Constraint-based Program Reasoning with Heaps and Separation

Gregory J. Duck, Joxan Jaffar, and Nicolas C. H. Koh
National University of Singapore

Motivation

Reasoning about Heaps is hard

- $\{x = 4, y = 4\}$ `x = x + 1` $\{x = 5, y = 4\}$
- $\{*x = 4, *y = 4\}$ `*x = *x + 1` $\{*x = 5, *y = ???\}$
- Need to reason about $x = y$ (*aliasing*) at *verification* time
- Much harder when
  (a) using `malloc()` and `free()`
  (b) considering recursive data structures

Contribution

- A basis for *automatic* reasoning of heap-manipulating programs
- Method: *Symbolic Execution*

Background

- Naive Approach: *heap-as-a-giant-array*
  - \+ Constraint based (use an array constraint-solver)
  - – tedious to express "separation", e.g.
    $$list(H, l, F1) \wedge tree(H, t, F2) \wedge disjoint(F1, F2)$$
  - – Difficult to solve

- *Separation Logic* was a breakthrough
  - \+ supports local reasoning (Hoare triples, Frame Rule)
  - \+ Separation is explicated at the top-level on ONE heap:
    $$list(l) * tree(t)$$
  - – Defined as a logic
    No general algorithms (sym. execution, solving)

- Our Contribution:
  Reformulate Separation Logic into a constraint language:
  - \+ Constraint-based (like arrays)
  - \+ Representation of *multilple* heaps, and separation
  - \+ Program reasoning becomes constraint solving; have a solver

Part I

A Constraint Language for Heaps

Heaps

- DEFINITION: A *heap* is a *finite partial map* between integers

$$\text{Heaps} = \text{Values} \rightharpoonup_{\text{fin}} \text{Values}$$

- DEFINITION (alternative): A *heap* h is a set of tuples such that

$$\forall p, v, w : (p, v) \in h \wedge (p, w) \in w \Longrightarrow v = w$$

$\mathcal{H}$-Language

- DEFINITION: $\mathcal{H}$ is a first-order language over the Heaps.

  1. (*Empty Heap*):

  $$\emptyset \quad \stackrel{\text{def}}{=} \quad \text{a Heap with no elements}$$

  2. (*Singleton Heap*):

  $$p \mapsto v \quad \stackrel{\text{def}}{=} \quad \text{a Heap with exactly one element } (p, v)$$

  3. (*Separation*)

  $$(H \doteq H_1 * \ldots * H_n) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \text{Heaps } H_1, .., H_n \text{ are } \textit{separate/disjoint} \\ H = H_1 \cup .. \cup H_n \text{ as sets.} \end{cases}$$

- NOTE: $(\doteq) \not\equiv (=)$
  ($\doteq$ is *partial equality* w.r.t. $(*)$)

Program Reasoning with $\mathcal{H}$

- DEFINE: $\bar{\mathcal{H}} \in$ Heaps as the *Program Heap*
- Standard memory operations can be mapped to $\mathcal{H}$:

C Syntax	$\mathcal{H}$ *Encoding*
`v = p[0];`	$\exists H : \bar{\mathcal{H}} \doteq (p \mapsto v) * H$
`p = malloc(1);`	$\exists H, v : \bar{\mathcal{H}} \doteq (p \mapsto v) * H$
`free(p);`	$\exists H, v : H \doteq (p \mapsto v) * \bar{\mathcal{H}}$
`p[0] = v;`	$\exists H, H', w : \begin{cases} H \doteq (p \mapsto w) * H' \\ \bar{\mathcal{H}} \doteq (p \mapsto v) * H' \end{cases}$

Hoare Triples

- We use *Hoare Triples* to reason about programs.

$$\{\textit{Precondition}\}\ \texttt{Code}\ \{\textit{Postcondition}\}$$

  where

  1. *Precondition* = proposition about the *initial state*
  2. `Code` = code/program fragment
  3. *Postcondition* = proposition about the *final state*

- EXAMPLES:

  $\{x < y\}$ `x := x + 1` $\{x \leq y\}$ ✓

  ⋮

Note: (Hoare Logic is traditionally heap-free.)

Hoare Triples (cont.)

- *Access*:

$$\{\phi\}\ x := [y]\ \{\exists x', H' : \bar{\mathcal{H}} \doteq (y \mapsto x) * H' \wedge \phi[x'/x]\}$$

- *Assignment*:

$$\{\phi\}\ [x] := y\ \{\exists H', H'', v : \wedge \begin{array}{l} H' \doteq (x \mapsto v) * H'' \\ \bar{\mathcal{H}} \doteq (x \mapsto y) * H'' \end{array} \wedge \phi[H'/\bar{\mathcal{H}}]\}$$

- *Allocation*:

$$\{\phi\}\ x := \mathbf{alloc}(1)\ \{\exists x', v, H' : \bar{\mathcal{H}} \doteq (x \mapsto v) * H' \wedge \phi[H'/\bar{\mathcal{H}}, x'/x]\}$$

- *Deallocation*:

$$\{\phi\}\ \mathbf{free}(x)\ \{\exists H', v : H' \doteq (x \mapsto v) * \bar{\mathcal{H}} \wedge \phi[H'/\bar{\mathcal{H}}]\}$$

Symbol Execution with $\mathcal{H}$

- Hoare triples are in "*Strongest Post Condition*" (SPC) form

$$\forall \phi : \{\phi\}\ \mathit{Code}\ \{\mathit{SPC}(\mathit{Code}, \phi)\}$$

- SPC $\implies$ Automation via *Symbolic Execution*.

PROVE:

$$\{P\}\ \mathit{Code}\ \{Q\}$$

STEPS:

1. Use Hoare rules to compute $\mathit{SPC}(\mathit{Code}, P)$;
2. Prove (via a theorem prover) that

$$\mathit{SPC}(\mathit{Code}, P) \rightarrow Q$$

3. QED

Symbolic Execution with $\mathcal{H}$ (cont.)

- EXAMPLE: $\{H \doteq \bar{\mathcal{H}}\}\ x := \mathbf{alloc}(); \mathbf{free}(x)\ \{H \doteq \bar{\mathcal{H}}\}$ (1)
- Use Symbolic Execution to compute the SPC:

  $\{H \doteq \bar{\mathcal{H}}\} \quad x := \mathbf{alloc}(); \mathbf{free}(x)$

  $x := \mathbf{alloc}(); \quad \{H \doteq H_0 \wedge \bar{\mathcal{H}} \doteq (x \mapsto \_) * H_0\} \quad \mathbf{free}(x)$

  $x := \mathbf{alloc}(); \mathbf{free}(x) \quad \{H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}}\}$

  Since

  $$H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}} \rightarrow H \doteq \bar{\mathcal{H}} \quad (2)$$

  Triple (1) holds; QED
- ...but how to prove (2)?

Part II

A Solver for $\mathcal{H}$

A Solver for $\mathcal{H}$

- Symbolic Execution generates *Verification Conditions* of the form $SPC(C, P) \rightarrow Q$, e.g.:

$$H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}} \rightarrow H \doteq \bar{\mathcal{H}}$$

  holds iff

$$H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}} \wedge H \neq \bar{\mathcal{H}}$$

  is UNSAT.

- Approach:
  - STEP 1: *Normalization*
  - STEP 2: Constraint solver (hsolve) for flat $\mathcal{H}$-formulae
  - STEP 3: DPLL(hsolve) for the Boolean structure.

STEP 1: Normalization

- W.l.o.g. we can restrict $\mathcal{H}$ to three basic constraints:

Description	*Constraint*
(*Heap Empty*)	$H \doteq \emptyset$
(*Heap Singleton*)	$H \doteq (p \mapsto v)$
(*Heap Separation*)	$H \doteq H_1 * H_2$

- THEOREM: We can *normalize* arbitrary $\mathcal{H}$-formulae to these basic constraints

$$H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}} \wedge H \neq \bar{\mathcal{H}}$$

$$\downarrow$$

$$T_1 \doteq \emptyset \wedge H \doteq H_0 * T_1 \wedge T_2 \doteq (x \mapsto \_) \wedge H_1 \doteq T_2 * H_0 \wedge T_3 \doteq (x \mapsto \_) \wedge H_1 \doteq T_3 * \bar{\mathcal{H}} \wedge$$
$$(T_4 \doteq (s \mapsto t) \wedge T_5 \doteq (s \mapsto u) \wedge H \doteq T_4 * T_6 \wedge \bar{\mathcal{H}} \doteq T_5 * T_7 \wedge t \neq u \vee$$
$$H \doteq T_8 * T_9 \wedge \bar{\mathcal{H}} \doteq T_8 * T_{10} \wedge T_{11} \doteq T_9 * T_{10} \wedge T_{12} \doteq (x \mapsto y) \wedge T_{11} \doteq T_{12} * T_{13})$$

STEP 1: Normalization (cont.)

PROOF: $\mathcal{H}$ Normalization Rules (see paper)

$$
\begin{array}{rcll}
H \doteq E_1 * E_2 * S & \longrightarrow & H' \doteq E_1 * E_2 \wedge H \doteq H' * S & \\
H \doteq E_1 * E_2 & \longrightarrow & H' \doteq E_1 \wedge H \doteq H' * E_2 & (E_1 \text{ non-variable}) \\
H \doteq H_1 * E_2 & \longrightarrow & H' \doteq E_2 \wedge H \doteq H_1 * H' & (E_2 \text{ non-variable}) \\
H_1 \doteq H_2 & \longrightarrow & H' \doteq \emptyset \wedge H_1 \doteq H_2 * H' & \\
H \not\doteq E_1 * E_2 * S & \longrightarrow & \vee \begin{cases} E_1 \doteq (s \mapsto t) * H_1' \wedge E_2 \doteq (s \mapsto u) * H_2' \\ H' \doteq E_1 * E_2 \wedge H \not\doteq H' * S \end{cases} & \\
H \not\doteq E_1 * E_2 & \longrightarrow & H' \doteq E_1 \wedge H \not\doteq H' * E_2 & (E_1 \text{ non-variable}) \\
H \not\doteq H_1 * E_2 & \longrightarrow & H' \doteq E_2 \wedge H \not\doteq H_1 * H' & (E_2 \text{ non-variable}) \\
H \not\doteq \emptyset & \longrightarrow & H \doteq (s \mapsto t) * H' & \\
H \not\doteq (p \mapsto v) & \longrightarrow & \vee \begin{cases} H \doteq \emptyset \\ H \doteq (s \mapsto t) * H' \wedge (p \neq s \vee v \neq t) \end{cases} & \\
H \not\doteq H_1 * H_2 & \longrightarrow & \vee \begin{cases} H_1 \doteq (s \mapsto t) * H_1' \wedge H_2 \doteq (s \mapsto u) * H_2' \\ H' \doteq H_1 * H_2 \wedge H \not\doteq H' \end{cases} & \\
H_1 \not\doteq H_2 & \longrightarrow & \vee \begin{cases} H_1 \doteq (s \mapsto t) * H_1' \wedge H_2 \doteq (s \mapsto u) * H_2' \wedge t \neq u \\ H_1 \doteq I * H_1' \wedge H_2 \doteq I * H_2' \wedge H' \doteq H_1' * H_2' \wedge H' \not\doteq \emptyset \end{cases} &
\end{array}
$$

STEP 2: $\mathcal{H}$-Solver for Flat Constraints

- Basic idea: propagate *heap membership constraints*; define:

$$\mathsf{in}(H, p, v) \quad \stackrel{\text{def}}{=} \quad (p, v) \in H$$

- Heap membership *propagation rules*:
  - *Functional Dependency*:

$$\mathsf{in}(H, p, v) \wedge \mathsf{in}(H, p, w) \Longrightarrow v = w$$

  - *Empty Heap*:

$$H \doteq \emptyset \wedge \mathsf{in}(H, p, v) \Longrightarrow \mathit{false}$$

  - *Singleton Heap*:

$$H \doteq (p \mapsto v) \Longrightarrow \mathsf{in}(H, p, v)$$
$$H \doteq (p \mapsto v) \wedge \mathsf{in}(H, q, w) \Longrightarrow p = q \wedge v = w$$

STEP 2: $\mathcal{H}$-Solver (cont.)

- *Separation*:

$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H, p, v) \Longrightarrow \mathsf{in}(H_1, p, v) \vee \mathsf{in}(H_2, p, v)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H_1, p, v) \Longrightarrow \mathsf{in}(H, p, v)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H_2, p, v) \Longrightarrow \mathsf{in}(H, p, v)$$
$$H \doteq H_1 * H_2 \wedge \mathsf{in}(H_1, p, v) \wedge \mathsf{in}(H_2, q, w) \Longrightarrow p \neq q$$

- $\mathcal{H}$-Solver Algorithm (hsolve) = *Constraint Handling Rules with Disjunction*

> *"Given a constraint store S, repeatedly apply propagation rules until a fixed point is reached."*

Disjunction is handled by branching and backtracking.

STEP 2: $\mathcal{H}$-Solver Algorithm

$$\begin{aligned}
\mathsf{in}(H, p, v) \wedge \mathsf{in}(H, p, w) &\Longrightarrow v = w \\
H \doteq \emptyset \wedge \mathsf{in}(H, p, v) &\Longrightarrow \mathit{false} \\
H \doteq (p \mapsto v) &\Longrightarrow \mathsf{in}(H, p, v) \\
H \doteq (p \mapsto v) \wedge \mathsf{in}(H, q, w) &\Longrightarrow p = q \wedge v = w \\
H \doteq H_1 * H_2 \wedge \mathsf{in}(H, p, v) &\Longrightarrow \mathsf{in}(H_1, p, v) \vee \mathsf{in}(H_2, p, v) \\
H \doteq H_1 * H_2 \wedge \mathsf{in}(H_1, p, v) &\Longrightarrow \mathsf{in}(H, p, v) \\
H \doteq H_1 * H_2 \wedge \mathsf{in}(H_2, p, v) &\Longrightarrow \mathsf{in}(H, p, v) \\
H \doteq H_1 * H_2 \wedge \mathsf{in}(H_1, p, v) \wedge \mathsf{in}(H_2, q, w) &\Longrightarrow p \neq q
\end{aligned}$$

$H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), v \neq w$

$H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), v \neq w, \mathsf{in}(H, p, v)$

$H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), v \neq w, \mathsf{in}(H, p, v), \mathsf{in}(J, p, w)$

$H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), v \neq w, \mathsf{in}(H, p, v), \mathsf{in}(J, p, w), \mathsf{in}(H, p, w)$

$H \doteq (p \mapsto v), H \doteq I * J, J \doteq (p \mapsto w), v \neq w, \mathsf{in}(H, p, v), \mathsf{in}(J, p, w), v = w$

false

$\therefore$ Goal is UNSAT.

STEP 2: Main $\mathcal{H}$-Solver Results

Theorem (Soundness)

The $\mathcal{H}$-Solver is sound.

Proof: By the correctness of the CHR rules.

Theorem (Completeness)

The $\mathcal{H}$-Solver is complete.[1]

Proof: (see paper)

1. Assumes complete equality theory

STEP 3: DPLL(hsolve)

- DPLL(hsolve) for non-conjunctive goals, e.g.

$$T_1 \doteq \emptyset \wedge H \doteq H_0 * T_1 \wedge T_2 \doteq (x \mapsto \_) \wedge H_1 \doteq T_2 * H_0 \wedge T_3 \doteq (x \mapsto \_) \wedge H_1 \doteq T_3 * \bar{\mathcal{H}} \wedge$$
$$(T_4 \doteq (s \mapsto t) \wedge T_5 \doteq (s \mapsto u) \wedge H \doteq T_4 * T_6 \wedge \bar{\mathcal{H}} \doteq T_5 * T_7 \wedge t \neq u \vee$$
$$H \doteq T_8 * T_9 \wedge \bar{\mathcal{H}} \doteq T_8 * T_{10} \wedge T_{11} \doteq T_9 * T_{10} \wedge T_{12} \doteq (x \mapsto y) \wedge T_{11} \doteq T_{12} * T_{13})$$

$$\downarrow$$

$$b_1 \wedge b_2 \wedge b_3 \wedge b_4 \wedge b_5 \wedge b_6 \wedge (b_7 \wedge b_8 \wedge b_9 \wedge b_{10} \wedge \neg b_{11} \vee b_{12} \wedge b_{13} \wedge b_{14}) \wedge$$
$$b_1 \leftrightarrow T_1 \doteq \emptyset \wedge b_2 \leftrightarrow H \doteq H_0 * T_1 \wedge b_3 \leftrightarrow T_2 \doteq (x \mapsto \_) \wedge b_4 \leftrightarrow H_1 \doteq T_2 * H_0 \wedge$$
$$b_5 \leftrightarrow T_3 \doteq (x \mapsto \_) \wedge b_6 \leftrightarrow H_1 \doteq T_3 * \bar{\mathcal{H}} \wedge b_7 \leftrightarrow T_4 \doteq (s \mapsto t) \wedge b_8 \leftrightarrow T_5 \doteq (s \mapsto u) \wedge$$
$$b_9 \leftrightarrow H \doteq T_4 * T_6 \wedge b_{10} \leftrightarrow \bar{\mathcal{H}} \doteq T_5 * T_7 \wedge b_{11} \leftrightarrow t = u \wedge b_{12} \leftrightarrow T_{11} \doteq T_9 * T_{10} \wedge$$
$$b_{13} \leftrightarrow T_{12} \doteq (x \mapsto y) \wedge b_{14} \leftrightarrow T_{11} \doteq T_{12} * T_{13}$$

- DPLL($\mathcal{H}$) implemented in *Satisfiability Modulo Constraint Handling Rules* (SMCHR).
  Details/Download:
  http://www.comp.nus.edu.sg/~gregory/smchr.html

STEP 3: DPLL(hsolve) (cont.)

- EXAMPLE (complete):

```
$ ./smchr -s heaps,linear,eq
> emp(T_1) /\ sep(H, H_0, T_1) /\ one(T_2, x, v0) /\
    sep(H_1, T_2, H_0) /\ one(T_3, x, v1) /\ sep(H_1, T_3, Heap) /\
    ((one(T_4, s, t) /\ one(T_5, s, u) /\ sep(H, T_4, T_6) /\
    sep(Heap, T_5, T_7) /\ t != u) \ /
    (sep(H, T_8, T_9) /\ sep(Heap, T_8, T_10) /\ sep(T_11, T_9, T_10) /\
    one(T_12, x, y) /\ sep(T_11, T_12, T_13)))
UNSAT
```

Therefore:

$$T_1 \doteq \emptyset \wedge H \doteq H_0 * T_1 \wedge T_2 \doteq (x \mapsto \_) \wedge H_1 \doteq T_2 * H_0 \wedge T_3 \doteq (x \mapsto \_) \wedge H_1 \doteq T_3 * \bar{\mathcal{H}} \wedge (T_4 \doteq (s \mapsto t) \wedge T_5 \doteq (s \mapsto u) \wedge H \doteq T_4 * T_6 \wedge \bar{\mathcal{H}} \doteq T_5 * T_7 \wedge t \neq u \vee H \doteq T_8 * T_9 \wedge \bar{\mathcal{H}} \doteq T_8 * T_{10} \wedge T_{11} \doteq T_9 * T_{10} \wedge T_{12} \doteq (x \mapsto y) \wedge T_{11} \doteq T_{12} * T_{13})$$

is UNSAT. Therefore:

$$H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \bar{\mathcal{H}} \rightarrow H \doteq \bar{\mathcal{H}}$$

is VALID.Therefore:

$$\{H \doteq \bar{\mathcal{H}}\}\ x := \mathbf{alloc}(); \mathbf{free}(x)\ \{H \doteq \bar{\mathcal{H}}\}$$

Part III

Experiments

Experimental Results

- BENCHMARKS:
  1. subsets_N - sum-of-subsets
  2. expr_N - expression evaluation
  3. stack_N - stack
  4. filter_N - TCP/IP filtering
  5. sort_N - Bubblesort
  6. search234_N - 234-tree search
  7. insert234_N - 234-tree insert

  TRIPLES:

  (F) $\{\bar{\mathcal{H}} \doteq (p \mapsto v) * F\}\ C\ \{\exists F' : \bar{\mathcal{H}} \doteq (p \mapsto v) * F'\}$
  (OP) $\{H \doteq \bar{\mathcal{H}}\}\ C\ \{H\ OP\ \bar{\mathcal{H}}\}$
  (A) $\{\ldots\}\ C\ \{\exists F', v : \bar{\mathcal{H}} \doteq (p \mapsto v) * F'\}$
  ($\emptyset$) $\{\bar{\mathcal{H}} \doteq \emptyset\}\ C\ \{false\}$

  where $OP \in \{\sqsubseteq, \sqsupseteq, \doteq\}$

- We compare SMCHR($\mathcal{H}$) vs. Verifast (Separation Logic).

Experimental Results (cont.)

				Heaps		Verifast	
Bench.	Safety	LOC	type	time(s)	#bt	time(s)	#forks
subsets_16	F	50	`rw-`	0.00	17	10.69	65546
expr_2	F	69	`rw-`	0.05	124	18.38	136216
stack_80	F	976	`rwa`	8.66	320	68.20	9963
filter_1	F	192	`r--`	0.03	80	0.75	8134
filter_2	F	321	`r--`	0.11	307	–	–
sort_6	F	178	`rw-`	0.03	54	2.66	35909
search234_3	F	251	`r--`	0.02	46	0.67	1459
search234_5	F	399	`r--`	0.05	76	90.65	118099
insert234_5	F	839	`rwa`	1.19	120	52.87	36885
expr_2	$\sqsubseteq$	69	`rw-`	0.20	1329	n.a.	n.a.
stack_80	$\sqsubseteq$	976	`rwa`	8.07	322	n.a.	n.a.
filter_2	OP	321	`r--`	0.00	2	n.a.	n.a.
stack_80	A	976	`rwa`	8.90	320	65.68	9801
insert234_5	A	839	`rwa`	1.50	60	40.64	55423
subsets_16	$\emptyset$	50	`rw-`	0.00	33	n.a.	n.a.

Experimental Results (cont.)

- RESULTS:

  1. **Interpolation**: Constraint-based approach allows for search-space pruing a la no-good learning/interpolation.

  2. **Expressivity**: E.g. the (heap equivalence) triple:

     $$\{H \simeq \bar{\mathcal{H}}\}\ C\ \{H \simeq \bar{\mathcal{H}}\}$$

     cannot be directly expressed in Verifast/Separation Logic.

Conclusion

- Constraint language for expressive formulas about heaps
- Can conjoin with *recursive definitions* of heaps and other variables
- By symbolic execution, provided a basis for automatic program verification
- Practical solver because of reduction into simple *set-based* reasoning