Biabduction (and Related Problems) in Array Separation Logic

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

[1]UCL

[2]Middlesex University

University of Vienna, 14 Mar 2017

Compositional proofs in separation logic (1)

- Separation logic is based on Hoare triples $\{A\}\ C\ \{B\}$, where C is a program and A, B are formulas.

Compositional proofs in separation logic (1)

- Separation logic is based on Hoare triples $\{A\}\ C\ \{B\}$, where C is a program and A, B are formulas.
- Its compositional nature, the key to scalable analysis, is supported by two main pillars.

Compositional proofs in separation logic (1)

- Separation logic is based on Hoare triples $\{A\}\ C\ \{B\}$, where C is a program and A, B are formulas.
- Its compositional nature, the key to scalable analysis, is supported by two main pillars.
- The first pillar is the soundness of the following frame rule:

$$\frac{\{A\}\ C\ \{B\}}{\{A * F\}\ C\ \{B * F\}}\ \text{(Frame)}$$

where the separating conjunction $*$ is read, intuitively, as *"and separately in memory"*.

Compositional proofs in separation logic (2)

- The second pillar is given by solving the biabduction problem:

Compositional proofs in separation logic (2)

- The second pillar is given by solving the biabduction problem: given formulas A and B, find formulas X, Y with

$$A * X \models B * Y \text{ , and } A * X \text{ is satisfiable.}$$

Compositional proofs in separation logic (2)

- The second pillar is given by solving the biabduction problem: given formulas A and B, find formulas X, Y with

$$A * X \models B * Y \text{ , and } A * X \text{ is satisfiable.}$$

- Then, if we have $\{A'\}\, C_1\, \{A\}$ and $\{B\}\, C_2\, \{B'\}$, we can infer a spec for $C_1; C_2$:

Compositional proofs in separation logic (2)

- The second pillar is given by solving the biabduction problem: given formulas A and B, find formulas X, Y with

$$A * X \models B * Y \ , \text{ and } A * X \text{ is satisfiable.}$$

- Then, if we have $\{A'\}\, C_1\, \{A\}$ and $\{B\}\, C_2\, \{B'\}$, we can infer a spec for $C_1; C_2$:

$$\dfrac{\dfrac{\dfrac{\{A'\}\, C_1\, \{A\}}{\{A' * X\}\, C_1\, \{A * X\}}\ (\text{Frame})}{\{A' * X\}\, C_1\, \{B * Y\}}\ (\models) \qquad \dfrac{\{B\}\, C_2\, \{B'\}}{\{B * Y\}\, C_2\, \{B' * Y\}}\ (\text{Frame})}{\{A' * X\}\, C_1; C_2\, \{B' * Y\}}\ (;)$$

Symbolic-heap separation logic

- Terms t, pure formulas Π and spatial formulas F given by:

$$
\begin{array}{rcl}
t & ::= & x \in \mathsf{Var} \mid \mathsf{nil} \\
\Pi & ::= & t = t \mid t \neq t \mid \Pi \wedge \Pi \\
F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{ls}(t\,, t) \mid F * F
\end{array}
$$

Symbolic-heap separation logic

- Terms t, pure formulas Π and spatial formulas F given by:

$$
\begin{array}{rcl}
t & ::= & x \in \mathsf{Var} \mid \mathsf{nil} \\
\Pi & ::= & t = t \mid t \neq t \mid \Pi \wedge \Pi \\
F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{ls}(t\,, t) \mid F * F
\end{array}
$$

- $t_1 \mapsto t_2$ ("points-to") denotes a pointer in the heap.

Symbolic-heap separation logic

- Terms t, pure formulas Π and spatial formulas F given by:

$$\begin{array}{rcl} t & ::= & x \in \mathsf{Var} \mid \mathsf{nil} \\ \Pi & ::= & t = t \mid t \neq t \mid \Pi \wedge \Pi \\ F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{ls}(t\,, t) \mid F * F \end{array}$$

- $t_1 \mapsto t_2$ ("points-to") denotes a pointer in the heap.
- $\mathsf{ls}(t_1\,, t_2)$ denotes a linked list segment in the heap.

Symbolic-heap separation logic

- Terms t, pure formulas Π and spatial formulas F given by:

$$
\begin{array}{rcl}
t & ::= & x \in \mathsf{Var} \mid \mathsf{nil} \\
\Pi & ::= & t = t \mid t \neq t \mid \Pi \wedge \Pi \\
F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{ls}(t\,,t) \mid F * F
\end{array}
$$

- $t_1 \mapsto t_2$ ("points-to") denotes a pointer in the heap.
- $\mathsf{ls}(t_1\,,t_2)$ denotes a linked list segment in the heap.
- $*$ ("and separately") demarks domain-disjoint heaps.

Symbolic-heap separation logic

- Terms t, pure formulas Π and spatial formulas F given by:

$$\begin{array}{rcl} t & ::= & x \in \mathsf{Var} \mid \mathsf{nil} \\ \Pi & ::= & t = t \mid t \neq t \mid \Pi \wedge \Pi \\ F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{ls}(t\,,t) \mid F * F \end{array}$$

- $t_1 \mapsto t_2$ ("points-to") denotes a pointer in the heap.
- $\mathsf{ls}(t_1\,,t_2)$ denotes a linked list segment in the heap.
- $*$ ("and separately") demarks domain-disjoint heaps.
- Symbolic heaps given by $\exists \mathbf{x}.\ \Pi : F$.

Array separation logic, ASL

- Here we focus on a different data structure, namely arrays.

Array separation logic, ASL

- Here we focus on a different data structure, namely arrays.
- Terms t, pure formulas Π and spatial formulas F given by:

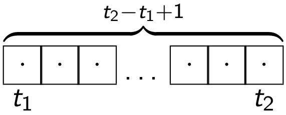
$$
\begin{array}{rcl}
t & ::= & x \in \mathsf{Var} \mid n \in \mathbb{N} \mid t + t \\
\Pi & ::= & t = t \mid t \neq t \mid t \leq t \mid t < t \mid \Pi \wedge \Pi \\
F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{array}(t, t) \mid F * F
\end{array}
$$

Array separation logic, ASL

- Here we focus on a different data structure, namely arrays.
- Terms t, pure formulas Π and spatial formulas F given by:

$$\begin{array}{rcl} t & ::= & x \in \mathsf{Var} \mid n \in \mathbb{N} \mid t + t \\ \Pi & ::= & t = t \mid t \neq t \mid t \leq t \mid t < t \mid \Pi \wedge \Pi \\ F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{array}(t, t) \mid F * F \end{array}$$

- $\mathsf{array}(t_1, t_2)$ denotes an array from t_1 to t_2 (inclusive):

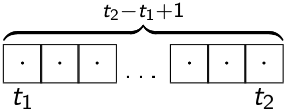


Array separation logic, ASL

- Here we focus on a different data structure, namely arrays.
- Terms t, pure formulas Π and spatial formulas F given by:

$$
\begin{array}{rcl}
t & ::= & x \in \mathsf{Var} \mid n \in \mathbb{N} \mid t + t \\
\Pi & ::= & t = t \mid t \neq t \mid t \leq t \mid t < t \mid \Pi \wedge \Pi \\
F & ::= & \mathsf{emp} \mid t \mapsto t \mid \mathsf{array}(t, t) \mid F * F
\end{array}
$$

- $\mathsf{array}(t_1, t_2)$ denotes an array from t_1 to t_2 (inclusive):



- We also allow linear arithmetic in the pure part.

Semantics of ASL

- Stacks are $s : \mathsf{Var} \to \mathsf{Val}$; heaps are $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$; $\circ$ is union of domain-disjoint heaps; e is the empty heap.

Semantics of ASL

- Stacks are $s : \mathsf{Var} \to \mathsf{Val}$; heaps are $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$; $\circ$ is union of domain-disjoint heaps; e is the empty heap.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models t_1 \sim t_2 & \Leftrightarrow & s(t_1) \sim s(t_2) \quad (\sim \in \{=, \neq, <, \leq\}) \\
s, h \models \Pi_1 \wedge \Pi_2 & \Leftrightarrow & s, h \models \Pi_1 \text{ and } s, h \models \Pi_2
\end{array}
$$

Semantics of ASL

- Stacks are $s : \mathsf{Var} \to \mathsf{Val}$; heaps are $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$; $\circ$ is union of domain-disjoint heaps; e is the empty heap.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models t_1 \sim t_2 & \Leftrightarrow & s(t_1) \sim s(t_2) \quad (\sim \in \{=, \neq, <, \leq\}) \\
s, h \models \Pi_1 \wedge \Pi_2 & \Leftrightarrow & s, h \models \Pi_1 \text{ and } s, h \models \Pi_2 \\
s, h \models \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models t_1 \mapsto t_2 & \Leftrightarrow & \mathrm{dom}(h) = \{s(t_1)\} \text{ and } h(s(t_1)) = s(t_2)
\end{array}
$$

Semantics of ASL

- Stacks are $s : \mathsf{Var} \rightarrow \mathsf{Val}$; heaps are $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$; $\circ$ is union of domain-disjoint heaps; e is the empty heap.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models t_1 \sim t_2 & \Leftrightarrow & s(t_1) \sim s(t_2) \quad (\sim \in \{=, \neq, <, \leq\}) \\
s, h \models \Pi_1 \wedge \Pi_2 & \Leftrightarrow & s, h \models \Pi_1 \text{ and } s, h \models \Pi_2 \\
s, h \models \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models t_1 \mapsto t_2 & \Leftrightarrow & \mathrm{dom}(h) = \{s(t_1)\} \text{ and } h(s(t_1)) = s(t_2) \\
s, h \models \mathsf{array}(t_1, t_2) & \Leftrightarrow & s(t_1) \leq s(t_2) \text{ and } \mathrm{dom}(h) = \{s(t_1), \ldots, s(t_2)\}
\end{array}
$$

Semantics of ASL

- Stacks are $s : \mathsf{Var} \to \mathsf{Val}$; heaps are $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$; $\circ$ is union of domain-disjoint heaps; e is the empty heap.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models t_1 \sim t_2 & \Leftrightarrow & s(t_1) \sim s(t_2) \quad (\sim \in \{=, \neq, <, \leq\}) \\
s, h \models \Pi_1 \wedge \Pi_2 & \Leftrightarrow & s, h \models \Pi_1 \text{ and } s, h \models \Pi_2 \\
s, h \models \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models t_1 \mapsto t_2 & \Leftrightarrow & \mathrm{dom}(h) = \{s(t_1)\} \text{ and } h(s(t_1)) = s(t_2) \\
s, h \models \mathsf{array}(t_1, t_2) & \Leftrightarrow & s(t_1) \leq s(t_2) \text{ and } \mathrm{dom}(h) = \{s(t_1), \ldots, s(t_2)\} \\
s, h \models F_1 * F_2 & \Leftrightarrow & h = h_1 \circ h_2 \text{ and } s, h_1 \models F_1 \text{ and } s, h_2 \models F_2
\end{array}
$$

Semantics of ASL

- Stacks are $s : \mathsf{Var} \to \mathsf{Val}$; heaps are $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} \mathsf{Val}$; $\circ$ is union of domain-disjoint heaps; e is the empty heap.
- Forcing relation $s, h \models A$ given by

$$
\begin{array}{rcl}
s, h \models t_1 \sim t_2 & \Leftrightarrow & s(t_1) \sim s(t_2) \quad (\sim \in \{=, \neq, <, \leq\}) \\
s, h \models \Pi_1 \wedge \Pi_2 & \Leftrightarrow & s, h \models \Pi_1 \text{ and } s, h \models \Pi_2 \\
s, h \models \mathsf{emp} & \Leftrightarrow & h = e \\
s, h \models t_1 \mapsto t_2 & \Leftrightarrow & \mathrm{dom}(h) = \{s(t_1)\} \text{ and } h(s(t_1)) = s(t_2) \\
s, h \models \mathsf{array}(t_1, t_2) & \Leftrightarrow & s(t_1) \leq s(t_2) \text{ and } \mathrm{dom}(h) = \{s(t_1), \ldots, s(t_2)\} \\
s, h \models F_1 * F_2 & \Leftrightarrow & h = h_1 \circ h_2 \text{ and } s, h_1 \models F_1 \text{ and } s, h_2 \models F_2 \\
s, h \models \exists \mathbf{z}.\ \Pi : F & \Leftrightarrow & \exists \mathbf{v}.\ s[\mathbf{z} \mapsto \mathbf{v}], h \models \Pi \text{ and } s[\mathbf{z} \mapsto \mathbf{v}], h \models F
\end{array}
$$

Motivating example

Suppose we have procedure `foo` with spec

$$\{\mathsf{array}(c, d)\}\,\texttt{foo(c, d)}\,\{Q\}$$

Motivating example

Suppose we have procedure `foo` with spec

$$\{\mathsf{array}(c, d)\}\,\texttt{foo(c,d)}\,\{Q\}$$

Now, consider code $C;\texttt{foo(c,d)};\ldots$, with spec for C

$$\{\mathsf{emp}\}\;C\;\{\mathsf{array}(a, b)\}$$

Motivating example

Suppose we have procedure `foo` with spec

$$\{\mathsf{array}(c, d)\}\, \mathtt{foo(c,d)}\, \{Q\}$$

Now, consider code $C; \mathtt{foo(c,d)}; \ldots$, with spec for C

$$\{\mathsf{emp}\}\, C\, \{\mathsf{array}(a, b)\}$$

By solving the biabduction problem

$$\mathsf{array}(a, b) * X \models \mathsf{array}(c, d) * Y$$

we get a valid spec $\{X\}\, C; \mathtt{foo(c,d)}\, \{Q * Y\}$.

Motivating example

Suppose we have procedure `foo` with spec

$$\{\mathsf{array}(c,d)\}\,\texttt{foo}(\texttt{c},\texttt{d})\,\{Q\}$$

Now, consider code $C;\texttt{foo}(\texttt{c},\texttt{d});\ldots$, with spec for C

$$\{\mathsf{emp}\}\,C\,\{\mathsf{array}(a,b)\}$$

By solving the biabduction problem

$$\mathsf{array}(a,b) * X \models \mathsf{array}(c,d) * Y$$

we get a valid spec $\{X\}\,C;\texttt{foo}(\texttt{c},\texttt{d})\,\{Q * Y\}$.
Spatially minimal, and incomparable, solutions include:

$$\begin{array}{l}
X := a = c \wedge b = d : \mathsf{emp} \quad \text{and} \quad Y := \mathsf{emp} \\
X := d < a : \mathsf{array}(c,d) \quad \text{and} \quad Y := \mathsf{array}(a,b) \\
X := a < c \wedge b < d : \mathsf{emp} \quad \text{and} \quad Y := \mathsf{array}(a,c-1) * \mathsf{array}(b+1,d) \\
X := a < c < b < d : \mathsf{array}(b+1,d) \quad \text{and} \quad Y := \mathsf{array}(a,c-1)
\end{array}$$

Satisfiability, upper bound

Satisfiability problem for ASL**.** *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

Satisfiability, upper bound

Satisfiability problem for ASL**.** *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

- Write A as $\Pi : \bigstar_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigstar_{i=1}^{k} t_i \mapsto u_i$.

Satisfiability, upper bound

Satisfiability problem for ASL**.** *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

- Write A as $\Pi : \bigstar_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigstar_{i=1}^{k} t_i \mapsto u_i$.
- Observe A is satisfiable iff there is stack s such that
  - $s \models \Pi$, and

Satisfiability, upper bound

Satisfiability problem for ASL. *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

- Write A as $\Pi : \bigast_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigast_{i=1}^{k} t_i \mapsto u_i$.
- Observe A is satisfiable iff there is stack s such that
    - $s \models \Pi$, and
    - each array is well-defined $(s(a_i) \leq s(b_i))$, and

Satisfiability, upper bound

Satisfiability problem for ASL**.** *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

- Write A as $\Pi : \bigast_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigast_{i=1}^{k} t_i \mapsto u_i$.
- Observe A is satisfiable iff there is stack s such that
  - $s \models \Pi$, and
  - each array is well-defined $(s(a_i) \leq s(b_i))$, and
  - all pointers and arrays are mutually non-overlapping $((s(b_1) < s(a_2) \vee s(a_1) > s(b_2)) \wedge \ldots)$.

Satisfiability, upper bound

Satisfiability problem for ASL. *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

- Write A as $\Pi : \bigstar_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigstar_{i=1}^{k} t_i \mapsto u_i$.
- Observe A is satisfiable iff there is stack s such that
  - $s \models \Pi$, and
  - each array is well-defined ($s(a_i) \leq s(b_i)$), and
  - all pointers and arrays are mutually non-overlapping ($(s(b_1) < s(a_2) \vee s(a_1) > s(b_2)) \wedge \ldots$).
- We can code this up as a formula $\gamma(A)$ in Σ_1^0 Presburger arithmetic.

Satisfiability, upper bound

Satisfiability problem for ASL. *Given symbolic heap A, decide if there is a stack s and heap h with $s, h \models A$.*

- Write A as $\Pi : \bigstar_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigstar_{i=1}^{k} t_i \mapsto u_i$.
- Observe A is satisfiable iff there is stack s such that
  - $s \models \Pi$, and
  - each array is well-defined ($s(a_i) \leq s(b_i)$), and
  - all pointers and arrays are mutually non-overlapping ($(s(b_1) < s(a_2) \vee s(a_1) > s(b_2)) \wedge \ldots$).
- We can code this up as a formula $\gamma(A)$ in Σ^0_1 Presburger arithmetic.
- Thus the problem is in NP.

Satisfiability, lower bound

- NP-hardness follows by reduction from

Satisfiability, lower bound

- NP-hardness follows by reduction from

  3-partition problem. *Given $B \in \mathbb{N}$ and a sequence of natural numbers $\mathcal{S} = (k_1, k_2, \ldots, k_{3m})$ with $\sum_{j=1}^{3m} k_j = mB$ and $B/4 < k_j < B/2$ for all $j \in [1, 3m]$, decide whether there is a complete 3-partition of $\mathcal{S}$ s.t. each partition sums to B.*

Satisfiability, lower bound

- NP-hardness follows by reduction from

  3-partition problem. *Given $B \in \mathbb{N}$ and a sequence of natural numbers $\mathcal{S} = (k_1, k_2, \ldots, k_{3m})$ with $\sum_{j=1}^{3m} k_j = mB$ and $B/4 < k_j < B/2$ for all $j \in [1, 3m]$, decide whether there is a complete 3-partition of $\mathcal{S}$ s.t. each partition sums to B.*

- We can encode an instance $(B, \mathcal{S})$ as a symbolic heap in ASL.

Satisfiability, lower bound

- NP-hardness follows by reduction from

  3-partition problem. *Given* $B \in \mathbb{N}$ *and a sequence of natural numbers* $\mathcal{S} = (k_1, k_2, \ldots, k_{3m})$ *with* $\sum_{j=1}^{3m} k_j = mB$ *and* $B/4 < k_j < B/2$ *for all* $j \in [1, 3m]$*, decide whether there is a complete 3-partition of* $\mathcal{S}$ *s.t. each partition sums to* B*.*

- We can encode an instance $(B, \mathcal{S})$ as a symbolic heap in ASL.
- Roughly, the idea is that we have $m + 1$ "delimiters" d_i at intervals of B cells, and $3m$ arrays of length k_j.

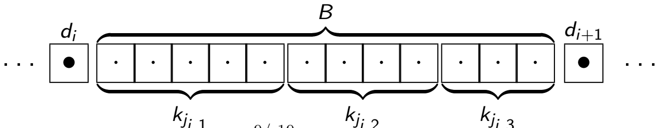
Satisfiability, lower bound

- NP-hardness follows by reduction from

  3-partition problem. *Given $B \in \mathbb{N}$ and a sequence of natural numbers $\mathcal{S} = (k_1, k_2, \ldots, k_{3m})$ with $\sum_{j=1}^{3m} k_j = mB$ and $B/4 < k_j < B/2$ for all $j \in [1, 3m]$, decide whether there is a complete 3-partition of $\mathcal{S}$ s.t. each partition sums to B.*

- We can encode an instance $(B, \mathcal{S})$ as a symbolic heap in ASL.

- Roughly, the idea is that we have $m + 1$ "delimiters" d_i at intervals of B cells, and $3m$ arrays of length k_j. We can fit all the arrays between the d_i iff there is a 3-partition:



Biabduction

Biabduction problem for ASL. *Given satisfiable symbolic heaps A and B, find symbolic heaps X and Y such that $A * X$ is satisfiable and $A * X \models B * Y$.*

Biabduction

Biabduction problem for ASL. *Given satisfiable symbolic heaps A and B, find symbolic heaps X and Y such that $A * X$ is satisfiable and $A * X \models B * Y$.*

- We concentrate on the quantifier-free case.

Biabduction

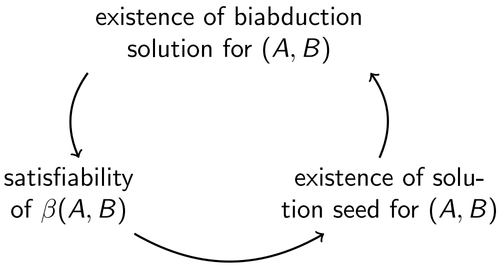
Biabduction problem for ASL. *Given satisfiable symbolic heaps A and B, find symbolic heaps X and Y such that $A * X$ is satisfiable and $A * X \models B * Y$.*

- We concentrate on the quantifier-free case.
- Our approach, diagrammatically, is as follows:

Biabduction

Biabduction problem for ASL. *Given satisfiable symbolic heaps A and B, find symbolic heaps X and Y such that $A * X$ is satisfiable and $A * X \models B * Y$.*

- We concentrate on the quantifier-free case.
- Our approach, diagrammatically, is as follows:



The formula $\beta(A, B)$

- Let (A, B) be an instance of the biabduction problem, where

$$A = \Pi : \mathop{\ast}_{i=1}^{n} \mathsf{array}(a_i, b_i) * \mathop{\ast}_{i=1}^{k} t_i \mapsto u_i$$

$$B = \Pi' : \mathop{\ast}_{i=1}^{m} \mathsf{array}(c_i, d_i) * \mathop{\ast}_{i=1}^{\ell} v_i \mapsto w_i$$

The formula $\beta(A, B)$

- Let (A, B) be an instance of the biabduction problem, where

$$A = \Pi : \mathop{\ast}_{i=1}^{n} \mathsf{array}(a_i, b_i) * \mathop{\ast}_{i=1}^{k} t_i \mapsto u_i$$

$$B = \Pi' : \mathop{\ast}_{i=1}^{m} \mathsf{array}(c_i, d_i) * \mathop{\ast}_{i=1}^{\ell} v_i \mapsto w_i$$

- For a solution to exist, we need to know that
  - A and B are simultaneously satisfiable; and

The formula $\beta(A, B)$

- Let (A, B) be an instance of the biabduction problem, where

$$A = \Pi : \mathop{\huge *}_{i=1}^{n} \mathsf{array}(a_i, b_i) * \mathop{\huge *}_{i=1}^{k} t_i \mapsto u_i$$

$$B = \Pi' : \mathop{\huge *}_{i=1}^{m} \mathsf{array}(c_i, d_i) * \mathop{\huge *}_{i=1}^{\ell} v_i \mapsto w_i$$

- For a solution to exist, we need to know that
  - A and B are simultaneously satisfiable; and
  - pointers $v_j \mapsto w_j$ in B are either covered by pointers $t_i \mapsto u_i$ in A with the right data value $(t_i = v_j \wedge u_i = w_j)$, or else not covered by anything in A.

The formula $\beta(A, B)$

- Let (A, B) be an instance of the biabduction problem, where

$$A = \Pi : \bigast_{i=1}^{n} \mathsf{array}(a_i, b_i) * \bigast_{i=1}^{k} t_i \mapsto u_i$$

$$B = \Pi' : \bigast_{i=1}^{m} \mathsf{array}(c_i, d_i) * \bigast_{i=1}^{\ell} v_i \mapsto w_i$$

- For a solution to exist, we need to know that
  - A and B are simultaneously satisfiable; and
  - pointers $v_j \mapsto w_j$ in B are either covered by pointers $t_i \mapsto u_i$ in A with the right data value $(t_i = v_j \wedge u_i = w_j)$, or else not covered by anything in A.
- This can be coded up as a Presburger formula $\beta(A, B)$, using the $\gamma(-)$ encoding of satisfiability.

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:
  1. Δ is satisfiable, and $\Delta \models \beta(A, B)$;

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:
  1. Δ is satisfiable, and $\Delta \models \beta(A, B)$;
  2. each δ_i is of the form $(t < u)$ or $(t = u)$, where $t, u \in \mathcal{T}_{A,B}$;

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:
  1. Δ is satisfiable, and $\Delta \models \beta(A, B)$;
  2. each δ_i is of the form $(t < u)$ or $(t = u)$, where $t, u \in \mathcal{T}_{A,B}$;
  3. all terms in $\mathcal{T}_{A,B}$ are ordered by a conjunct of Δ.

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:
  1. Δ is satisfiable, and $\Delta \models \beta(A, B)$;
  2. each δ_i is of the form $(t < u)$ or $(t = u)$, where $t, u \in \mathcal{T}_{A,B}$;
  3. all terms in $\mathcal{T}_{A,B}$ are ordered by a conjunct of Δ.
- That is, solution seeds enforce a total ordering on $\mathcal{T}_{A,B}$, including all array bounds and pointer addresses.

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:
  1. Δ is satisfiable, and $\Delta \models \beta(A, B)$;
  2. each δ_i is of the form $(t < u)$ or $(t = u)$, where $t, u \in \mathcal{T}_{A,B}$;
  3. all terms in $\mathcal{T}_{A,B}$ are ordered by a conjunct of Δ.
- That is, solution seeds enforce a total ordering on $\mathcal{T}_{A,B}$, including all array bounds and pointer addresses.
- It is fairly straightforward to show
  - $\exists$ biabduction soln. for $(A, B) \Rightarrow \beta(A, B)$ is satisfiable;

Solution seeds

- Write $\mathcal{T}_{A,B}$ for the set of all terms in A and B. A solution seed for (A, B) is a pure formula $\Delta = \bigwedge_{i \in I} \delta_i$ such that:
  1. Δ is satisfiable, and $\Delta \models \beta(A, B)$;
  2. each δ_i is of the form $(t < u)$ or $(t = u)$, where $t, u \in \mathcal{T}_{A,B}$;
  3. all terms in $\mathcal{T}_{A,B}$ are ordered by a conjunct of Δ.
- That is, solution seeds enforce a total ordering on $\mathcal{T}_{A,B}$, including all array bounds and pointer addresses.
- It is fairly straightforward to show
  - $\exists$ biabduction soln. for $(A, B) \Rightarrow \beta(A, B)$ is satisfiable;
  - $\beta(A, B)$ is satisfiable $\Rightarrow \exists$ solution seed for (A, B).

From seeds to solutions

- A seed defines a total ordering of all array endpoints and pointer addresses in A and B.

From seeds to solutions

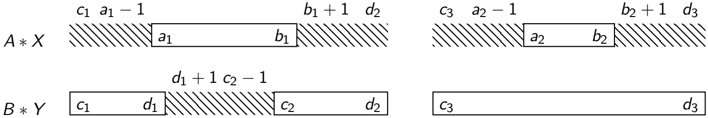
- A seed defines a total ordering of all array endpoints and pointer addresses in A and B.
- Given this info, computing X and Y becomes a relatively simple (PTIME) process!

From seeds to solutions

- A seed defines a total ordering of all array endpoints and pointer addresses in A and B.
- Given this info, computing X and Y becomes a relatively simple (PTIME) process!
- First we compute X by covering every array / pointer in B not already covered by A; then we compute Y the same way:

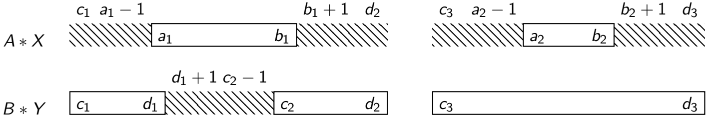
From seeds to solutions

- A seed defines a total ordering of all array endpoints and pointer addresses in A and B.
- Given this info, computing X and Y becomes a relatively simple (PTIME) process!
- First we compute X by covering every array / pointer in B not already covered by A; then we compute Y the same way:



From seeds to solutions

- A seed defines a total ordering of all array endpoints and pointer addresses in A and B.
- Given this info, computing X and Y becomes a relatively simple (PTIME) process!
- First we compute X by covering every array / pointer in B not already covered by A; then we compute Y the same way:



- We have to be a little careful about the pointer / array distinction though.

Lower bounds and quantification

- Quantifier-free case is NP-hard, again by reduction from the 3-partition problem.

Lower bounds and quantification

- Quantifier-free case is NP-hard, again by reduction from the 3-partition problem.
- When we disallow $\exists$ over R-values ($\exists y . x \mapsto y$) in B, the problem is equivalent to the quantifier-free case.

Lower bounds and quantification

- Quantifier-free case is NP-hard, again by reduction from the 3-partition problem.
- When we disallow $\exists$ over R-values ($\exists y.x \mapsto y$) in B, the problem is equivalent to the quantifier-free case.
- Otherwise, we get Π_2^P-hardness by reduction from

Lower bounds and quantification

- Quantifier-free case is NP-hard, again by reduction from the 3-partition problem.
- When we disallow $\exists$ over R-values ($\exists y.x \mapsto y$) in B, the problem is equivalent to the quantifier-free case.
- Otherwise, we get Π_2^P-hardness by reduction from

  2-**round** 3-**colourability problem** Given an undirected graph G, decide whether every 3-colouring of the leaves can be extended to a 3-colouring of G, such that no two adjacent vertices have the same colour.

Lower bounds and quantification

- Quantifier-free case is NP-hard, again by reduction from the 3-partition problem.
- When we disallow $\exists$ over R-values ($\exists y.x \mapsto y$) in B, the problem is equivalent to the quantifier-free case.
- Otherwise, we get Π_2^P-hardness by reduction from

  2-round 3-colourability problem Given an undirected graph G, decide whether every 3-colouring of the leaves can be extended to a 3-colouring of G, such that no two adjacent vertices have the same colour.
- (Given G, we define A_G to encode a 3-colouring of the leaves, and B_G to encode a 3-colouring of G.)

Entailment, upper bound

Entailment problem for ASL**.** *Given symbolic heaps A and B, decide whether $A \models B$.*

Entailment, upper bound

Entailment problem for ASL. *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:

Entailment, upper bound

Entailment problem for ASL**.** *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:
    1. B becomes unsatisfiable; or

Entailment, upper bound

Entailment problem for ASL**.** *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:
  1. B becomes unsatisfiable; or
  2. some heap location is covered by an array or pointer in A, but not by any array or pointer in B, or vice versa; or

Entailment, upper bound

Entailment problem for ASL**.** *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:
  1. B becomes unsatisfiable; or
  2. some heap location is covered by an array or pointer in A, but not by any array or pointer in B, or vice versa; or
  3. the LHS of some pointer in B is covered by an array in A; or

Entailment, upper bound

Entailment problem for ASL. *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:
  1. B becomes unsatisfiable; or
  2. some heap location is covered by an array or pointer in A, but not by any array or pointer in B, or vice versa; or
  3. the LHS of some pointer in B is covered by an array in A; or
  4. some pointer in B is covered by a pointer in A, but their data contents disagree.

Entailment, upper bound

Entailment problem for ASL**.** *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:
  1. B becomes unsatisfiable; or
  2. some heap location is covered by an array or pointer in A, but not by any array or pointer in B, or vice versa; or
  3. the LHS of some pointer in B is covered by an array in A; or
  4. some pointer in B is covered by a pointer in A, but their data contents disagree.

- Thus we can encode existence of a countermodel as a Σ^0_2 Presburger formula. Entailment becomes a Π^0_2 formula.

Entailment, upper bound

Entailment problem for ASL. *Given symbolic heaps A and B, decide whether $A \models B$.*

- Intuition: a stack s yields a countermodel for $A \models B$ if A is satisfiable under s and for every instantiation of existential variables $\mathbf{z}$, either:
  1. B becomes unsatisfiable; or
  2. some heap location is covered by an array or pointer in A, but not by any array or pointer in B, or vice versa; or
  3. the LHS of some pointer in B is covered by an array in A; or
  4. some pointer in B is covered by a pointer in A, but their data contents disagree.
- Thus we can encode existence of a countermodel as a Σ^0_2 Presburger formula. Entailment becomes a Π^0_2 formula.
- Due to item 3, we can't allow $\exists$ over R-values in pointers.

Entailment, lower bound

- We get Π_2^P-hardness of entailment, even for restricted $\exists$ quantifiers, by reduction from the previous colourability problem.

Entailment, lower bound

- We get Π_2^P-hardness of entailment, even for restricted $\exists$ quantifiers, by reduction from the previous colourability problem.
- Let's not go into the details!

Entailment, lower bound

- We get Π_2^P-hardness of entailment, even for restricted $\exists$ quantifiers, by reduction from the previous colourability problem.
- Let's not go into the details!
- This gives a gap in our complexity bounds for entailment:

Entailment, lower bound

- We get Π_2^P-hardness of entailment, even for restricted $\exists$ quantifiers, by reduction from the previous colourability problem.
- Let's not go into the details!
- This gives a gap in our complexity bounds for entailment:
    - lower bound of Π_2^P;

Entailment, lower bound

- We get Π_2^P-hardness of entailment, even for restricted $\exists$ quantifiers, by reduction from the previous colourability problem.
- Let's not go into the details!
- This gives a gap in our complexity bounds for entailment:
  - lower bound of Π_2^P;
  - upper bound of Π_1^{EXP} in the exponential-time hierarchy.

Entailment, lower bound

- We get Π_2^P-hardness of entailment, even for restricted $\exists$ quantifiers, by reduction from the previous colourability problem.
- Let's not go into the details!
- This gives a gap in our complexity bounds for entailment:
    - lower bound of Π_2^P;
    - upper bound of Π_1^{EXP} in the exponential-time hierarchy.
- I suspect the upper bound is closer to the "true complexity".

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.
- Our biabduction algorithm could be improved:

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.
- Our biabduction algorithm could be improved:
  - commit to as little ordering as possible;

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.
- Our biabduction algorithm could be improved:
  - commit to as little ordering as possible;
  - find heuristics for improving solution quality.

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.
- Our biabduction algorithm could be improved:
  - commit to as little ordering as possible;
  - find heuristics for improving solution quality.
- One could also try to do biabduction proof-theoretically.

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.
- Our biabduction algorithm could be improved:
  - commit to as little ordering as possible;
  - find heuristics for improving solution quality.
- One could also try to do biabduction proof-theoretically.
- Another essential program analysis component is abstraction heuristics for finding invariants, etc.

Future work

- Obvious thing to do: implement a prototype INFER-style analysis for array programs.
- Our biabduction algorithm could be improved:
  - commit to as little ordering as possible;
  - find heuristics for improving solution quality.
- One could also try to do biabduction proof-theoretically.
- Another essential program analysis component is abstraction heuristics for finding invariants, etc.
- Extension of ASL with more expressive features (e.g. combine with list segments?).

Conclusions

- We propose ASL, a version of symbolic-heap separation logic for arrays.

Conclusions

- We propose ASL, a version of symbolic-heap separation logic for arrays.
- Biabduction is the most critical step in inferring specifications of whole programs.

Conclusions

- We propose ASL, a version of symbolic-heap separation logic for arrays.
- Biabduction is the most critical step in inferring specifications of whole programs.
- We give a sound, complete biabduction algorithm that runs in NP-time.

Conclusions

- We propose ASL, a version of symbolic-heap separation logic for arrays.
- Biabduction is the most critical step in inferring specifications of whole programs.
- We give a sound, complete biabduction algorithm that runs in NP-time.
- Indeed, biabduction is NP-complete, climbing higher when $\exists$ quantifiers are added.

Conclusions

- We propose ASL, a version of symbolic-heap separation logic for arrays.
- Biabduction is the most critical step in inferring specifications of whole programs.
- We give a sound, complete biabduction algorithm that runs in NP-time.
- Indeed, biabduction is NP-complete, climbing higher when $\exists$ quantifiers are added.
- We also establish decision procedures and complexity bounds for satisfiability and entailment.

Thanks for listening!

Paper available on arXiv:

`arXiv:1607.01993`