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 20$^{\text{th}}$ January 2016

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

[2]Foundations 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 φ: 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 φ: does $S \models \varphi$?

- Typically, S is a Kripke structure representing a program, and φ 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 φ: does $S \models \varphi$?

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

- More generally, S could be any kind of mathematical structure and φ a formula in a language describing such structures.

- Separation logic (SL) facilitates verification of imperative pointer programs by describing program memory.

Model Checking for Separation Logic

- 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.

Model Checking for Separation Logic

- 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.

Model Checking for Separation Logic

- 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 φ: 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 φ: 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

Overview of our Results

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

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

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

Symbolic Heaps with Inductive Predicates

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

Symbolic Heaps with Inductive Predicates

Terms:	$t ::= x \mid \mathbf{nil}$
Pure Formulas:	$\pi ::= t = t \mid t \neq t$

Symbolic Heaps with Inductive Predicates

Terms:	$t ::= x \mid \mathsf{nil}$
Pure Formulas:	$\pi ::= t = t \mid t \neq t$
Spatial Formulas:	$\Sigma ::= \mathsf{emp} \mid x \mapsto \boldsymbol{t} \mid P\boldsymbol{t} \mid \Sigma * \Sigma$

(P a predicate symbol, $\boldsymbol{t}$ a tuple of terms)

Symbolic Heaps with Inductive Predicates

Terms:	$t ::= x \mid \mathbf{nil}$
Pure Formulas:	$\pi ::= t = t \mid t \neq t$
Spatial Formulas:	$\Sigma ::= \mathbf{emp} \mid x \mapsto \boldsymbol{t} \mid P\,\boldsymbol{t} \mid \Sigma * \Sigma$
	(P a predicate symbol, $\boldsymbol{t}$ a tuple of terms)

- **emp** is the empty heap

Symbolic Heaps with Inductive Predicates

Terms:	$t ::= x \mid \mathbf{nil}$
Pure Formulas:	$\pi ::= t = t \mid t \neq t$
Spatial Formulas:	$\Sigma ::= \mathbf{emp} \mid x \mapsto \boldsymbol{t} \mid P\boldsymbol{t} \mid \Sigma * \Sigma$
	(P a predicate symbol, $\boldsymbol{t}$ a tuple of terms)

- **emp** is the empty heap
- $\mapsto$ ("points to") denotes a *pointer* to a single heap record

Symbolic Heaps with Inductive Predicates

Terms:	$t ::= x \mid \mathbf{nil}$	
Pure Formulas:	$\pi ::= t = t \mid t \neq t$	
Spatial Formulas:	$\Sigma ::= \mathbf{emp} \mid x \mapsto \boldsymbol{t} \mid P\,\boldsymbol{t} \mid \Sigma * \Sigma$	

(P a predicate symbol, $\boldsymbol{t}$ a tuple of terms)

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

Symbolic Heaps with Inductive Predicates

Terms:	$t ::= x \mid \mathbf{nil}$	
Pure Formulas:	$\pi ::= t = t \mid t \neq t$	
Spatial Formulas:	$\Sigma ::= \mathbf{emp} \mid x \mapsto \boldsymbol{t} \mid P\,\boldsymbol{t} \mid \Sigma * \Sigma$	

(P a predicate symbol, $\boldsymbol{t}$ a tuple of terms)

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

Symbolic heaps F given by $\exists \boldsymbol{x}.\Pi : \Sigma$ (Π a set of pure formulas)

Inductive Definitions

- 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:

$$\exists z.\Pi : \Sigma \Rightarrow P\,\mathbf{x}$$

e.g. nil-terminated linked lists with root x:

$$\begin{aligned} x = \mathbf{nil} \;&:\; \mathbf{emp} \;\Rightarrow\; \mathsf{List}\,x \\ \exists y.\, x \neq \mathbf{nil} \;&:\; x \mapsto y \,*\, \mathsf{List}\,y \;\Rightarrow\; \mathsf{List}\,x \end{aligned}$$

Model Checking: Problem Statement

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

Model Checking: Problem Statement

- Recall the *general* form: given a structure S and a formula φ, 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 φ, 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

Model Checking: Problem Statement

- Recall the *general* form: given a structure S and a formula φ, 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 φ, 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 Φ, stack s, heap h and symbolic heap formula F, we must decide whether $(s, h) \models_{\Phi} F$

Model Checking: Subtleties

$P\,\boldsymbol{x}$ $\qquad$ (s, h)

Model Checking: Subtleties

$$\exists z.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \Rightarrow P\,\mathbf{x} \qquad\qquad (s, h)$$

Model Checking: Subtleties

$$\exists z.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \xleftarrow{\mathit{unfold}} P\,x \qquad\qquad (s, h)$$

Model Checking: Subtleties

$$\exists z.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \xleftarrow{\mathit{unfold}} P\,x \qquad\qquad (s, h)$$

- How do we decompose h into $h_1, \ldots, h_n$ to match $\Sigma_1, \ldots, \Sigma_n$?

Model Checking: Subtleties

$$\exists \mathbf{z}.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \xleftarrow{\mathit{unfold}} P\,\mathbf{x} \qquad\qquad (s, h)$$

- 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 $\mathbf{z}$?

Model Checking: Subtleties

$$\exists \boldsymbol{z}.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \xleftarrow{unfold} P\,\boldsymbol{x} \qquad\qquad (s, h)$$

- 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 $\boldsymbol{z}$?
    - We may need values that do not even occur in s or h!

Model Checking: Subtleties

$$\exists \boldsymbol{z}.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \xleftarrow{\textit{unfold}} P\,\boldsymbol{x} \qquad\qquad (s, h)$$

- 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 $\boldsymbol{z}$?
  - We may need values that do not even occur in s or h!
- How to prove termination of such a procedure?

Model Checking: Subtleties

$$\exists \mathbf{z}.\, \Pi : \Sigma_1 * \ldots * \Sigma_n \xleftarrow{unfold} P\,\mathbf{x} \qquad\qquad (s, h)$$

- 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!

Model Checking: Solution

How to decide whether $(s, h) \models_{\Phi} F$

Model Checking: Solution

How to decide whether $(s, h) \models_{\Phi} P\,\boldsymbol{x}$

Model Checking: Solution

How to decide whether $(s, h) \models_{\Phi} P\mathbf{x}$

- We give a bottom-up fixed-point algorithm which:

Model Checking: Solution

How to decide whether $(s, h) \models_\Phi P\,\mathbf{x}$

- We give a bottom-up fixed-point algorithm which:
  - only considers sub-heaps of h

Model Checking: Solution

How to decide whether $(s, h) \models_{\Phi} P\mathbf{x}$

- 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

Model Checking: Solution

How to decide whether $(s, h) \models_{\Phi} P\,\mathbf{x}$

- 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 Φ, then checks if (s, h) is in the set for P

Model Checking: Solution

How to decide whether $(s, h) \models_{\Phi} P\,\mathbf{x}$

- 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 Φ, then checks if (s, h) is in the set for P
- We show that this procedure is complete and has EXPTIME complexity

Restricting Inductive Definitions

$$x = \mathsf{nil} \ : \ \mathsf{emp} \ \Rightarrow \ \mathsf{List}\, x \qquad\qquad \exists y.\, x \neq \mathsf{nil} \ : \ x \mapsto y \ * \ \mathsf{List}\, y \ \Rightarrow \ \mathsf{List}\, x$$

Restricting Inductive Definitions

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

$$x = \mathbf{nil} \ : \ \mathbf{emp} \ \Rightarrow \ \mathbf{List}\, x \qquad\qquad \exists y.\, x \neq \mathbf{nil} \ : \ x \mapsto y \ * \ \mathbf{List}\, y \ \Rightarrow \ \mathbf{List}\, x$$

Restricting Inductive Definitions

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

$$x = \mathbf{nil} \;:\; \mathbf{emp} \;\Rightarrow\; \mathbf{List}\, x \qquad\qquad \exists y.\; x \neq \mathbf{nil} \;:\; x \mapsto y \;*\; \mathbf{List}\, y \;\Rightarrow\; \mathbf{List}\, x$$

Restricting Inductive Definitions

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

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

$$x = \mathbf{nil} \ : \ \mathbf{emp} \ \Rightarrow \ \mathbf{List}\, x \qquad\qquad \exists y.\, x \neq \mathbf{nil} \ : \ x \mapsto y \ * \ \mathbf{List}\, y \ \Rightarrow \ \mathbf{List}\, x$$

Complexity of Model Checking Restricted Fragments

		CV	DET	CV+DET
non-MEM	EXPTIME	EXPTIME	EXPTIME	$\geq$ PSPACE
MEM	NP	NP	NP	PTIME

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[1] test suite
    - harvested over 2150 concrete models at runtime

[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

Experimental Results

- 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/ngorogiannis/cyclist`

Related Work

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

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

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