Horn Clauses for Data Structures

Joxan Jaffar

National University of Singapore (NUS)

HCVS 2015, July 19, 2015

Overview

HOLY GRAIL: Automatic reasoning about Data Structures

- Assertion Language ($\mathcal{H}$, explicit heaps)
- Horn Clauses for Data Structures (CLP($\mathcal{H}$))
- Proving Horn Clauses (automatic induction)
- Local Reasoning, Compositional Proofs (frame rule)

$\mathcal{H}$-Language

- DEFINITION: A *heap* is a *finite partial map* between integers
$$\text{Heaps} = \text{Values} \rightharpoonup_{\text{fin}} \text{Values}$$
- DEFINITION: $\mathcal{H}$ is a first-order language over the Heaps.

1. (*Empty Heap*):
$$\Omega \overset{\mathit{def}}{=} \text{a Heap with no elements}$$
2. (*Singleton Heap*):
$$p \mapsto v \overset{\mathit{def}}{=} \text{a Heap with exactly one element } (p, v)$$
3. (*Separation*)
$$(H \doteq H_1 * \ldots * H_n) \overset{\mathit{def}}{=} \begin{cases} \text{Heaps } H_1, .., H_n \text{ are } \textit{separate/disjoint} \\ H = H_1 \cup .. \cup H_n \text{ as sets.} \end{cases}$$

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

Program Reasoning with $\mathcal{H}$

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

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

Hoare Triples (cont.)

- *Access*:

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

- *Assignment*:

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

- *Allocation*:

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

- *Deallocation*:

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

Symbol Execution with $\mathcal{H}$

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

$$\forall\phi : \langle \phi, Code, SPC(Code, \phi) \rangle$$

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

  PROVE:

$$\langle P, Code, Q \rangle$$

  STEPS:

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

$$SPC(Code, P) \rightarrow Q$$

  3. QED

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

- EXAMPLE: prove:

$$\langle H \doteq \mathcal{M}, x := \mathbf{alloc}(); \mathbf{free}(x), H \doteq \mathcal{M} \rangle \tag{1}$$

- Use Symbolic Execution to compute the SPC:

$$\begin{array}{l}
\{H \doteq \mathcal{M}\} \quad x := \mathbf{alloc}(); \mathbf{free}(x) \\
x := \mathbf{alloc}(); \quad \{H \doteq H_0 \wedge \mathcal{M} \doteq (x \mapsto \_) * H_0\} \quad \mathbf{free}(x) \\
x := \mathbf{alloc}(); \mathbf{free}(x) \quad \{H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \mathcal{M}\} \\
x := \mathbf{alloc}(); \mathbf{free}(x) \quad \{H \doteq H_0 \wedge H_1 \doteq (x \mapsto \_) * H_0 \wedge H_1 \doteq (x \mapsto \_) * \mathcal{M}\}
\end{array}$$

Since

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

Triple (1) holds; QED

- ...but how to prove (2)?

A Solver for $\mathcal{H}$

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

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

  holds iff

  $$H \doteq H_0 \wedge H_1 \doteq (x{\mapsto}\_)*H_0 \wedge H_1 \doteq (x{\mapsto}\_)*\mathcal{H} \wedge H \not\doteq \mathcal{H}$$

  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*)	$H \bumpeq \Omega$
(*Heap Singleton*)	$H \bumpeq (p \mapsto v)$
(*Heap Separation*)	$H \bumpeq H_1 * H_2$

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

$$H \bumpeq H_0 \wedge H_1 \bumpeq (x \mapsto \_) * H_0 \wedge H_1 \bumpeq (x \mapsto \_) * \mathcal{M} \wedge H \not\bumpeq \mathcal{M}$$

$$\downarrow$$

$$\begin{gathered} T_1 \bumpeq \Omega \wedge H \bumpeq H_0 * T_1 \wedge T_2 \bumpeq (x \mapsto \_) \wedge H_1 \bumpeq T_2 * H_0 \wedge T_3 \bumpeq (x \mapsto \_) \wedge H_1 \bumpeq T_3 * \mathcal{H} \wedge \\ (T_4 \bumpeq (s \mapsto t) \wedge T_5 \bumpeq (s \mapsto u) \wedge H \bumpeq T_4 * T_6 \wedge \mathcal{H} \bumpeq T_5 * T_7 \wedge t \neq u \vee \\ H \bumpeq T_8 * T_9 \wedge \mathcal{M} \bumpeq T_8 * T_{10} \wedge T_{11} \bumpeq T_9 * T_{10} \wedge T_{12} \bumpeq (x \mapsto y) \wedge T_{11} \bumpeq T_{12} * T_{13}) \end{gathered}$$

STEP 1: Normalization (cont.)

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

$$
\begin{array}{rcll}
H \bumpeq E_1 * E_2 * S & \longrightarrow & H' \bumpeq E_1 * E_2 \wedge H \bumpeq H' * S & \\
H \bumpeq E_1 * E_2 & \longrightarrow & H' \bumpeq E_1 \wedge H \bumpeq H' * E_2 & (E_1 \text{ non-variable}) \\
H \bumpeq H_1 * E_2 & \longrightarrow & H' \bumpeq E_2 \wedge H \bumpeq H_1 * H' & (E_2 \text{ non-variable}) \\
H_1 \bumpeq H_2 & \longrightarrow & H' \bumpeq \Omega \wedge H_1 \bumpeq H_2 * H' & \\
H \not\bumpeq E_1 * E_2 * S & \longrightarrow & \bigvee \begin{cases} E_1 \bumpeq (s \mapsto t) * H'_1 \wedge E_2 \bumpeq (s \mapsto u) * H'_2 \\ H' \bumpeq E_1 * E_2 \wedge H \not\bumpeq H' * S \end{cases} & \\
H \not\bumpeq E_1 * E_2 & \longrightarrow & H' \bumpeq E_1 \wedge H \not\bumpeq H' * E_2 & (E_1 \text{ non-variable}) \\
H \not\bumpeq H_1 * E_2 & \longrightarrow & H' \bumpeq E_2 \wedge H \not\bumpeq H_1 * H' & (E_2 \text{ non-variable}) \\
H \not\bumpeq \emptyset & \longrightarrow & H \bumpeq (s \mapsto t) * H' & \\
H \not\bumpeq (p \mapsto v) & \longrightarrow & \bigvee \begin{cases} H \bumpeq \Omega \\ H \bumpeq (s \mapsto t) * H' \wedge (p \neq s \vee v \neq t) \end{cases} & \\
H \not\bumpeq H_1 * H_2 & \longrightarrow & \bigvee \begin{cases} H_1 \bumpeq (s \mapsto t) * H'_1 \wedge H_2 \bumpeq (s \mapsto u) * H'_2 \\ H' \bumpeq H_1 * H_2 \wedge H \not\bumpeq H' \end{cases} & \\
H_1 \not\bumpeq H_2 & \longrightarrow & \bigvee \begin{cases} H_1 \bumpeq (s \mapsto t) * H'_1 \wedge H_2 \bumpeq (s \mapsto u) * H'_2 \wedge t \neq u \\ H_1 \bumpeq I * H'_1 \wedge H_2 \bumpeq I * H'_2 \wedge H' \bumpeq H'_1 * H'_2 \wedge H' \not\bumpeq \Omega \end{cases} &
\end{array}
$$

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

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

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

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

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

  - *Empty Heap*:

$$H \risingdotseq \Omega \wedge \mathrm{in}(H, p, v) \Longrightarrow \mathit{false}$$

  - *Singleton Heap*:

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

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

- *Separation*:

$$
\begin{aligned}
H \doteq H_1 * H_2 \wedge \mathrm{in}(H, p, v) &\Longrightarrow \mathrm{in}(H_1, p, v) \vee \mathrm{in}(H_2, p, v) \\
H \doteq H_1 * H_2 \wedge \mathrm{in}(H_1, p, v) &\Longrightarrow \mathrm{in}(H, p, v) \\
H \doteq H_1 * H_2 \wedge \mathrm{in}(H_2, p, v) &\Longrightarrow \mathrm{in}(H, p, v) \\
H \doteq H_1 * H_2 \wedge \mathrm{in}(H_1, p, v) \wedge \mathrm{in}(H_2, q, w) &\Longrightarrow p \neq q
\end{aligned}
$$

- $\mathcal{H}$-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

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

$H \mathrel{\hat{=}} (p {\mapsto} v), H \mathrel{\hat{=}} I {*} J, J \mathrel{\hat{=}} (p {\mapsto} w), v \neq w$

$H \mathrel{\hat{=}} (p {\mapsto} v), H \mathrel{\hat{=}} I {*} J, J \mathrel{\hat{=}} (p {\mapsto} w), v \neq w, \mathrm{in}(H,p,v)$

$H \mathrel{\hat{=}} (p {\mapsto} v), H \mathrel{\hat{=}} I {*} J, J \mathrel{\hat{=}} (p {\mapsto} w), v \neq w, \mathrm{in}(H,p,v), \mathrm{in}(J,p,w)$

$H \mathrel{\hat{=}} (p {\mapsto} v), H \mathrel{\hat{=}} I {*} J, J \mathrel{\hat{=}} (p {\mapsto} w), v \neq w, \mathrm{in}(H,p,v), \mathrm{in}(J,p,w), \mathrm{in}(H,p,w)$

$H \mathrel{\hat{=}} (p {\mapsto} v), H \mathrel{\hat{=}} I {*} J, J \mathrel{\hat{=}} (p {\mapsto} w), v \neq w, \mathrm{in}(H,p,v), \mathrm{in}(J,p,w), v = w$

false

∴ 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.

$$\begin{gathered}
T_1 \bumpeq \Omega \wedge H \bumpeq H_0 * T_1 \wedge T_2 \bumpeq (x \mapsto \_) \wedge H_1 \bumpeq T_2 * H_0 \wedge T_3 \bumpeq (x \mapsto \_) \wedge H_1 \bumpeq T_3 * \mathcal{M} \wedge \\
(T_4 \bumpeq (s \mapsto t) \wedge T_5 \bumpeq (s \mapsto u) \wedge H \bumpeq T_4 * T_6 \wedge \mathcal{M} \bumpeq T_5 * T_7 \wedge t \neq u \vee \\
H \bumpeq T_8 * T_9 \wedge \mathcal{M} \bumpeq T_8 * T_{10} \wedge T_{11} \bumpeq T_9 * T_{10} \wedge T_{12} \bumpeq (x \mapsto y) \wedge T_{11} \bumpeq T_{12} * T_{13}) \\
\downarrow \\
b_1 \wedge b_2 \wedge b_3 \wedge b_4 \wedge b_5 \wedge b_6 \wedge (b_7 \wedge b_8 \wedge b_9 \wedge b_{10} \wedge \neg b_{11} \vee b_{12} \wedge b_{13} \wedge b_{14}) \wedge \\
b_1 \leftrightarrow T_1 \bumpeq \Omega \wedge b_2 \leftrightarrow H \bumpeq H_0 * T_1 \wedge b_3 \leftrightarrow T_2 \bumpeq (x \mapsto \_) \wedge b_4 \leftrightarrow H_1 \bumpeq T_2 * H_0 \wedge \\
b_5 \leftrightarrow T_3 \bumpeq (x \mapsto \_) \wedge b_6 \leftrightarrow H_1 \bumpeq T_3 * \mathcal{M} \wedge b_7 \leftrightarrow T_4 \bumpeq (s \mapsto t) \wedge b_8 \leftrightarrow T_5 \bumpeq (s \mapsto u) \wedge \\
b_9 \leftrightarrow H \bumpeq T_4 * T_6 \wedge b_{10} \leftrightarrow \mathcal{M} \bumpeq T_5 * T_7 \wedge b_{11} \leftrightarrow t = u \wedge b_{12} \leftrightarrow T_{11} \bumpeq T_9 * T_{10} \wedge \\
b_{13} \leftrightarrow T_{12} \bumpeq (x \mapsto y) \wedge b_{14} \leftrightarrow T_{11} \bumpeq T_{12} * T_{13}
\end{gathered}$$

- 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:

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

is UNSAT. Therefore:

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

is VALID. Therefore:

$$\langle H \bumpeq \mathcal{M}, x := \mathbf{alloc}(); \mathbf{free}(x), H \bumpeq \mathcal{M} \rangle$$

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) $\langle \mathcal{M} \doteqdot (p \mapsto v) * F, C, \exists F' : \mathcal{M} \doteqdot (p \mapsto v) * F' \rangle$
  (OP) $\langle H \doteqdot \mathcal{M}, C, H\ OP\ \mathcal{M} \rangle$
  (A) $\langle \ldots, C, \exists F', v : \mathcal{M} \doteqdot (p \mapsto v) * F' \rangle$
  (Ω) $\langle \mathcal{M} \doteqdot \Omega, C, \mathit{false} \rangle$

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

- 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	Ω	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:

     $$\langle H \doteq \mathcal{M}, C, H \doteq \mathcal{M} \rangle$$

     cannot be directly expressed in Verifast/Separation Logic.

Summary for $\mathcal{H}$

- Explicit heaps for expressiveness
- Promising Solver
- Symbolic Execution via Strongest Postcondition $\longrightarrow$ Automatic Verification of $\mathcal{H}$ assertions on whole-program, straight-line code

Overview (Recall)

- Assertion Language ($\mathcal{H}$, explicit heaps)
- Horn Clauses for Data Structures (CLP($\mathcal{H}$))
- Proving Horn Clauses (automatic induction)
- Local Reasoning, Compositional Proofs (frame rule)

CLP($\mathcal{H}$): Horn Clauses for Data Structures

Example: the predicate $\texttt{list}(h, x)$,
specifies a *skeleton list* in the heap h rooted at x.

$$\begin{array}{lll}\texttt{list}(h,x) & \texttt{:-} & h \doteq \Omega, x = \mathsf{null}\texttt{.} \\ \texttt{list}(h,x) & \texttt{:-} & h \doteq (x \mapsto y) \, * \, h_1\texttt{,}\ \texttt{list}(h_1, y)\texttt{.}\end{array}$$

CLP($\mathcal{H}$): Horn Clauses for Data Structures

```
struct node {
   int data;
   struct node *next;
};
```

$\{\ list(\mathcal{H}, x),\ \ \mathcal{H} \sqsubseteq \mathcal{M}\ \ \}$

```
y = x;
while (y) {
   y->data++;
   y = y->next;
}
```

$\{\ increment\_list(\mathcal{H}_1, \mathcal{H}, x),\ \ \mathcal{H}_1 \sqsubseteq \mathcal{M}\ \ \}$

where the predicate `increment_list` is defined as follows.

```
increment_list($h_1, h_2, x$) :-
   $h_1 \doteq \Omega, h_2 \doteq \Omega, x = \mathrm{null}$.
increment_list($h_1, h_2, x$) :-
   $h_1 \doteq (x \mapsto (d+1, next)) * h'_1$,
   $h_2 \doteq (x \mapsto (d, next)) * h'_2$,
   increment_list($h1'$,  $h'_2$,  $next$).
```

Note: this is an example of a *summary*

Overview (Recall)

- Assertion Language ($\mathcal{H}$, explicit heaps)
- Horn Clauses for Data Structures (CLP($\mathcal{H}$))
- Proving Horn Clauses (automatic induction)
- Local Reasoning, Compositional Proofs (Frame Rule)

How to prove Predicates in Assertions?

- Verifying functional correctness of dynamic data structures
- Properties are formalized using a logic of heaps and separation
  - A core feature is the use of user-defined recursive predicates
- The Problem: entailment checking, where both LHS and RHS involve such predicates
- A fully automatic solution is not possible
- The state-of-the-art for automatic methods is inadequate

The State-of-the-Art: Unfold-and-Match

- Performs systematic *folding* and *unfolding* steps of the recursive rules, and succeeds when we produce a formula which is *obviously provable*:
  - no recursive predicate in RHS of the proof obligation, and a direct proof can be achieved by consulting some generic SMT solver;
  - no special consideration is needed on any occurrence of a predicate appearing in the formula, i.e., *formula abstraction* can be applied.
- Notable systems: DRYAD and HIP/SLEEK

Example: Unfold-and-Match

Consider $\widehat{\mathsf{ls}}(x,y) \overset{def}{=} x{=}y \wedge \mathbf{emp} \;\mid\; x{\neq}y \wedge (\mathtt{x}{\mapsto}\mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},y)$

```
Pre:  ls^(x,y)
   assume(x != y)
   z = x.next
Post:  ls^(z,y)
```

Unfold the precondition $\widehat{\mathsf{ls}}(x,y)$

- Case 1: holds because `(x = y)` and `assume(x != y)` implies *false*
- Case 2: holds by matching `z` with `t`

Shortcomings

1. Recursion Divergence: when the “recursion” in the recursive rules is structurally *dissimilar* to the program code
2. Generalization of Predicate: when the predicate describing a loop invariant or a function is used later to prove a weaker property

(occurs often in practice, especially in iterative programs)

Recursion Divergence

- When the "recursion" in the recursive rules is structurally *dissimilar* to the program code

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) * (\mathtt{y} \mapsto \_)$$

```
  z = y.next
```

$$\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{z})$$

Fundamentally, it is about relating two definitions of a list segment: (recurse rightwards, and recurse leftwards)

$$\begin{aligned}
\widehat{\mathsf{ls}}(\mathtt{x},\mathtt{y}) &\overset{def}{=} \mathtt{x}=\mathtt{y} \wedge \mathbf{emp} \mid \mathtt{x}\neq\mathtt{y} \wedge (\mathtt{x}\mapsto\mathtt{t}) * \widehat{\mathsf{ls}}(\mathtt{t},\mathtt{y}) \\
\mathsf{ls}(\mathtt{x},\mathtt{y}) &\overset{def}{=} \mathtt{x}=\mathtt{y} \wedge \mathbf{emp} \mid \mathtt{x}\neq\mathtt{y} \wedge (\mathtt{t}\mapsto\mathtt{y}) * \mathsf{ls}(\mathtt{x},\mathtt{t})
\end{aligned}$$

(sometimes inevitable, e.g., queue implementation using list segment)

Generalization of Predicate:

- When the predicate describing a loop invariant or a function is used later to prove a weaker property
    - sorted_list(x, len, min) $\models$ list(x, len)
    - ls(x, y) * list(y) $\models$ list(x)

What is Needed: INDUCTION

- Traditional works on automated induction generally require variables of inductive type (so that the notions of base case and induction step are well-defined)
- Our predicates are (user-)defined over pointer variables, which are not inductive

The Specification Language

- We use the language $\mathcal{H}$, a logic with the features of *explicit heaps* and *a separation operator*
  - It facilitates symbolic execution and therefore VC generation
  - It has little/no bearing on the effectiveness of our induction method
- E.g. the below defines a skeleton list (we inherit the CLP semantics)

$$\begin{array}{lll} \mathrm{list}(x, L) & \texttt{:-} & x = 0,\ L \doteq \emptyset. \\ \mathrm{list}(x, L) & \texttt{:-} & L \doteq (x \mapsto t) * L_1,\ \mathrm{list}(t, L_1). \end{array}$$

(note that $*$ applies to terms, and not predicates as in traditional Separation Logic)

General Cut-Rule

$$\text{(CUT)}\ \frac{\mathcal{L}_1 \models \mathcal{R}_1 \qquad \mathcal{L}_2 \wedge \mathcal{R}_1 \models \mathcal{R}}{\mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}}$$

- Trivial from the deduction point of view (top to bottom)
- For proof derivation (bottom to top), obtaining an appropriate $\mathcal{R}_1$ is tantamount to a magic step
    - In manual proofs, we perform this magic step all the time
    - Automating this step is extremely hard

Induction Rule 1

$$(\textsc{Induction-1})\ \frac{\boxed{\mathcal{L}_1 \models \mathcal{R}_1} \qquad \mathcal{L}_2 \wedge \mathcal{R}_1 \models \mathcal{R}}{\mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}}$$

- $\boxed{\mathcal{L}_1 \models \mathcal{R}_1}$ is “the same” as some obligation encountered in the proof path (which acts as an induction hypothesis), thus it will be discharged immediately
- We discover $\mathcal{R}_1$ and proceed with the other obligation

Induction Rule 2

$$(\textsc{Induction-2})\ \frac{\mathcal{L}_1 \models \mathcal{R}_1 \qquad \boxed{\mathcal{L}_2 \wedge \mathcal{R}_1 \models \mathcal{R}}}{\mathcal{L}_1 \wedge \mathcal{L}_2 \models \mathcal{R}}$$

- $\boxed{\mathcal{L}_2 \wedge \mathcal{R}_1 \models \mathcal{R}}$ is “the same” as some obligation encountered in the proof path (which acts as an induction hypothesis), thus it will be discharged immediately
- We discover $\mathcal{R}_1$ and proceed with the other obligation

Summary

- Our automated induction rules allow for
  - a systematic method to discover $\mathcal{R}_1$ (in the cut-rule)
  - application of induction to discharge a proof obligation

- A significant technicality is to ensure induction applications do not lead to *circular* (i.e., wrong) reasoning

Example (simplified by ignoring heaps)

```
even(x) :-  x = 0.
even(x) :-  y = x − 2, even(y).

m4(x) :-  x = 0.
m4(x) :-  z = x − 4, m4(z).
```

$$\texttt{m4(x)} \models \texttt{even(x)}$$

- Unfold-and-Match will not work: there always remains obligation with predicate `m4` in the LHS and predicate `even` in the RHS

Example: Induction Works

$$
(\textsc{lu})\;\dfrac{
  (\textsc{ru})\;\dfrac{(\textsc{smt})\;\dfrac{\mathtt{True}}{\mathtt{x{=}0} \models \mathtt{x{=}0}}}{\mathtt{x{=}0} \models \mathtt{even(x)}}
  \qquad
  \dfrac{
    \boxed{\mathtt{m4(z)} \models \mathtt{even(z)}}
    \quad
    \dfrac{
      \dfrac{
        \dfrac{\mathtt{True}}{\mathtt{z{=}x{-}4}, \mathtt{even(z)} \models \mathtt{y{=}x{-}2}, \mathtt{t{=}y{-}2}, \mathtt{even(t)}}\;(\textsc{smt})
      }{\mathtt{z} = \mathtt{x} - 4, \mathtt{even(z)} \models \mathtt{y} = \mathtt{x} - 2,\ \mathtt{even(y)}}\;(\textsc{ru})
    }{\mathtt{z} = \mathtt{x} - 4, \mathtt{even(z)} \models \mathtt{even(x)}}\;(\textsc{ru})
  }{\mathtt{z} = \mathtt{x} - 4, \mathtt{m4(z)} \models \mathtt{even(x)}}\;(\textsc{i-1})
}{\mathtt{m4(x)} \models \mathtt{even(x)}}
$$

Example: Induction Works

$$(\textsc{lu})\ \dfrac{(\textsc{ru})\ \dfrac{(\textsc{smt})\ \dfrac{\textsc{True}}{\texttt{x=0} \models \texttt{x=0}}}{\texttt{x=0} \models \texttt{even(x)}} \qquad \vdots}{\texttt{m4(x)} \models \texttt{even(x)}}$$

Example: Induction Works

$$\text{(LU)}\;\dfrac{\text{(I-1)}\;\dfrac{\boxed{\mathtt{m4(z)} \models \mathtt{even(z)}} \quad \text{(RU)}\;\dfrac{\text{(RU)}\;\dfrac{\text{(SMT)}\;\dfrac{\mathtt{True}}{\mathtt{z}=\mathtt{x}-4, \mathtt{even}(\mathtt{z}) \models \mathtt{y}=\mathtt{x}-2, \mathtt{t}=\mathtt{y}-2, \mathtt{even}(\mathtt{t})}}{\mathtt{z} = \mathtt{x} - 4, \mathtt{even}(\mathtt{z}) \models \mathtt{y} = \mathtt{x} - 2,\ \mathtt{even}(\mathtt{y})}}{\mathtt{z} = \mathtt{x} - 4, \mathtt{even}(\mathtt{z}) \models \mathtt{even}(\mathtt{x})}}{\mathtt{z} = \mathtt{x} - 4, \mathtt{m4}(\mathtt{z}) \models \mathtt{even}(\mathtt{x})}}{\mathtt{m4}(\mathtt{x}) \models \mathtt{even}(\mathtt{x})}$$

- Applying induction rule 1, we discover `even(z)` as a candidate for a cut point.

  This step allows us to “flip” the predicate `even(z)` into the LHS so that subsequently Unfold-and-Match can work.

Results

- Proving commonly-used "lemmas" (or "axioms"); many existing systems simply accept them as facts from the users

$$\mathrm{sorted\_list}(x, min) \models \mathrm{list}(x)$$
$$\mathrm{sorted\_list}_1(x, len, min) \models \mathrm{list}_1(x, len)$$
$$\mathrm{sorted\_list}_1(x, len, min) \models \mathrm{sorted\_list}(x, min)$$
$$\mathrm{sorted\_ls}(x, y, min, max) * \mathrm{sorted\_list}(y, min_2) \land max \leq min_2 \models \mathrm{sorted\_list}(x, min)$$
$$\widehat{\mathrm{ls}}_1(x, y, len_1) * \widehat{\mathrm{ls}}_1(y, z, len_2) \models \widehat{\mathrm{ls}}_1(x, z, len_1 + len_2)$$
$$\mathrm{ls}_1(x, y, len_1) * \mathrm{list}_1(y, len_2) \models \mathrm{list}_1(x, len_1 + len_2)$$
$$\widehat{\mathrm{ls}}_1(x, last, len) * (last \mapsto new) \models \widehat{\mathrm{ls}}_1(x, new, len + 1)$$
$$\mathrm{avl}(x, hgt, min, max, balance) \models \mathrm{bstree}(x, hgt, min, max)$$
$$\mathrm{bstree}(x, height, min, max) \models \mathrm{bintree}(x, height)$$
$$\ldots$$

(running time ranges from 0.2 – 1 second per benchmark)

Results

- Eliminate the usage of lemmas: it indeed runs faster
  - we only look at the available induction hypotheses (0 – 3)
  - other systems look at all the "lemmas" (or "axioms")

Table: Verification of Open-Source Libraries.

Program	Function	T/F
glib/gslist.c `Singly Linked-List`	`find, position, index, nth,last,length,append, insert_at_pos,merge_sort, remove,insert_sorted_list`	<1s
glib/glist.c `Doubly Linked-List`	`nth, position, find, index, last, length`	<1s
OpenBSD/ queue.h `Queue`	`simpleq_remove_after, simpleq_insert_tail, simpleq_insert_after`	<1s
ExpressOS/ cachePage.c	`lookup_prev, add_cachepage`	<1s
linux/mmap.c	`insert_vm_struct`	<1s

What Next?

- Improve the robustness
  - e.g. works for $A \models B$, but might fail if we strengthen A (or weaken B)
  - having too strong antecedent (or too weak consequent) is an obstacle to the usage of induction

Overview (Recall)

Toward automatic reasoning about Data Structures

- Assertion Language ($\mathcal{H}$, explicit heaps)
- Horn Clauses for Data Structures (CLP($\mathcal{H}$))
- Proving Horn Clauses (automatic induction) **?**
- Local Reasoning, Compositional Proofs (Frame Rule)

Local / Compositional Reasoning

The Rule in Separation Logic which allows local reasoning:

$$\frac{\{\ \phi\ \}\ P\ \{\ \psi\ \}}{\{\ \phi * \pi\ \}\ P\ \{\ \psi * \pi\ \}}$$

the premise $\{\ \phi\ \}\ P\ \{\ \psi\ \}$ ensures that the implicit heap arising from the formula ϕ captures all the heap accesses, read or write, in the program fragment P.

The Frame Rule does not Apply with Explicit Heaps

- if $\{ \phi \}$ *P* $\{ \psi \}$ is established because ψ follows from the strongest postcondition of *P* executed from ϕ, it is not the case that any heap separate from ψ remains unchanged by the execution of *P*.
- because there are multiple heaps, only those which are affected by the program must be isolated.

Our new Frame Rule:

- used by specifying explicitly named *subheaps* in order to elegantly isolate relevant portions of the global heap.
- As a significant result, our frame rule is concerned only on heap *updates*, as opposed to being concerned about *all* heap references as in traditional SL.

Why do we need a Frame Rule?

- So far, only *straight-line* verification
- Loop invariants
- Procedure calls
- Local Reasoning / Compositional Proofs

All Heaps are Ghost except for the Global Heap $\mathcal{M}$

The postconditions shown are the *strongest postconditions*:

$$
\begin{array}{ll}
\{\ \phi\ \}\ x = \textbf{malloc}(1)\ \{\ \mathrm{alloc}(\phi, x)\ \} & \text{(Heap allocation)} \\
\{\ \phi\ \}\ \textbf{free}(x)\ \{\ \mathrm{free}(\phi, x)\ \} & \text{(Heap deallocation)} \\
\{\ \phi\ \}\ x = {*y}\ \{\ \mathrm{access}(\phi, y, x)\ \} & \text{(Heap access)} \\
\{\ \phi\ \}\ {*x} = y\ \{\ \mathrm{assign}(\phi, x, y)\ \} & \text{(Heap assignment)}
\end{array}
$$

where the auxiliary macros alloc, free, access, and assign expand as follows:

$$
\begin{array}{lcl}
\mathrm{alloc}(\phi, x) & \overset{\mathit{def}}{=} & \mathcal{M} \bumpeq (x \mapsto v) * \mathcal{H} \wedge \phi[\mathcal{H}/\mathcal{M}, v_1/x] \\
\mathrm{free}(\phi, x) & \overset{\mathit{def}}{=} & \mathcal{H} \bumpeq (x \mapsto v) * \mathcal{M} \wedge \phi[\mathcal{H}/\mathcal{M}] \\
\mathrm{access}(\phi, y, x) & \overset{\mathit{def}}{=} & \mathcal{M} \bumpeq (y \mapsto x) * \mathcal{H} \wedge \phi[v/x] \\
\mathrm{assign}(\phi, x, y) & \overset{\mathit{def}}{=} & \mathcal{M} \bumpeq (x \mapsto y) * \mathcal{H}_1 \wedge \\
 & & \mathcal{H} \bumpeq (x \mapsto v) * \mathcal{H}_1 \wedge \phi[\mathcal{H}/\mathcal{M}]
\end{array}
$$

where $\mathcal{H}$ and $\mathcal{H}_1$ are *fresh* heap variables, and v and v_1 are fresh value variables. ☐

Ghosts and Heap Reality

- User-defined Predicates use only ghost variables
- Connection the global heap is by means of $\mathcal{H} \sqsubseteq \mathcal{M}$
  ("heap reality" of $\mathcal{H}$)
- User-defined Predicates in an assertion can always be framed.
- What is interesting, therefore, is the preservation of heap reality

Example

```
struct node {
   int data;
   struct node *next;
};
```

$\{ \ \mathit{list}(\mathcal{H}, x), \ \ \mathcal{H} \sqsubseteq \mathcal{M} \ \}$

```
y = x;
while (y) {
   y->data++;
   y = y->next;
}
```

$\{ \ \mathit{increment\_list}(\mathcal{H}_1, \mathcal{H}, x), \ \ \mathcal{H}_1 \sqsubseteq \mathcal{M} \ \}$

where the predicate `increment_list` is defined as follows.

```
increment_list($h_1, h_2, x$) :-
   $h_1 \doteq \Omega, h_2 \doteq \Omega, x = \text{null}$.
increment_list($h_1, h_2, x$) :-
   $h_1 \doteq (x \mapsto (d+1, next)) * h'_1$,
   $h_2 \doteq (x \mapsto (d, next)) * h'_2$,
   increment_list($h1'$, $h'_2$, $next$).
```

Note: $\mathit{list}(\mathcal{H}, x)$ frames through, but not necessarily $\mathcal{H} \sqsubseteq \mathcal{M}$.

Heap Evolution

Let $T = \{\ \phi\ \}\ P\ \{\ \psi\ \}$ where $\tilde{\mathcal{H}}$ appears in ϕ and $\tilde{\mathcal{H}}'$ appears in ψ.
Then:

$$T \rightsquigarrow \tilde{\mathcal{H}}\ \triangleright\ \tilde{\mathcal{H}}'$$

means that the largest $\tilde{\mathcal{H}}'$ can be is $\tilde{\mathcal{H}}$ plus any new cells allocated by P, and minus any that are freed by P.

Usage: if $T \rightsquigarrow \tilde{\mathcal{H}}\ \triangleright\ \tilde{\mathcal{H}}'$ then any heap that is separate from $\tilde{\mathcal{H}}$ at the point of the precondition of T (i.e., before P is executed) will be separate from $\tilde{\mathcal{H}}'$ at the point of the postcondition (i.e., after P is executed).

EVOLUTION RULES (Basic)

$$\frac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \quad \mathit{dom}(\tilde{\mathcal{H}}') \subseteq \mathit{dom}(\tilde{\mathcal{H}}) \cup \{x\}}{\{\ \phi\ \}\ \texttt{x = malloc(1)}\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}\ \text{MALLOC}$$

$$\frac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \quad \mathit{dom}(\tilde{\mathcal{H}}') \subseteq \mathit{dom}(\tilde{\mathcal{H}}) \setminus \{x\}}{\{\ \phi\ \}\ \texttt{free(x)}\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}\ \text{FREE}$$

$$\frac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \quad \psi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M} \quad \mathit{dom}(\tilde{\mathcal{H}}') \subseteq \mathit{dom}(\tilde{\mathcal{H}})}{\{\ \phi\ \}\ s\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}\ \text{OTHER-STATEMENTS}$$

$$\frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}' \qquad \{\ \psi\ \}\ Q\ \{\ \gamma\ \} \leadsto \tilde{\mathcal{H}}' \rhd \tilde{\mathcal{H}}''}{\{\ \phi\ \}\ P; Q\ \{\ \gamma\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}''}\ \text{SEQ-COMPOSITION}$$

$$\frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'_1 \qquad \{\ \phi\ \}\ P\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'_2}{\{\ \phi\ \}\ P\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}} \rhd (\tilde{\mathcal{H}}'_1 \cup \tilde{\mathcal{H}}'_2)}\ \text{COMPOSITION}$$

EVOLUTION RULES (Structural)

$$\textsc{If-Then-Else}\quad \frac{\begin{array}{cc}\{\ \phi\ \}\ \mathit{assume}(b);P_1\ \{\ \psi_1\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'_1 & \tilde{\mathcal{H}}' \sqsubseteq \tilde{\mathcal{H}}'_1 \\ \{\ \phi\ \}\ \mathit{assume}(\neg b);P_2\ \{\ \psi_2\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'_2 & \tilde{\mathcal{H}}' \sqsubseteq \tilde{\mathcal{H}}'_2\end{array}}{\{\ \phi\ \}\ \textbf{if}\ (b)\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \{\ \psi_1 \vee \psi_2\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}$$

$$\textsc{Narrowing-Post}\quad \frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'_1 \qquad \tilde{\mathcal{H}}' \sqsubseteq \tilde{\mathcal{H}}'_1}{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}$$

$$\textsc{Widening-Pre}\quad \frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}}_1 \rhd \tilde{\mathcal{H}}' \qquad \phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad \tilde{\mathcal{H}}_1 \sqsubseteq \tilde{\mathcal{H}}}{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}$$

$$\textsc{Call}\quad \frac{\{\ \phi\ \}\ p()\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}' \in \mathit{Specs} \qquad \phi' \models \phi}{\{\ \phi'\ \}\ \texttt{call}\ p()\ \{\ \_\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}'}$$

Evolution Theorem

$$\frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \leadsto \tilde{\mathcal{H}}\ \triangleright\ \tilde{\mathcal{H}}'}{\{\ \phi\ \wedge\ \tilde{\mathcal{H}} * \mathcal{H}_0\ \}\ P\ \{\ \psi\ \wedge\ \tilde{\mathcal{H}}' * \mathcal{H}_0\ \}}$$

Update Enclosure (Our version of Memory Safety)

Suppose that P is of the form $P_1; s; P_2$.

We say $\tilde{\mathcal{H}}$ *encloses* the update s of P if $\{\ \phi\ \}\ P_1\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}}\ \triangleright\ \tilde{\mathcal{H}}'$ holds, and for each model $\mathcal{I}$ of ψ, $x \in dom(\mathcal{I}(\tilde{\mathcal{H}}'))$ holds.

$$T \rightsquigarrow \tilde{\mathcal{H}} \gg P.$$

denotes that $\mathcal{H}$ encloses all the updates of P.

Usage: Heap reality $\mathcal{H} \sqsubseteq \mathcal{M}$ falsified only if program updates a cell in $dom(\mathcal{H})$, or deallocates a cell in $\mathcal{M}$ whose address is also in $dom(\mathcal{H})$.

Rules for Update Enclosure (Basic)

$$\dfrac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad x \in dom(\tilde{\mathcal{H}})}{\{\ \phi\ \}\ \texttt{*x = y}\ \{\ \_\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg \texttt{*x:= y}}\ \text{HEAP-ASSIGN}$$

$$\dfrac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M} \qquad x \in dom(\tilde{\mathcal{H}})}{\{\ \phi\ \}\ \texttt{free(x)}\ \{\ \_\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg \texttt{free(x)}}\ \text{FREE}$$

$$\dfrac{\phi \models \tilde{\mathcal{H}} \sqsubseteq \mathcal{M}}{\{\ \phi\ \}\ s\ \{\ \_\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg s}\ \text{OTHER-STATEMENTS}$$

$$\dfrac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg P \quad \{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \rhd \tilde{\mathcal{H}}' \qquad \{\ \psi\ \}\ Q\ \{\ \gamma\ \} \rightsquigarrow \tilde{\mathcal{H}}' \gg Q}{\{\ \phi\ \}\ P; Q\ \{\ \gamma\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg (P; Q)}\ \text{SEQ-COMPOSITION}$$

Rules for Update Enclosure (Structural)

$$\frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg P \qquad \phi \models \tilde{\mathcal{H}}' \sqsubseteq \mathcal{M}}{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow (\tilde{\mathcal{H}} \cup \tilde{\mathcal{H}}') \gg P}\ \text{WIDENING-PRE}$$

$$\frac{\begin{array}{c}\{\ \phi\ \}\ \mathit{assume}(b); P_1\ \{\ \psi_1\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg (\mathit{assume}(b); P_1) \\ \{\ \phi\ \}\ \mathit{assume}(\neg b); P_2\ \{\ \psi_2\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg (\mathit{assume}(\neg b); P_2)\end{array}}{\{\ \phi\ \}\ P \equiv \textbf{if}\ (b)\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \{\ \psi_1 \vee \psi_2\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg P}\ \text{IF-THEN-ELSE}$$

$$\frac{(\{\ \phi\ \}\ p()\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg \texttt{[p's body]}) \in \mathit{Specs} \qquad \phi' \models \phi}{\{\ \phi'\ \}\ \texttt{call}\ p()\ \{\ \_\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg \texttt{call}\ p()}\ \text{CALL}$$

The New Frame Rule

$$\frac{\{\ \phi\ \}\ P\ \{\ \psi\ \} \rightsquigarrow \tilde{\mathcal{H}} \gg P}{\{\ \phi\ \wedge\ \tilde{\mathcal{H}} * \mathcal{H}_0\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}\ \}\ P\ \{\ \psi\ \wedge\ \mathcal{H}_0 \sqsubseteq \mathcal{M}\ \}}$$

Solves Two Problem Areas

For the first time, we have a systematic method for automatic proof in two settings:

- Summaries
- Structure Sharing

Cyclic Graph (Basic Setup)

Consider a generic predicate which describes a general, possibly cyclic, graph. We assume that each node has exactly two successors "left" and "right". Some key points:

- the subheaps h_1 and h_2 are separate and together house a graph rooted at x and where the "visited" nodes are kept in the *set of values* t.
- t represents a set of locations, "visited" during *previous processing of a predecessor node*. By construction t will be disjoint from $dom(h_1) \cup dom(h_2)$,
- the heap h_1 represents the nodes the left subtree of x that are visited *for the first time* in a left-to-right preorder traversal.
- Similarly, the second heap h_2 represents the nodes the right subtree of x that are visited for the first time.

Cyclic Graph

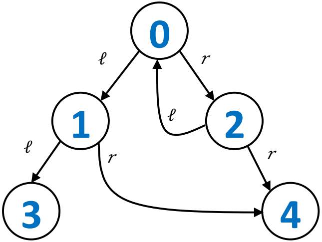
```
graph_root($h_1, h_2, x$) :- graph($h_1, h_2, x, \emptyset$).

graph($h_1, h_2, x, t$) :-
    $h_1 \bumpeq \Omega$, $h_2 \bumpeq \Omega$, $x = \text{null} \vee x \in t$.

graph($h_1, h_2, x, t$) :-
    $h_x \bumpeq (x \mapsto (\_, \mathit{left}, \mathit{right}))$,
    $x \notin t$, $t_1 = t \cup \{x\}$,
    graph($h_{1a}, h_{1b}, \mathit{left}, t_1$), $h_1 \bumpeq h_x * h_{1a} * h_{1b}$,
    $t_2 = t_1 \cup \mathit{dom}(h_{1a}) \cup \mathit{dom}(h_{1b})$
    graph($h_{2a}, h_{2b}, \mathit{right}, t_2$), $h_2 \bumpeq h_{2a} * h_{2b}$.
```

Cyclic Graph

This graph is a model for $\texttt{graph\_root}(h_1, h_2, x)$.
Variable x is node 0. The heap h_1 comprises nodes 0,1,3,4; while h_2 comprises just node 2. Consider $\texttt{graph}(h_{2a}, h_{2b}, \mathit{right}, t_2)$ where *right* is node 2. This is in fact an expression obtained by unfolding $\texttt{graph}(h_1, h_2, x, \emptyset)$. Now h_{2a} comprises just node 2, while $h_{2b} \doteq \Omega$.



Marking a Cyclic Graph

```
struct node {
   int m;
   struct node *left, *right;
};

void mark(struct node *x) {
   if (!x || x->m == 1) return;
   struct node *l = x->left, *r = x->right;
   x->m = 1; mark(l); mark(r);
}
```

Marking a Cyclic Graph

```
mgraph($h_1, h_2, x, t$) :-
    $h_1 \hat{=} \Omega, h_2 \hat{=} \Omega, x = \text{null} \vee x \in t$.
mgraph($h_1, h_2, x, t$) :-  // marked
    $h_x \hat{=} (x \mapsto (1, \mathit{left}, \mathit{right}))$, $x \notin t, t_1 = t \cup \{x\}$,
    mgraph($h_{1a}, h_{1b}, \mathit{left}, t_1$), $h_1 \hat{=} h_x * h_{1a} * h_{1b}$,
    $t_2 = t_1 \cup \mathit{dom}(h_{1a}) \cup \mathit{dom}(h_{1b})$,
    mgraph($h_{2a}, h_{2b}, \mathit{right}, t_2$), $h_2 \hat{=} h_{2a} * h_{2b}$, $h_1 * h_2$.

pmgraph($h_1, h_2, x, t$) :-  mgraph($h_1, h_2, x, t$).
pmgraph($h_1, h_2, x, t$) :-  // unmarked
    $h_x \hat{=} (x \mapsto (0, \mathit{left}, \mathit{right}))$, $x \notin t$, $t_1 = t \cup \{x\}$,
    pmgraph($h_{1a}, h_{1b}, \mathit{left}, t_1$), $h_1 \hat{=} h_x * h_{1a} * h_{1b}$,
    $t_2 = t_1 \cup \mathit{dom}(h_{1a}) \cup \mathit{dom}(h_{1b})$,
    pmgraph($h_{2a}, h_{2b}, \mathit{right}, t_2$), $h_2 \hat{=} h_{2a} * h_{2b}$, $h_1 * h_2$.
```

Marking a Cyclic Graph

requires: $\texttt{pmgraph}(\mathcal{H}_1, \mathcal{H}_2, x, t),\ \mathcal{H}_1 \sqsubseteq \mathcal{M},\ \mathcal{H}_2 \sqsubseteq \mathcal{M}$

ensures: $\texttt{mgraph}(\mathcal{H}'_1, \mathcal{H}'_2, x, t),\ \mathcal{H}'_1 \sqsubseteq \mathcal{M},\ \mathcal{H}'_2 \sqsubseteq \mathcal{M},$
$dom(\mathcal{H}_1) = dom(\mathcal{H}'_1),\ dom(\mathcal{H}_2) = dom(\mathcal{H}'_2)$

frame: $(\mathcal{H}_1 \cup \mathcal{H}_2) \gg \cdot,\ \mathcal{H}_1 \rhd \mathcal{H}'_1,\ \mathcal{H}_2 \rhd \mathcal{H}'_2$

`void mark(struct node *x) {`

$\{\ pmgraph(\mathcal{H}_1, \mathcal{H}_2, x, t),\ \mathcal{H}_1 \sqsubseteq \mathcal{M},\ \mathcal{H}_2 \sqsubseteq \mathcal{M}\ \}$

1 `assume(x && x->m != 1); l = x->left; r = x->right;`

$\{\ \mathcal{H}_x \stackrel{\frown}{=} (x \mapsto (0, l, r)),\ x \notin t,\ t_1 = t \ \cup\ \{x\},\ pmgraph(\mathcal{H}_{1a}, \mathcal{H}_{1b}, l, t_1),$
$\mathcal{H}_1 \stackrel{\frown}{=} \mathcal{H}_x \ * \ \mathcal{H}_{1a} \ * \ \mathcal{H}_{1b},$
$t_2 = t_1 \cup dom(\mathcal{H}_{1a}) \ \cup \ dom(\mathcal{H}_{1b}),\ pmgraph(\mathcal{H}_{2a}, \mathcal{H}_{2b}, r, t_2),$
$\mathcal{H}_2 \stackrel{\frown}{=} \mathcal{H}_{2a} \ * \ \mathcal{H}_{2b},\ \mathcal{H}_1 \ * \ \mathcal{H}_2,\ \mathcal{H}_1 \sqsubseteq \mathcal{M},\ \mathcal{H}_2 \sqsubseteq \mathcal{M}\ \}$

2 `x->m = 1;`

$\{\ pmgraph(\mathcal{H}_{1a}, \mathcal{H}_{1b}, l, t_1),\ \mathcal{H}_{1a} * \mathcal{H}_{1b} * \mathcal{H}_{2a} \ * \ \mathcal{H}_{2b} * (x \mapsto (1, l, r)) \sqsubseteq \mathcal{M},$
$\mathcal{H}_1 \stackrel{\frown}{=} \mathcal{H}_x \ * \ \mathcal{H}_{1a} \ * \ \mathcal{H}_{1b},\ \mathcal{H}_2 \stackrel{\frown}{=} \mathcal{H}_{2a} \ * \ \mathcal{H}_{2b},$
$pmgraph(\mathcal{H}_{2a}, \mathcal{H}_{2b}, r, t_2),\ x \notin t,\ t_1 = t \ \cup\ \{x\},\ t_2 = t_1 \cup dom(\mathcal{H}_{1a}) \ \cup \ dom(\mathcal{H}_{1b})\ \}$

Marking a Cyclic Graph

3 `mark(l);`

$\{\ mgraph(\mathcal{H}'_{1a}, \mathcal{H}'_{1b}, l, t_1),\ \ \mathcal{H}'_{1a} \sqsubseteq \mathcal{M},\ \ \mathcal{H}'_{1b} \sqsubseteq \mathcal{M},$

$dom(\mathcal{H}_{1a}) = dom(\mathcal{H}'_{1a}),\ \ dom(\mathcal{H}_{1b}) = dom(\mathcal{H}'_{1b}),$ `// postcondition`

$\mathcal{H}_1 \hat{=} \mathcal{H}_x * \mathcal{H}_{1a} * \mathcal{H}_{1b},\ \ \mathcal{H}_2 \hat{=} \mathcal{H}_{2a} * \mathcal{H}_{2b},$ `// Rule` (HOARE-FR)

$pmgraph(\mathcal{H}_{2a}, \mathcal{H}_{2b}, r, t_2),$

$x \notin t,\ \ t_1 = t \cup \{x\},\ \ t_2 = t_1 \cup dom(\mathcal{H}_{1a}) \cup dom(\mathcal{H}_{1b}),$

$\mathcal{H}'_{1a} * \mathcal{H}_{1b} * \mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r)),$

$\mathcal{H}'_{1b} * \mathcal{H}_{1a} * \mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r)),$ `// Rule` (EV)

$\mathcal{H}_{2a} * \mathcal{H}_{2b} * (x \mapsto (1, l, r)) \sqsubseteq \mathcal{M}\ \}$ `// Rule` (FR)

4 `mark(r);`

$\{\ mgraph(\mathcal{H}'_{1a}, \mathcal{H}'_{1b}, l, t_1),\ \ dom(\mathcal{H}_{1a}) = dom(\mathcal{H}'_{1a}),\ \ dom(\mathcal{H}_{1b}) = dom(\mathcal{H}'_{1b})$

$x \notin t,\ \ t_1 = t \cup \{x\},\ \ t_2 = t_1 \cup dom(\mathcal{H}_{1a}) \cup dom(\mathcal{H}_{1b}),$

$mgraph(\mathcal{H}'_{2a}, \mathcal{H}'_{2b}, r, t_2),\ \ \mathcal{H}'_{2a} \sqsubseteq \mathcal{M},\ \ \mathcal{H}'_{2b} \sqsubseteq \mathcal{M},$

$dom(\mathcal{H}_{2a}) = dom(\mathcal{H}'_{2a}),\ \ dom(\mathcal{H}_{2b}) = dom(\mathcal{H}'_{2b})$ `// postcondition`

$\mathcal{H}'_{2a} * \mathcal{H}'_{1a} * \mathcal{H}_{1b} * \mathcal{H}_{2b} * (x \mapsto (1, l, r)),$

$\mathcal{H}'_{2b} * \mathcal{H}'_{1b} * \mathcal{H}_{1a} * \mathcal{H}_{2a} * (x \mapsto (1, l, r)),$ `// Rule` (EV)

$\mathcal{H}'_{1a} \sqsubseteq \mathcal{M},\ \ \mathcal{H}'_{1b} \sqsubseteq \mathcal{M},\ \ (x \mapsto (1, l, r)) \sqsubseteq \mathcal{M}\ \}$ `// Rule` (FR)

$\}\ \{\ mgraph(\mathcal{H}'_1, \mathcal{H}'_2, x, t),\ \ \mathcal{H}'_1 \sqsubseteq \mathcal{M},\ \ \mathcal{H}'_2 \sqsubseteq \mathcal{M},$

$dom(\mathcal{H}_1) = dom(\mathcal{H}'_1),\ \ dom(\mathcal{H}_2) = dom(\mathcal{H}'_2)\ \}$

Conclusion

- Expressive assertion language for dynamic data structures
- Strongest Postcondition semantics
- Automatic Induction for a Class of VC's
- New Frame Rule for Local Reasoning / Compositional Proofs
- All the above in (regular) Hoare Logic

Some References

- HIP/SLEEK
  W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In SCP, 10061036, 2012
- DRYAD
  X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In PLDI, pages 231242, 2013
- $\mathcal{H}$ Language
  G. Duck, J. Jaffar, and N. Koh. A constraint solver for heaps with separation. In CP, pages 282-298, 2013
- Auto Induction
  D. H. Chu, J. Jaffar, and M. T. Trinh, Automatic induction proofs of data-structures in imperative programs, In PLDI, 2015
- New Frame Rule
  D.H. Chu and J. Jaffar, Local Reasoning with First-Class Heaps and a new Frame Rule, draft, `www.comp.nus.edu.sg/∼joxan/papers/frame.pdf`