Interactive Software Verification

Spring Term 2013

Holger Gast
gasth@in.tum.de

9.7.2013

Separation Logic So Far

- Motivation: Short-comings of Burstall's memory model
  - No byte-addressed memory (low-level C programs)
  - Disjointness assertions cumbersome
  - Disjointness proofs time-consuming
- Separation Logic
  - Express heap layout syntactically
  - $\Rightarrow$ Aliasing/disjointness clear from syntax
  - Manipulate heap formulae algebraically
- So far: basic notions & definitions

Today: A Separation Logic Framework

- Symbolic Execution — the key to success [2, 11, 4, 5]

⇒ Develop framework within Isabelle

- Re-use Simpl-embedding [6]

Recap

Recap: Heaps for Separation Logic

- Keep memory in global variable `heap`

  $\textbf{datatype}\ \mathsf{addr} = \mathsf{Addr\ nat}$
  $\textbf{datatype}\ \mathsf{heap} = \mathsf{Heap}\ \text{"}\mathsf{addr} \rightharpoonup \mathsf{heap\text{-}val}\text{"}$

- Establish heap access layer

$$\mathsf{heap\text{-}get}\ \mathsf{ct}\ \mathsf{p}\ \mathsf{h} \equiv \mathsf{ty\text{-}val}\ \mathsf{ct}\ (\mathsf{map}\ (\mathsf{the} \circ \mathsf{hcnt}\ \mathsf{h})\ (\mathsf{addr\text{-}ivl}\ \mathsf{p}\ (\mathsf{ty\text{-}size}\ \mathsf{ct})))$$

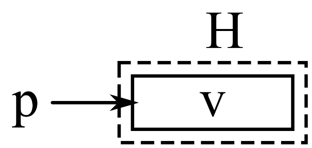
$$\begin{aligned} &\mathsf{heap\text{-}put}\ \mathsf{ct}\ \mathsf{p}\ \mathsf{v}\ \mathsf{h} \equiv \\ &\quad \mathsf{Heap}\ (\mathsf{hcnt}\ \mathsf{h} ++ (\lambda \mathsf{q}.\ \mathsf{if}\ \mathsf{q} \in \mathsf{set}\ (\mathsf{addr\text{-}ivl}\ \mathsf{p}\ (\mathsf{ty\text{-}size}\ \mathsf{ct})) \\ &\qquad\qquad\qquad\qquad \mathsf{then}\ \mathsf{Some}\ (\mathsf{nth}\ (\mathsf{ty\text{-}rep}\ \mathsf{ct}\ \mathsf{v})\ (\mathsf{q} \ominus \mathsf{p})) \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{None})) \end{aligned}$$

$$\begin{aligned} &\mathsf{heap\text{-}acc\text{-}cond}\ \mathsf{ct}\ \mathsf{p} \equiv \\ &\quad \lambda \mathsf{h}.\ \mathsf{set}\ (\mathsf{addr\text{-}ivl}\ \mathsf{p}\ (\mathsf{ty\text{-}size}\ \mathsf{ct})) \subseteq \mathsf{hdom}\ \mathsf{h} \end{aligned}$$

- Parameterized by type descriptors: representation of values

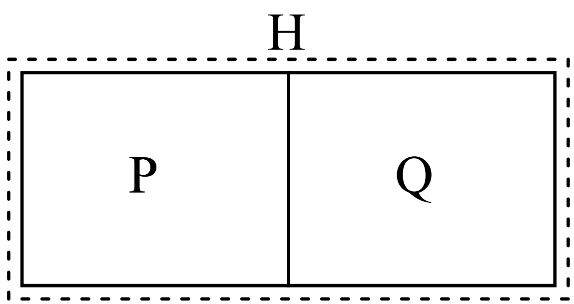
Recap: Spatial Assertions

- Points-to: values in the heap



$$\mathsf{p} \mapsto \mathsf{ct}.\,\mathsf{v} \equiv$$
$$\lambda \mathsf{h}.\ \mathsf{hdom}\ \mathsf{h} = \mathsf{set}\,(\mathsf{addr\text{-}ivl}\ \mathsf{p}\,(\mathsf{ty\text{-}size}\ \mathsf{ct})) \wedge \mathbf{heap\text{-}get\ ct\ p\ h = v} \wedge \mathsf{p} \neq \mathsf{NULL}$$

- Star: disjoint parts of the heap



$$\mathsf{P} \star \mathsf{Q} \equiv \lambda \mathsf{h0}.\ \exists\, \mathsf{h}\ \mathsf{h'}.\ \mathsf{h} \perp \mathsf{h'} \wedge \mathsf{h0} = \mathsf{h} \uplus \mathsf{h'} \wedge \mathbf{P\ h} \wedge \mathbf{Q\ h'}$$

- Core idea: tight specifications

Extension: Records

- C Structs

  define-struct
  "struct node { int data; struct node *next; }"

- HOL value is an Isabelle record

  $(\!|\, \mathsf{node\text{-}data} = \mathsf{data},\ \mathsf{node\text{-}next} = \mathsf{next} \,|\!)$

- Introduce constants for field offsets (`fo-`...)
- Define spatial abstraction predicate

$$\begin{aligned}\mathsf{node}\ \mathsf{p}\ \mathsf{v} \equiv \mathcal{E}\,\mathsf{data}\ \mathsf{next}.\ & \mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}data} \mapsto \mathsf{cty\text{-}int}.\ \mathsf{data} \star \\ & \mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}next} \mapsto \mathsf{cty\text{-}ptr}.\ \mathsf{next} \bullet \\ & \mathsf{v} = (\!|\, \mathsf{node\text{-}data} = \mathsf{data},\ \mathsf{node\text{-}next} = \mathsf{next} \,|\!)\end{aligned}$$

⇒ Simple notation for records on the heap

Symbolic Execution

Symbolic Execution: the Core Idea

- Goal: automatic verification in separation logic [2, 11, 4, 5]
- Hoare rule for forward reasoning

$$\{\, p \mapsto - \star Q \,\} \; *p := e \; \{\, p \mapsto e \star Q \,\}$$

- Procedural interpretation
  - Re-arrange pre-condition by syntactic manipulations
  - Match against pre-condition of rule

  $\Rightarrow$ Fills Q
  - Replace Q and e in post-condition

$\Rightarrow$ Compute post-condition from pre-condition

- Local variables: treat as in Simpl

Symbolic Heaps

- General manipulation of sep-logic formulae hard [8, 10]
- Specifications of most programs fall into fragment

$$\exists x_1 \ldots x_n.\, P_1 \star \cdots \star P_m \wedge (Q_1 \wedge \cdots Q_r)$$

with spatial assertions P_i and pure assertions Q_j

- Intuition
  - spatial part used to "read" values from heap
  - pure part specifies relationships between values

⇒ Term: symbolic heap

- In Isabelle (because other symbols are taken)

$$\mathcal{E} x_1 \ldots x_n.\, \mathsf{P}_1 \star \ldots \star \mathsf{P}_m \cdot \mathsf{Q}_1 \wedge \ldots \wedge \mathsf{Q}_r$$

- As premise: $\bigwedge x_1 \ldots x_n.\, [\![\, (\mathsf{P}_1 \star \ldots \star \mathsf{P}_m)\, \mathsf{heap}; \mathsf{Q}_1; \ldots\ \mathsf{Q}_r \,]\!] \Longrightarrow \ldots$

Towards Symbolic Execution in Simpl

- Consider example

```
*p = 1;
r = *q;
*q = 2;
```

- Simpl generates

$$
\begin{array}{l}
(\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\, X \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\, Y)\ \mathsf{heap} \Longrightarrow \\
\quad \mathsf{heap\text{-}acc\text{-}cond}\ \mathsf{cty\text{-}int}\ \mathsf{p}\ \mathsf{heap} \wedge \\
\quad \mathsf{heap\text{-}acc\text{-}cond}\ \mathsf{cty\text{-}int}\ \mathsf{q}\ (\mathsf{heap\text{-}put}\ \mathsf{cty\text{-}int}\ \mathsf{p}\ 1\ \mathsf{heap}) \wedge \\
\quad (\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\, 1 \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\, 2 \bullet \\
\quad\ \mathsf{heap\text{-}get}\ \mathsf{cty\text{-}int}\ \mathsf{q}\ (\mathsf{heap\text{-}put}\ \mathsf{cty\text{-}int}\ \mathsf{p}\ 1\ \mathsf{heap}) = Y) \\
\quad\ (\mathsf{heap\text{-}put}\ \mathsf{cty\text{-}int}\ \mathsf{q}\ 2\ (\mathsf{heap\text{-}put}\ \mathsf{cty\text{-}int}\ \mathsf{p}\ 1\ \mathsf{heap}))
\end{array}
$$

$\Rightarrow$ Where is the "execution" in this mess?

Generated Verification Conditions

- Recall side-effects: `a[i] = 42;` with post: $\mathsf{aget\ a\ i = 42}$

  $[\![0 \le \mathsf{i}; \mathsf{i} < \mathsf{alength\ a}]\!] \Longrightarrow \mathsf{acheck\ i\ a} \wedge \textbf{aget (aset i 42 a)}\ \mathsf{i} = 42$

- Analogous: `*p = 1; r = *q; *q = 2;`

  $(\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\, 1 \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\, 2 \cdot$
  $\mathsf{heap\text{-}get\ cty\text{-}int\ q}\ \textbf{(heap-put cty-int p 1 heap)} = \mathsf{Y})$
  $\textbf{(heap-put cty-int q 2 (heap-put cty-int p 1 heap))}$

$\Rightarrow$ Obtain assertion about "changed heap", as usual

- Central insight [6]

  The nesting of the heap access conditions reflects the original execution order.

Re-constructing the Execution Order

- Consider again `*p = 1; *r = *q;`

  heap-get cty-int q (heap-put cty-int p 1 heap)

- Plan of method `prep_vc` [6]
  - Name all heap accesses (get & put)

  ⇒ Have names for all intermediate heap states
  - Build dependency graph
    - Nested accesses executed before outer accesses
    - `heap-put` after `heap-get` on same heap state
  - Introduce let-bindings according to dependencies

⇒ Let-binding reflect execution order [1, 7, 9]

- Technical detail: use HeapOp ≡ Let as markup

Example

- Code `*p = 1; r = *q; *q = 2;`
- Result

$$
\begin{array}{l}
\text{heap-acc-cond cty-int p heap} \wedge \\
\quad (\textbf{let}\ \text{H-1} = \text{heap-put cty-int p 1 heap} \\
\quad\ \textbf{in}\ \text{heap-acc-cond cty-int q H-1} \wedge \\
\quad\quad (\textbf{let}\ \text{H-2} = \text{heap-get cty-int q H-1}; \\
\quad\quad\quad\quad \text{H-3} = \text{heap-put cty-int q 2 H-1} \\
\quad\quad\ \textbf{in}\ (\text{p} \mapsto \text{cty-int}.\ 1 \star \text{q} \mapsto \text{cty-int}.\ 2 \cdot \text{H-2} = \text{Y})\ \text{H-3}))
\end{array}
$$

- Remark: technically

$$
\begin{array}{l}
\text{heap-acc-cond cty-int p heap} \wedge \\
\quad (\text{heapop H-1} \leftarrow \text{heap-put cty-int p 1 heap}; \\
\quad\ \text{heap-acc-cond cty-int q H-1} \wedge \\
(\text{heapop H-2} \leftarrow \text{heap-get cty-int q H-1}; \\
\quad\quad \text{heapop H-3} \leftarrow \text{heap-put cty-int q 2 H-1}; \\
\quad\quad (\text{p} \mapsto \text{cty-int}.\ 1 \star \text{q} \mapsto \text{cty-int}.\ 2 \cdot \text{H-2} = \text{Y})\ \text{H-3}))
\end{array}
$$

Symbolic Execution in Goal Format

- Idea: current state reflected in goal's premises
- Example

$$
\begin{aligned}
&(\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\,\mathsf{X} \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\,\mathsf{Y})\ \mathsf{heap} \Longrightarrow \\
&\quad \mathsf{heap\text{-}acc\text{-}cond\ cty\text{-}int\ p\ heap} \wedge \\
&\quad (\mathsf{heapop\ H\text{-}1} \leftarrow \mathbf{heap\text{-}put\ cty\text{-}int\ p\ 1}\ \mathsf{heap}; \\
&\quad\ \mathsf{heap\text{-}acc\text{-}cond\ cty\text{-}int\ q\ H\text{-}1} \wedge
\end{aligned}
$$

- Next step

$$
\begin{aligned}
&(\mathbf{p} \mapsto \mathbf{cty\text{-}int.\,1} \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\,\mathsf{Y})\ \mathsf{h} \Longrightarrow \\
&\quad \mathsf{heap\text{-}acc\text{-}cond\ cty\text{-}int\ q\ h} \wedge \\
&\quad (\mathsf{heapop\ H\text{-}2} \leftarrow \mathbf{heap\text{-}get\ cty\text{-}int\ q}\ \mathsf{h}; \\
&\quad\ \mathsf{heapop\ H\text{-}3} \leftarrow \mathsf{heap\text{-}put\ cty\text{-}int\ q\ 2\ h}; \\
&\quad\ (\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\,1 \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\,2 \cdot \mathsf{H\text{-}2} = \mathsf{Y})\ \mathsf{H\text{-}3})
\end{aligned}
$$

- Next

$$
\begin{aligned}
&(\mathsf{q} \mapsto \mathsf{cty\text{-}int}.\,\mathsf{Y} \star \mathsf{p} \mapsto \mathsf{cty\text{-}int}.\,1)\ \mathsf{h} \Longrightarrow \\
&\quad \mathsf{HeapOp}\ (\mathbf{heap\text{-}put\ cty\text{-}int\ q\ 2}\ \mathsf{h}) \\
&\quad\ (\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\,1 \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\,2 \cdot \mathbf{Y} = \mathsf{Y})
\end{aligned}
$$

Handling Heap-Puts

- Recall classical rule

$$\{\, p \mapsto - \star Q \,\} \; *p := e \; \{\, p \mapsto e \star Q \,\}$$

- In the case of goals: replace heap premise with different one

$$\begin{array}{l} \llbracket\, (\mathsf{p} \mapsto \mathsf{ct}.\,\mathsf{a} \star \mathsf{H})\; \mathsf{h}; \\ \quad \mathsf{ty\text{-}ok}\; \mathsf{ct}; \\ \quad \bigwedge \mathsf{h}.\, (\mathsf{p} \mapsto \mathsf{ct}.\,\mathsf{v} \star \mathsf{H})\; \mathsf{h} \Longrightarrow \mathsf{Q}\; \mathsf{h} \\ \quad \rrbracket \Longrightarrow \mathsf{HeapOp}\; (\mathsf{heap\text{-}put}\; \mathsf{ct}\; \mathsf{p}\; \mathsf{v}\; \mathsf{h})\; \mathsf{Q} \end{array}$$

  - Match first premise against current heap
  - Side-condition: type is well-formed
  - Result: new heap `h` with new assertion
- Auxiliary tactic: drop old heap assertion

$\Rightarrow$ Effect: “in-place” update of heap assertion in current goal

Handling Heap-Get

- Situation

$$(\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\, 1 \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{Y})\, \mathsf{h} \Longrightarrow$$
$$(\mathsf{heapop\text{-}H\text{-}2} \leftarrow \mathsf{heap\text{-}get}\; \mathsf{cty\text{-}int}\; \mathsf{q}\; \mathsf{h}; \cdots)$$

- Basic insight: lookup "the value" at p

$$(\mathsf{p} \mapsto \mathsf{ct}.\, \mathsf{v} \star \mathsf{R})\, \mathsf{h} \Longrightarrow \mathsf{heap\text{-}get}\; \mathsf{ct}\; \mathsf{p}\; \mathsf{h} = \mathsf{v}$$

- Yields

$$(\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\, 1 \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{Y})\, \mathsf{h} \Longrightarrow$$
$$(\mathsf{heapop\text{-}H\text{-}2} \leftarrow \mathsf{Y} \cdots)$$

- Then unfold the let-binding ⇨ syntactic replacement

Handling Allocatedness

- Situation

$$(\mathsf{p} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{X} \star \mathsf{q} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{Y})\, \mathsf{heap} \Longrightarrow \\ \quad \mathsf{heap\text{-}acc\text{-}cond}\, \mathsf{cty\text{-}int}\, \mathsf{p}\, \mathsf{heap} \wedge \cdots$$

- Rule: points-to includes side-condition on allocatedness

$$(\mathsf{p} \mapsto \mathsf{ct}.\, \mathsf{a} \star \mathsf{H})\, \mathsf{h} \Longrightarrow \mathsf{heap\text{-}acc\text{-}cond}\, \mathsf{ct}\, \mathsf{p}\, \mathsf{h}$$

⇒ Simply apply rule to prove side-condition

Re-arranging Symbolic Heaps

- All three rules have premise $\mathsf{p} \mapsto \mathsf{ct.\,a} \star \mathsf{H}$
- Re-arrange compartments of current heap to match
- Star $\star$ is associative & commutative ⇨ order irrelevant
- Likewise: heap-matching in final proof obligation
- In example:

$$(\mathsf{q} \mapsto \mathsf{cty\text{-}int.\,2} \star \mathsf{p} \mapsto \mathsf{cty\text{-}int.\,1})\,\mathsf{ha} \Longrightarrow$$
$$(\mathsf{p} \mapsto \mathsf{cty\text{-}int.\,1} \star \mathsf{q} \mapsto \mathsf{cty\text{-}int.\,2})\,\mathsf{ha}$$

$\Rightarrow$ Method `heap`

Automatic Unfolding

- Next up: sub-structures & unfolding

  define-struct
  "struct node { int data; struct node *next; }"

- Example: $\mathsf{r = p \rightarrow data;}$

  $\mathbf{node\ p\ v\ heap} \Longrightarrow$
  $\quad \mathsf{heap\text{-}acc\text{-}cond\ cty\text{-}int\ (}\mathbf{p \oplus fo\text{-}node\text{-}data}\mathsf{)\ heap} \wedge$
  $\quad \mathsf{node\ p\ v\ heap} \wedge$
  $\quad \mathsf{(heapop\ H\text{-}1 \leftarrow heap\text{-}get\ cty\text{-}int\ (p \oplus fo\text{-}node\text{-}data)\ heap;}$
  $\quad\ \mathsf{H\text{-}1 = node\text{-}data\ v)}$

- Heap matching applies unfolding rules automatically [3, 2]
- Example

  $\mathsf{node\ p\ v =}$
  $\mathsf{(\mathcal{E}\,data\ next.}$
  $\quad \mathsf{p \oplus fo\text{-}node\text{-}data \mapsto cty\text{-}int.\ data \star p \oplus fo\text{-}node\text{-}next \mapsto cty\text{-}ptr.\ next \bullet}$
  $\quad \mathsf{v = (\!|\, node\text{-}data = data,\ node\text{-}next = next\, |\!))}$

Example Unfolding

- Example `r = p->data;` requires $\mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}data} \mapsto \ldots$

⇒ Heap matching applies rules and leaves

$$
\begin{array}{l}
[\![\mathsf{v} = (\!| \mathsf{node\text{-}data} = \mathsf{data}, \mathsf{node\text{-}next} = \mathsf{next} |\!);\\
\quad (\mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}data} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{data} \star\\
\quad\ \mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}next} \mapsto \mathsf{cty\text{-}ptr}.\, \mathsf{next})\\
\quad\ \mathsf{heap}\\
]\!] \Longrightarrow \mathsf{data} = \mathsf{node\text{-}data}\, \mathsf{v}
\end{array}
$$

- Post-condition requires "re-assembled" heap

$$
\begin{array}{l}
(\mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}data} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{data} \star\\
\ \mathsf{p} \oplus \mathsf{fo\text{-}node\text{-}next} \mapsto \mathsf{cty\text{-}ptr}.\, \mathsf{next})\ \mathsf{heap}\\
\Longrightarrow \mathsf{node}\, \mathsf{p}\ (\!| \mathsf{node\text{-}data} = \mathsf{data}, \mathsf{node\text{-}next} = \mathsf{next} |\!)\ \mathsf{heap}
\end{array}
$$

⇒ Apply unfolding rules "backwards" in `heap` tactic

Detail: Normalizing the Symbolic Heap

- Application of unfolding rules leaves $\mathcal{E}, \cdot$ within heap

$\Rightarrow$ Apply rewriting rules to "pull out" these symbols

$$"\bigwedge P\,Q.\,(\mathcal{E}\,x.\,P\,x) \star Q = (\mathcal{E}\,x.\,P\,x \star Q)"$$
$$"\bigwedge P\,Q.\,P \star (\mathcal{E}\,x.\,Q\,x) = (\mathcal{E}\,x.\,P \star Q\,x)"$$
$$"\bigwedge P\,Q.\,[[\,\mathcal{E}\,x.\,P\,x]] = (\exists\,x.\,[[\,P\,x]])"$$
$$"\bigwedge P\,Q\,c.\,(P \cdot c) \star Q = ((P \star Q) \cdot c)"$$
$$"\bigwedge P\,Q\,c.\,P \star (Q \cdot c) = ((P \star Q) \cdot c)"$$
$$"\bigwedge P\,Q.\,[[\,P \cdot c]] = ([[\,P\,]] \wedge c)"$$

- . . . and to make spatial fragment a linear list

$$"\bigwedge P\,Q\,R.\,(P \star Q) \star R = (P \star (Q \star R))"$$

$\Rightarrow$ Heap is always in very simple form, ready for matching

Summary: Verification by Symbolic Execution

- Use spatial operations in assertions
- Apply Simpl `vcg` to generate verification condition
- Apply `prep_vc` to re-construct execution order
- Apply `step`/`run` for symbolic execution
- Match resulting heap against post-condition with `heap`
- Prove pure side-conditions on heap values

Lists in Separation Logic

- Recursive definition: enumerate nodes

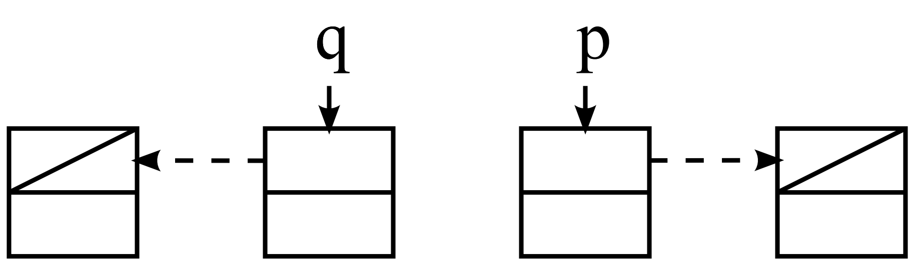
  $$\textbf{fun}\ \mathsf{list} :: \text{''}\mathsf{addr} \Rightarrow \mathsf{int\ list} \Rightarrow \mathsf{addr} \Rightarrow \mathsf{hassn}\text{''}$$
  $$\textbf{where}$$
  $$\text{''}\mathsf{list\ p}\ []\ \mathsf{q} = (\mathsf{emp} \cdot (\mathsf{p} = \mathsf{q}))\text{''}$$
  $$|\ \text{''}\mathsf{list\ p}\ (\mathsf{x} \# \mathsf{xs'})\ \mathsf{q} = (\mathcal{E}\ \mathsf{v.\ node\ p\ v} \star \mathsf{list}\ (\mathsf{node\text{-}next\ v})\ \mathsf{xs'}\ \mathsf{q} \cdot \mathsf{x} = \mathsf{node\text{-}data\ v})\text{''}$$

- Many programs access first node ⇨ auto-unfold

  $$\textbf{lemma}\ \mathsf{list\text{-}step[sep\text{-}unfold]}:$$
  $$\text{''}\mathsf{p} \neq \mathsf{q} \Longrightarrow \mathsf{list\ p\ xs\ q} = (\mathcal{E}\ \mathsf{v\ xs'}.\ \mathsf{node\ p\ v} \star (\mathsf{list}\ (\mathsf{node\text{-}next\ v})\ \mathsf{xs'}\ \mathsf{q}) \cdot (\mathsf{xs} = \mathsf{node\text{-}data\ v} \# \mathsf{xs'}))\text{''}$$

Example: List-Reversal

```
pre: "(list p XS NULL) heap"
post: "(list q (rev XS) NULL) heap"
"q = null;
 /*@
  (ℰ xs ys. (list p xs NULL ⋆ list q ys NULL) • XS = (rev ys @ xs)) heap
  */
 while (p != null) {
  t = p->next;
  p->next = q;
  q = p;
  p = t;
 }"
```



Proof of List-Reversal

```
apply vcg
apply simp-all
apply prep-vc
apply heap
apply run
apply simp
apply heap
apply simp
apply run
apply simp
apply heap
apply simp
done
```

- heap instantiates quantifiers

⇒ No need for **apply** (rule-tac x="..." in exI)

Variant with Ghost-Variables

```
q = null;
 //@ xs = XS
 //@ ys = []
 /*@
  (list p xs NULL ⋆ list q ys NULL · (XS = (rev ys @ xs))) heap
 */
 while (p != null) {
  t = p->next;
  p->next = q;
  q = p;
  p = t;
  //@ ys = hd xs # ys
  //@ xs = tl xs
 }
```

- Proof: similar
- Advantage: less proof search in `heap`

Summary

- Tool for symbolic execution in separation logic
  - Target: heap-manipulating programs
  - Mostly automatic proofs
  - Disjointness considerations solved syntactically
  - Beyond "records on heap" memory model
- Techniques
  - Symbolic heaps & heap matching
  - Automatic unfolding
  - Solving heap-related proof obligations
  - Reconstruction of execution order from Simpl VC

References

[1] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference*, volume 4732 of *LNCS*. Springer, 2007.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium (FMCO)*, volume 4111 of *LNCS*. Springer, 2005.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium (APLAS)*, volume 3780 of *LNCS*. Springer, 2005.

[4] Matko Botincan, Matthew Parkinson, and Wolfram Schulte. Separation logic verification of c programs with an smt solver. In *4th International Workshop on Systems Software Verification (SSV)*, volume 254. Elsevier, 2009.

[5] Dino Distefano and Matthew J. Parkinson J. jStar: towards practical verification for Java. *SIGPLAN Not.*, 43(10):213–226, 2008.

[6] Holger Gast. Semi-automatic proofs about object graphs in separation logic. In Stephan Merz and Gerald Lüttgen, editors, *Automated Verification of Critical Systems (AVoCS '12)*, 2012.

[7] Christoph Lüth and Dennis Walter. Certifiable specification and verification of C programs. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress*, volume 5850 of *LNCS*. Springer, 2009.

[8] Andrew McCreight. *The Mechanized Verification of Garbage Collector Implementations*. PhD thesis, Department of Computer Science, Yale University, December 2008.

[9] Andrew McCreight. Practical tactics for separation logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs)*, volume 5674 of *LNCS*. Springer, 2009.

[10] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, 2007.

[11] Thomas Tuerk. A formalisation of Smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs)*, volume 5674 of *LNCS*. Springer, 2009.