsto$ ("points to") denotes a pointer to a single heap record
- $\ast$ ("separating conjunction") describes the combining of two domain-disjoint heaps

Symbolic heaps $F$ given by $\exists x. \Pi : \Sigma$ (\(\Pi\) a set of pure formulas)
• Inductive predicates defined by (finite) sets of rules of the form:

\[ \exists z. \Pi : \Sigma \Rightarrow P x \]
Inductive Definitions

- **Inductive predicates** defined by (finite) sets of rules of the form:

\[ \exists z. \Pi : \Sigma \Rightarrow P x \]

E.g. nil-terminated linked lists with root \( x \):

- \( x = \text{nil} : \text{emp} \Rightarrow \text{List} x \)
- \( \exists y. x \neq \text{nil} : x \mapsto y \ast \text{List} \ y \Rightarrow \text{List} x \)
Model Checking: Problem Statement

- Recall the general form: given a structure $S$ and a formula $\varphi$, decide whether $S \models \varphi$
Model Checking: Problem Statement

- Recall the *general* form: given a structure $S$ and a formula $\varphi$, decide whether $S \models \varphi$

- Models of symbolic heaps are pairs $(s, h)$ where:
Model Checking: Problem Statement

• Recall the general form: given a structure $S$ and a formula $\varphi$, decide whether $S \vDash \varphi$

• Models of symbolic heaps are pairs $(s, h)$ where:

  • $s$ is a stack mapping variables to heap locations / null value
• Recall the *general* form: given a structure $S$ and a formula $\varphi$, decide whether $S \models \varphi$

• Models of symbolic heaps are pairs $(s, h)$ where:
  
  • $s$ is a *stack* mapping variables to heap locations / null value
  
  • $h$ is a *heap*: a finite map from locations to heap records
Model Checking: Problem Statement

- Recall the *general* form: given a structure $S$ and a formula $\varphi$, decide whether $S \models \varphi$.

- Models of symbolic heaps are pairs $(s, h)$ where:
  - $s$ is a *stack* mapping variables to heap locations / null value
  - $h$ is a *heap*: a finite map from locations to heap records

- Given an inductive rule set $\Phi$, stack $s$, heap $h$ and symbolic heap formula $F$, we must decide whether $(s, h) \models_{\Phi} F$
Model Checking: Subtleties

\[ P \times (s, h) \]

- How do we decompose \( h \) into \( h_1; \ldots; h_n \) to match \( 1; \ldots; n \)?
- How do we pick values for the existential variables \( z \)?
- We may need values that do not even occur in \( s \) or \( h \)!
- How to prove termination of such a procedure?
- Any of the \( h_i \) could be empty!
How do we decompose $h$ into $h_1, \ldots, h_n$ to match $1, \ldots, n$?

How do we pick values for the existential variables $z$?

We may need values that do not even occur in $s$ or $h$!

How to prove termination of such a procedure?

Any of the $h_i$ could be empty!

$$\exists \Pi : \Sigma_1 \times \ldots \times \Sigma_n \Rightarrow P x \quad (s, h)$$
\[ \exists z. \bigwedge \Sigma_1 \ldots \bigwedge \Sigma_n \xrightarrow{unfold} P x \quad (s, h) \]
Model Checking: Subtleties

\[ \exists z. \Pi : \Sigma_1 \times \ldots \times \Sigma_n \overset{\text{unfold}}{\leftrightarrow} P \times (s, h) \]

- How do we decompose \( h \) into \( h_1, \ldots, h_n \) to match \( \Sigma_1, \ldots, \Sigma_n \)?
Model Checking: Subtleties

\[ \exists z. \prod : \Sigma_1 \times \ldots \times \Sigma_n \xrightarrow{\text{unfold}} P \times (s, h) \]

- How do we decompose \( h \) into \( h_1, \ldots, h_n \) to match \( \Sigma_1, \ldots, \Sigma_n \)?
- How do we pick values for the existential variables \( z \)?
\[ \exists \mathbf{z}. \Pi : \Sigma_1 \times \ldots \times \Sigma_n \xleftarrow{\text{unfold}} P \mathbf{x} \quad (s, h) \]

- How do we decompose \( h \) into \( h_1, \ldots, h_n \) to match \( \Sigma_1, \ldots, \Sigma_n \)?
- How do we pick values for the existential variables \( \mathbf{z} \)?
  - We may need values that do not even occur in \( s \) or \( h \)!
\[ \exists z. \Pi : \Sigma_1 \times \cdots \times \Sigma_n \overset{\text{unfold}}{\xrightleftharpoons{}} P x \quad (s, h) \]

- How do we decompose \( h \) into \( h_1, \ldots, h_n \) to match \( \Sigma_1, \ldots, \Sigma_n \)?
- How do we pick values for the existential variables \( z \)?
  - We may need values that do not even occur in \( s \) or \( h \)!
- How to prove termination of such a procedure?
Model Checking: Subtleties

\[ \exists \mathbf{z}. \Pi : \Sigma_1 \times \ldots \times \Sigma_n \xrightarrow{unfold} P \mathbf{x} \quad (s, h) \]

- How do we decompose \( h \) into \( h_1, \ldots, h_n \) to match \( \Sigma_1, \ldots, \Sigma_n \)?
- How do we pick values for the existential variables \( \mathbf{z} \)?
  - We may need values that do not even occur in \( s \) or \( h \)!
- How to prove termination of such a procedure?
  - Any of the \( h_i \) could be empty!
How to decide whether \((s, h) \models \phi \ F\)
How to decide whether \((s, h) \models \phi \ P x\)
Model Checking: Solution

How to decide whether \((s, h) \models \phi\) \(P\ x\)

- We give a bottom-up fixed-point algorithm which:
How to decide whether \((s, h) \models \phi P x\)

- We give a bottom-up fixed-point algorithm which:
  - only considers sub-heaps of \(h\)
How to decide whether \((s, h) \models \varphi \, Px\)

- We give a bottom-up fixed-point algorithm which:
  - only considers sub-heaps of \(h\)
  - instantiates existentially quantified variables from a well-defined finite set of values
How to decide whether \((s, h) \models_{\Phi} P x\)

- We give a bottom-up fixed-point algorithm which:
  - only considers sub-heaps of \(h\)
  - instantiates existentially quantified variables from a well-defined finite set of values
  - and computes the set of all such "sub-models" for each predicate in \(\Phi\), then checks if \((s, h)\) is in the set for \(P\)
How to decide whether \((s, h) \models \Phi \, P \, x\)

- We give a bottom-up fixed-point algorithm which:
  - only considers sub-heaps of \(h\)
  - instantiates existentially quantified variables from a well-defined finite set of values
  - and computes the set of all such "sub-models" for each predicate in \(\Phi\), then checks if \((s, h)\) is in the set for \(P\)

- We show that this procedure is complete and has EXPTIME complexity
MEM: (Memory-consuming) rule bodies may only contain predicates if they also contain explicit, non-empty memory fragments.

DET: (Deterministic) the sets of pure constraints of the rules for a given predicate $P$ are mutually exclusive with each other.

CV: (Constructively Valued) the values of the existentially quantified variables in rule bodies are uniquely determined by the parameters $x = \text{emp}$ and $y$.

$x = \text{nil} : \text{emp} \Rightarrow \text{List } x$

$\exists y. x \neq \text{nil} : x \leftrightarrow y \ast \text{List } y \Rightarrow \text{List } x$
MEM: (Memory-consuming) rule bodies may only contain predicates if they also contain explicit, non-empty memory fragments (⇒)

\[
x = \text{nil} : \text{emp} \Rightarrow \text{List}\ x \quad \exists y. x \neq \text{nil} : x \mapsto y * \text{List}\ y \Rightarrow \text{List}\ x
\]
**MEM:** (Memory-consuming) rule bodies may only contain predicates if they also contain explicit, non-empty memory fragments ($\mapsto$)

**DET:** (Deterministic) the sets of pure constraints of the rules for a given predicate $P$ are **mutually exclusive** with each other

\[
x = \text{nil} : \text{emp} \Rightarrow \text{List } x \quad \exists y. x \neq \text{nil} : x \mapsto y \ast \text{List } y \Rightarrow \text{List } x
\]
Restricting Inductive Definitions

MEM: (Memory-consuming) rule bodies may only contain predicates if they also contain explicit, non-empty memory fragments (\(\rightarrow\))

DET: (Deterministic) the sets of pure constraints of the rules for a given predicate \(P\) are mutually exclusive with each other

CV: (Constructively Valued) the values of the existentially quantified variables in rule bodies are uniquely determined by the parameters

\[
x = \text{nil} : \text{emp} \Rightarrow \text{List} \ x \\
\exists y. \ x \neq \text{nil} : \ x \leftrightarrow y * \text{List} \ y \Rightarrow \text{List} \ x
\]
## Complexity of Model Checking Restricted Fragments

<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 ≥ PSPACE</td>
</tr>
<tr>
<td>MEM</td>
<td>NP</td>
<td>NP</td>
<td>NP PTIME</td>
</tr>
</tbody>
</table>
Implementation

- Implemented both algorithms in OCaml
Implementation

• Implemented both algorithms in OCaml

• Formulated 'typical performance' benchmark suite:
Implementation

- Implemented both algorithms in OCaml
- Formulated 'typical performance' benchmark suite:
  - 6 annotated programs from the Verifast\(^1\) test suite

\(^1\)Bart Jacobs et al., KU Leuven
Implementation

- Implemented both algorithms in OCaml
- Formulated 'typical performance' benchmark suite:
  - 6 annotated programs from the Verifast\textsuperscript{1} test suite
  - harvested over 2150 concrete models at runtime

\textsuperscript{1}Bart Jacobs et al., KU Leuven
Implementation

• Implemented both algorithms in OCaml
• Formulated 'typical performance' benchmark suite:
  • 6 annotated programs from the Verifast\(^1\) test suite
  • harvested over 2150 concrete models at runtime
• Also tested worst-case performance
  • using hand-crafted predicates requiring the generation of all possible submodels

\(^1\)Bart Jacobs et al., KU Leuven
Implementation

- Implemented both algorithms in OCaml
- Formulated 'typical performance' benchmark suite:
  - 6 annotated programs from the Verifast\(^1\) test suite
  - harvested over 2150 concrete models at runtime
- Also tested worst-case performance
  - using hand-crafted predicates requiring the generation of all possible submodels
- Tested top-down algorithm on instances within MEM+CV+DET

\(^1\)Bart Jacobs et al., KU Leuven
• All runs of the top-down algorithm took ~10ms
Experimental Results

• All runs of the top-down algorithm took ~10ms

• Running times for the bottom-up algorithm indicate suitability for unit testing / debugging
Experimental Results

- All runs of the top-down algorithm took ~10ms

- Running times for the bottom-up algorithm indicate suitability for unit testing / debugging
  - for 10 heap cells – between 5 and 60ms
Experimental Results

• All runs of the top-down algorithm took ~10ms

• Running times for the bottom-up algorithm indicate suitability for unit testing / debugging
  
  • for 10 heap cells – between 5 and 60ms
  
  • for 30 heap cells – between 10ms and 10s
Experimental Results

- All runs of the top-down algorithm took ~10ms
- Running times for the bottom-up algorithm indicate suitability for unit testing / debugging
  - for 10 heap cells – between 5 and 60ms
  - for 30 heap cells – between 10ms and 10s
  - some instances with 100 heap cells still checking in ~100ms
Thank you for listening!

Implementation available at: github.com/ngorogoriannis/cyclist
Related Work


Future Work

• Investigate how adding *classical* conjunction affects the decidability / complexity results

• Model checking may facilitate *disproving* of entailments via generation and checking of concrete models