Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates

James Brotherston[1] Nikos Gorogiannis[2] Max Kanovich[1]
Reuben Rowe[1]

[1]UCL

[2]Middlesex University

Australian National University, Canberra, 9 December 2015

Model checking, in general

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

Model checking, in general

- Model checking is the problem of checking whether a structure S satisfies, or is a model of, some formula A: does $S \models A$?
- In computer science, S is typically a Kripke structure representing a system or program, and A a formula of modal or temporal logic.

Model checking, in general

- Model checking is the problem of checking whether a structure S satisfies, or is a model of, some formula A: does $S \models A$?
- In computer science, S is typically a Kripke structure representing a system or program, and A a formula of modal or temporal logic.
- More generally, S could be any kind of mathematical structure and A a formula describing such structures.

Model checking, in particular

- Our setting: separation logic, used as a formalism for verifying imperative pointer programs.

Model checking, in particular

- Our setting: separation logic, used as a formalism for verifying imperative pointer programs.
- Typically, we do static analysis: given an annotated program, prove that it meets its specification. There are many such automatic analyses!

Model checking, in particular

- Our setting: separation logic, used as a formalism for verifying imperative pointer programs.
- Typically, we do static analysis: given an annotated program, prove that it meets its specification. There are many such automatic analyses!
- When static analysis fails, we might try run-time verification: run the program and check that it does not violate the spec.

Model checking, in particular

- Our setting: separation logic, used as a formalism for verifying imperative pointer programs.
- Typically, we do static analysis: given an annotated program, prove that it meets its specification. There are many such automatic analyses!
- 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 specs A: does $S \models A$?

Model checking, in particular

- Our setting: separation logic, used as a formalism for verifying imperative pointer programs.
- Typically, we do static analysis: given an annotated program, prove that it meets its specification. There are many such automatic analyses!
- 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 specs A: does $S \models A$?
- We focus on the popular symbolic-heap fragment of separation logic, allowing arbitrary inductive predicates.

Symbolic-heap separation logic

- Terms t are either variables $x, y, z \ldots$ or the constant nil.

Symbolic-heap separation logic

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F given by:

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

(where P a predicate symbol, $\mathbf{t}$ a tuple of terms).

Symbolic-heap separation logic

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F given by:

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

  (where P a predicate symbol, $\mathbf{t}$ a tuple of terms).
- $\mapsto$ ("points-to") denotes an individual pointer to a record in the heap.

Symbolic-heap separation logic

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F given by:

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

  (where P a predicate symbol, $\mathbf{t}$ a tuple of terms).
- $\mapsto$ ("points-to") denotes an individual pointer to a record in the heap.
- $*$ ("and separately") demarks domain-disjoint heaps.

Symbolic-heap separation logic

- Terms t are either variables $x, y, z \ldots$ or the constant nil.
- Pure formulas π and spatial formulas F given by:

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

  (where P a predicate symbol, $\mathbf{t}$ a tuple of terms).
- $\mapsto$ ("points-to") denotes an individual pointer to a record in the heap.
- $*$ ("and separately") demarks domain-disjoint heaps.
- Symbolic heaps A given by $\exists \mathbf{x}.\ \Pi : F$, for Π a set of pure formulas.

Inductive definitions in separation logic

- Inductive predicates defined by a set of rules of form:

$$A \Rightarrow P\mathbf{t}$$

(We typically suppress the existential quantifiers in A.)

Inductive definitions in separation logic

- Inductive predicates defined by a set of rules of form:

$$A \Rightarrow P\mathbf{t}$$

(We typically suppress the existential quantifiers in A.)

- E.g., linked list segments with root x and tail element y:

$$\begin{array}{rcl} \mathsf{emp} & \Rightarrow & \mathsf{ls}\,x\,x \\ x \neq \mathsf{nil} : x \mapsto z * \mathsf{ls}\,z\,y & \Rightarrow & \mathsf{ls}\,x\,y \end{array}$$

Inductive definitions in separation logic

- Inductive predicates defined by a set of rules of form:

$$A \Rightarrow P\mathbf{t}$$

(We typically suppress the existential quantifiers in A.)

- E.g., linked list segments with root x and tail element y:

$$\begin{aligned} \mathsf{emp} &\Rightarrow \mathsf{ls}\,x\,x \\ x \neq \mathsf{nil} : x \mapsto z * \mathsf{ls}\,z\,y &\Rightarrow \mathsf{ls}\,x\,y \end{aligned}$$

- E.g., binary trees with root x given by:

$$\begin{aligned} x = \mathsf{nil} : \mathsf{emp} &\Rightarrow \mathsf{bt}\,x \\ x \neq \mathsf{nil} : x \mapsto (y, z) * \mathsf{bt}\,y * \mathsf{bt}\,z &\Rightarrow \mathsf{bt}\,x \end{aligned}$$

Semantics

- Models are stacks $s : \mathsf{Var} \to \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.

Semantics

- Models are stacks $s : \mathsf{Var} \rightarrow \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.
- Forcing relation $s, h \models A$ given by

$$s, h \models_{\Phi} t_1 = (\neq) t_2 \quad \Leftrightarrow \quad s(t_1) = (\neq) s(t_2)$$

Semantics

- Models are stacks $s : \mathsf{Var} \to \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.
- Forcing relation $s, h \models A$ given by

$$\begin{array}{rcl} s, h \models_\Phi t_1 = (\neq) t_2 & \Leftrightarrow & s(t_1) = (\neq) s(t_2) \\ s, h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \end{array}$$

Semantics

- Models are stacks $s : \mathsf{Var} \rightarrow \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.
- Forcing relation $s, h \models A$ given by

$$\begin{array}{rcl} s, h \models_{\Phi} t_1 = (\neq) t_2 & \Leftrightarrow & s(t_1) = (\neq) s(t_2) \\ s, h \models_{\Phi} \mathsf{emp} & \Leftrightarrow & h = e \\ s, h \models_{\Phi} x \mapsto \mathbf{t} & \Leftrightarrow & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{t}) \end{array}$$

Semantics

- Models are stacks $s : \mathsf{Var} \to \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models_\Phi t_1 = (\neq) t_2 & \Leftrightarrow & s(t_1) = (\neq) s(t_2) \\
s, h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models_\Phi x \mapsto \mathbf{t} & \Leftrightarrow & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{t}) \\
s, h \models_\Phi P_i \mathbf{t} & \Leftrightarrow & (s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi
\end{array}
$$

Semantics

- Models are stacks $s : \mathsf{Var} \to \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models_\Phi t_1 = (\neq) t_2 & \Leftrightarrow & s(t_1) = (\neq) s(t_2) \\
s, h \models_\Phi \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models_\Phi x \mapsto \mathbf{t} & \Leftrightarrow & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = s(\mathbf{t}) \\
s, h \models_\Phi P_i \mathbf{t} & \Leftrightarrow & (s(\mathbf{t}), h) \in [\![P_i]\!]^\Phi \\
s, h \models_\Phi F_1 * F_2 & \Leftrightarrow & \exists h_1, h_2.\ h = h_1 \circ h_2 \text{ and } s, h_1 \models_\Phi F_1 \\
 & & \text{and } s, h_2 \models_\Phi F_2
\end{array}
$$

Semantics

- Models are stacks $s : \mathsf{Var} \to \mathsf{Val}$ paired with heaps $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$. $\circ$ is union of domain-disjoint heaps; e is the empty heap; nil is a non-allocable value.

- Forcing relation $s, h \models A$ given by

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

Semantics of inductive predicates

Given inductive rule set Φ, the semantics $[\![P]\!]^{\Phi}$ of inductive predicate P is the least fixed point of a monotone operator constructed from Φ.

Semantics of inductive predicates

Given inductive rule set Φ, the semantics $[\![P]\!]^{\Phi}$ of inductive predicate P is the least fixed point of a monotone operator constructed from Φ.

E.g, recall linked list segments `ls`:

$$\begin{array}{rcl} \mathsf{emp} & \Rightarrow & \mathsf{ls}\,x\,x \\ x \neq \mathsf{nil} : x \mapsto z * \mathsf{ls}\,z\,y & \Rightarrow & \mathsf{ls}\,x\,y \end{array}$$

Semantics of inductive predicates

Given inductive rule set Φ, the semantics $[\![P]\!]^\Phi$ of inductive predicate P is the least fixed point of a monotone operator constructed from Φ.

E.g, recall linked list segments `ls`:

$$\begin{array}{rcl} \mathsf{emp} & \Rightarrow & \mathsf{ls}\,x\,x \\ x \neq \mathsf{nil} : x \mapsto z * \mathsf{ls}\,z\,y & \Rightarrow & \mathsf{ls}\,x\,y \end{array}$$

The corresponding operator is:

$$\begin{array}{rcl} \varphi(X) & = & \{(h, (s(x), s(y)) \mid \quad s, h \models x = y \text{ and } s, h \models \mathsf{emp}, \text{ or} \\ & & \qquad\qquad\qquad\quad\; s, h \models x \mapsto z * X zy\} \end{array}$$

where $X\,zy$ is interpreted as $(z, y) \in X$.

Problem statement

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

Problem statement

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

First, we can simplify the problem:

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

Problem statement

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

First, we can simplify the problem:

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

Proposition

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

Subtle problem 1

Naive idea: apply inductive rules backwards to $P\mathbf{x}$ until we reach the empty heap.

Subtle problem 1

Naive idea: apply inductive rules backwards to $P\mathbf{x}$ until we reach the empty heap.

But, suppose $((a,b),h) \in [\![P]\!]^\Phi$, and is generated by the rule

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

So, for some $c \in \mathsf{Val}$, we have both $((a,c),h_1) \in [\![P]\!]^\Phi$ and $((c,b),h_2) \in [\![P]\!]^\Phi$, where $h = h_1 \circ h_2$.

Subtle problem 1

Naive idea: apply inductive rules backwards to $P\mathbf{x}$ until we reach the empty heap.

But, suppose $((a,b),h) \in [\![P]\!]^\Phi$, and is generated by the rule

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

So, for some $c \in \mathsf{Val}$, we have both $((a,c),h_1) \in [\![P]\!]^\Phi$ and $((c,b),h_2) \in [\![P]\!]^\Phi$, where $h = h_1 \circ h_2$.

But we do *not* know that h_1, h_2 are smaller than h.

Subtle problem 1

Naive idea: apply inductive rules backwards to $P\mathbf{x}$ until we reach the empty heap.

But, suppose $((a,b),h) \in \llbracket P \rrbracket^{\Phi}$, and is generated by the rule

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

So, for some $c \in \mathsf{Val}$, we have both $((a,c),h_1) \in \llbracket P \rrbracket^{\Phi}$ and $((c,b),h_2) \in \llbracket P \rrbracket^{\Phi}$, where $h = h_1 \circ h_2$.

But we do *not* know that h_1, h_2 are smaller than h.

Moral: compute "sub-models" of $(\mathbf{a}, h)$ bottom-up until we reach a fixed point.

Subtle problem 2

Suppose $(a, e) \in [\![P]\!]^\Phi$ is generated by the rule

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

So, for some $b \in \mathsf{Val}$, we have $((a, b), e) \in [\![Q]\!]^\Phi$, where $b \neq a$ and b (trivially) does not appear in the empty heap e.

Subtle problem 2

Suppose $(a, e) \in [\![P]\!]^{\Phi}$ is generated by the rule

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

So, for some $b \in \mathsf{Val}$, we have $((a, b), e) \in [\![Q]\!]^{\Phi}$, where $b \neq a$ and b (trivially) does not appear in the empty heap e.

Thus we must allow our sub-models to mention fresh, or "spare", values not mentioned in $\mathbf{a}$ or h.

Subtle problem 2

Suppose $(a, e) \in [\![P]\!]^{\Phi}$ is generated by the rule

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

So, for some $b \in \mathsf{Val}$, we have $((a, b), e) \in [\![Q]\!]^{\Phi}$, where $b \neq a$ and b (trivially) does not appear in the empty heap e.

Thus we must allow our sub-models to mention fresh, or "spare", values not mentioned in $\mathbf{a}$ or h.

Fortunately, for any given set of definitions Φ, we can get away with using only finitely many of these spare values.

Our model checking constructions

Given Φ, $\mathbf{a}$ and h, define

$$\text{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \text{all values in } h.$$

Our model checking constructions

Given Φ, $\mathbf{a}$ and h, define

$$\mathrm{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \text{all values in } h.$$

Now let β be the maximum number of variables in any rule in Φ, and define $\mathrm{Spare}_{\Phi}(\mathbf{a}, h)$ to be a set of β fresh values.

Our model checking constructions

Given Φ, $\mathbf{a}$ and h, define

$$\mathrm{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \text{all values in } h.$$

Now let β be the maximum number of variables in any rule in Φ, and define $\mathrm{Spare}_{\Phi}(\mathbf{a}, h)$ to be a set of β fresh values.

Then, given Φ, values $\mathbf{a}$ and heap h we define a monotone operator, similar to the one that constructs the semantics of inductive predicates except that

Our model checking constructions

Given Φ, $\mathbf{a}$ and h, define

$$\mathrm{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \text{all values in } h.$$

Now let β be the maximum number of variables in any rule in Φ, and define $\mathrm{Spare}_{\Phi}(\mathbf{a}, h)$ to be a set of β fresh values.

Then, given Φ, values $\mathbf{a}$ and heap h we define a monotone operator, similar to the one that constructs the semantics of inductive predicates except that

- we only consider heaps $h' \subseteq h$,

Our model checking constructions

Given Φ, $\mathbf{a}$ and h, define

$$\text{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \text{all values in } h.$$

Now let β be the maximum number of variables in any rule in Φ, and define $\text{Spare}_\Phi(\mathbf{a}, h)$ to be a set of β fresh values.

Then, given Φ, values $\mathbf{a}$ and heap h we define a monotone operator, similar to the one that constructs the semantics of inductive predicates except that

- we only consider heaps $h' \subseteq h$, and
- all values instantiating variables must be taken from $\text{Good}(\mathbf{a}, h) \cup \text{Spare}_\Phi(\mathbf{a}, h)$.

Our model checking constructions

Given Φ, $\mathbf{a}$ and h, define

$$\mathrm{Good}(\mathbf{a}, h) = \mathbf{a} \cup \{nil\} \cup \text{all values in } h.$$

Now let β be the maximum number of variables in any rule in Φ, and define $\mathrm{Spare}_\Phi(\mathbf{a}, h)$ to be a set of β fresh values.

Then, given Φ, values $\mathbf{a}$ and heap h we define a monotone operator, similar to the one that constructs the semantics of inductive predicates except that

- we only consider heaps $h' \subseteq h$, and
- all values instantiating variables must be taken from $\mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_\Phi(\mathbf{a}, h)$.

Again, we take the least fixed point of the operator, and write $MC_i^\Phi(\mathbf{a}, h)$ for the component corresponding to ith predicate.

Correctness

Lemma

For each predicate P_i,

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

Correctness

Lemma

For each predicate P_i,

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

Soundness ($\Leftarrow$) is easy — $MC_i^\Phi(\mathbf{a}, h)$ only constructs models of P_i by construction.

Correctness

Lemma
For each predicate P_i,

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

Soundness ($\Leftarrow$) is easy — $MC_i^\Phi(\mathbf{a}, h)$ only constructs models of P_i by construction.

However, completeness ($\Rightarrow$) is hard: we have to show that $(\mathbf{a}, h)$ must eventually turn up in $MC_i^\Phi(\mathbf{a}, h)$, even if its derivation involves values outside $\mathrm{Good}(\mathbf{a}, h) \cup \mathrm{Spare}_\Phi(\mathbf{a}, h)$. Argument involves considering certain value substitutions and recycling values at each iteration of the fixed point construction.

Decidability

Theorem

The model checking problem MC *is decidable.*

Decidability

Theorem

The model checking problem MC *is decidable.*

Proof.

It suffices to show that RMC is decidable: does $(\mathbf{a}, h) \in [\![P_i]\!]^{\Phi}$?

Decidability

Theorem
The model checking problem MC *is decidable.*

Proof.
It suffices to show that RMC is decidable: does $(\mathbf{a}, h) \in [\![P_i]\!]^{\Phi}$?

By our correctness lemma, this is equivalent to deciding whether $(\mathbf{a}, h) \in MC_i^{\Phi}(\mathbf{a}, h)$.

Decidability

Theorem
The model checking problem MC *is decidable.*

Proof.
It suffices to show that RMC is decidable: does $(\mathbf{a}, h) \in [\![P_i]\!]^{\Phi}$?

By our correctness lemma, this is equivalent to deciding whether $(\mathbf{a}, h) \in MC_i^{\Phi}(\mathbf{a}, h)$.

But clearly $MC_i^{\Phi}(\mathbf{a}, h)$ is a finite and computable set (because we restrict to subheaps of h and a finite set of values), so this is a decidable problem. $\square$

Complexity of model checking

Theorem

MC *is* EXPTIME-*complete.*

Complexity of model checking

Theorem
MC *is* EXPTIME-*complete.*

Proof. Computing $MC_i^{\Phi}(\mathbf{a}, h)$ decides the problem and can be seen to run in exponential time in the size of $(\mathbf{a}, h, \Phi)$.

Complexity of model checking

Theorem
MC *is* EXPTIME-*complete.*

Proof. Computing $MC_i^{\Phi}(\mathbf{a}, h)$ decides the problem and can be seen to run in exponential time in the size of $(\mathbf{a}, h, \Phi)$.
EXPTIME-hardness is by reduction from the satisfiability problem for our logic, which is EXPTIME-hard [Brotherston et al., CSL-LICS'14].

Complexity of model checking

Theorem

MC *is* EXPTIME*-complete.*

Proof. Computing $MC_i^{\Phi}(\mathbf{a}, h)$ decides the problem and can be seen to run in exponential time in the size of $(\mathbf{a}, h, \Phi)$.
EXPTIME-hardness is by reduction from the satisfiability problem for our logic, which is EXPTIME-hard [Brotherston et al., CSL-LICS'14].

Proposition

When Φ *and* $\mathbf{a}$ *are fixed,* MC *is still* NP*-hard in the size of* h*.*

Complexity of model checking

Theorem

MC *is* EXPTIME-*complete.*

Proof. Computing $MC_i^{\Phi}(\mathbf{a}, h)$ decides the problem and can be seen to run in exponential time in the size of $(\mathbf{a}, h, \Phi)$. EXPTIME-hardness is by reduction from the satisfiability problem for our logic, which is EXPTIME-hard [Brotherston et al., CSL-LICS'14].

Proposition

When Φ *and* $\mathbf{a}$ *are fixed,* MC *is still* NP*-hard in the size of* h*.*

Proof. By reduction from the triangle partition problem: given a graph $G = (V, E)$ with $|V| = 3q$ for some $q > 0$, decide whether there is a partition of G into triangles.

MEM: *Restriction to memory-consuming rules*

An inductive rule set is memory-consuming (a.k.a. “in MEM”)

MEM: *Restriction to memory-consuming rules*

An inductive rule set is memory-consuming (a.k.a. "in MEM") if every rule in it is of the form

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

i.e., one or more pointers are "consumed" when recursing.

MEM: *Restriction to memory-consuming rules*

An inductive rule set is memory-consuming (a.k.a. "in MEM") if every rule in it is of the form

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

i.e., one or more pointers are "consumed" when recursing.

In practice, almost all predicate definitions in the literature fall into MEM.

Model checking in the MEM *fragment*

Theorem

MC $\in$ NP *when all predicates are restricted to* MEM.

Model checking in the MEM *fragment*

Theorem

$\mathsf{MC} \in \mathsf{NP}$ *when all predicates are restricted to* MEM.

Proof. Given predicate P_i, values $\mathbf{a}$ and heap h, we can search backwards by applying inductive rules to $(\mathbf{a}, h) \in [\![P_i]\!]$, noting that we can confine the search space of values using our previous observations. This search must terminate because at least one heap cell is consumed with each recursion.

Model checking in the MEM *fragment*

Theorem

MC $\in$ NP *when all predicates are restricted to* MEM.

Proof. Given predicate P_i, values $\mathbf{a}$ and heap h, we can search backwards by applying inductive rules to $(\mathbf{a}, h) \in [\![P_i]\!]$, noting that we can confine the search space of values using our previous observations. This search must terminate because at least one heap cell is consumed with each recursion.

Theorem

MC *is in fact* NP*-hard for* MEM *(thus* NP*-complete), even when some further restrictions are added.*

Restriction: constructively valued definitions (CV)

Informally, a rule set is constructively valued ("in CV") if values of existentially quantified variables are determined by a given heap and values for variables in the head.

Restriction: constructively valued definitions (CV)

Informally, a rule set is constructively valued ("in CV") if values of existentially quantified variables are determined by a given heap and values for variables in the head.

E.g., consider two list definitions

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

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

The existential z is constructively valued in ls, but not in rls.

Restriction: deterministic definitions (DET)

A predicate P_i is said to be *deterministic* (in an inductive rule set Φ) if for any two of its inductive rules and any stack, the stack can satisfy the pure part of at most one of the rules.

*Restriction: deterministic definitions (*DET*)*

A predicate P_i is said to be *deterministic* (in an inductive rule set Φ) if for any two of its inductive rules and any stack, the stack can satisfy the pure part of at most one of the rules.

Again, take the list definitions:

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

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

Here, rls is deterministic, but ls is not.

Results on CV + DET *fragments*

Theorem

MC *is* PTIME*-solvable when all predicates are in* MEM + CV + DET.

Results on $\mathsf{CV} + \mathsf{DET}$ *fragments*

Theorem

MC *is* PTIME*-solvable when all predicates are in* $\mathsf{MEM} + \mathsf{CV} + \mathsf{DET}$*.*

Proof.

Like in the MEM case, we can search backwards for a derivation of $(\mathbf{a}, h) \in [\![P_i]\!]^\Phi$ using inductive rules. MEM ensures termination. DET ensures at most one inductive rule can apply, and CV ensures it can be instantiated in only one way.

Results on $\mathsf{CV} + \mathsf{DET}$ *fragments*

Theorem

MC *is* PTIME-*solvable when all predicates are in* MEM + CV + DET.

Proof.

Like in the MEM case, we can search backwards for a derivation of $(\mathbf{a}, h) \in [\![P_i]\!]^{\Phi}$ using inductive rules. MEM ensures termination. DET ensures at most one inductive rule can apply, and CV ensures it can be instantiated in only one way. □

Theorem

If we remove any of the restrictions MEM, CV, DET, *then the complexity of* MC *becomes* PSPACE-*hard or worse!*

Summary of problem complexities

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

Implementation

- We have implemented the general EXPTIME algorithm and the PTIME algorithm for $\mathsf{MEM} + \mathsf{CV} + \mathsf{DET}$ in OcaML.

Implementation

- We have implemented the general EXPTIME algorithm and the PTIME algorithm for MEM + CV + DET in OcaML.
- Tested on a range of annotated test programs, falling into various fragments, taken from the Verifast tool (Jacobs et al., Leuvens).

Implementation

- We have implemented the general EXPTIME algorithm and the PTIME algorithm for MEM + CV + DET in OcaML.
- Tested on a range of annotated test programs, falling into various fragments, taken from the Verifast tool (Jacobs et al., Leuvens).
- Average-case performance is in line with predicted complexity bounds.

Implementation

- We have implemented the general EXPTIME algorithm and the PTIME algorithm for MEM + CV + DET in OcaML.
- Tested on a range of annotated test programs, falling into various fragments, taken from the Verifast tool (Jacobs et al., Leuvens).
- Average-case performance is in line with predicted complexity bounds.
- Thus, run-time verification is broadly practical for predicates in MEM + CV + DET; more complicated predicates can play a role in unit testing.

Conclusions

- Main contribution: for symbolic-heap separation logic with arbitrary inductive predicates, the model checking problem is decidable and indeed EXPTIME-complete.

Conclusions

- Main contribution: for symbolic-heap separation logic with arbitrary inductive predicates, the model checking problem is decidable and indeed EXPTIME-complete.
- However, in practice most predicates are memory-consuming, i.e. in MEM, in which case the problem becomes NP-complete.

Conclusions

- Main contribution: for symbolic-heap separation logic with arbitrary inductive predicates, the model checking problem is decidable and indeed EXPTIME-complete.
- However, in practice most predicates are memory-consuming, i.e. in MEM, in which case the problem becomes NP-complete.
- If we additionally insist on constructively valued (CV) and deterministic (DET) definitions (some are, some aren't), then the problem becomes PTIME-solvable.

Future work

- Investigate the complexity when we add classical conjunction $\wedge$ to the logic? (Satisfiability becomes undecidable.)

Future work

- Investigate the complexity when we add classical conjunction $\wedge$ to the logic? (Satisfiability becomes undecidable.)
- Investigate complexity of satisfiability for combinations of MEM/CV/DET.

Future work

- Investigate the complexity when we add classical conjunction $\wedge$ to the logic? (Satisfiability becomes undecidable.)
- Investigate complexity of satisfiability for combinations of MEM/CV/DET.
- Implementing the NP algorithm for the MEM fragment can be expected to yield better implementation performance (on MEM).

Future work

- Investigate the complexity when we add classical conjunction $\wedge$ to the logic? (Satisfiability becomes undecidable.)
- Investigate complexity of satisfiability for combinations of $\mathsf{MEM}/\mathsf{CV}/\mathsf{DET}$.
- Implementing the NP algorithm for the MEM fragment can be expected to yield better implementation performance (on MEM).
- Disprove entailments using model checking?

Thanks for listening!

Try our techniques within the Cyclist distribution:

`github.com/ngorogiannis/cyclist`

Also available as an official POPL'16 Artefact.