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
    ⇒ 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 \([2, 11, 4, 5]\)

\Rightarrow \text{Develop framework within Isabelle}

- Re-use Simpl-embedding \([6]\)
Recap
Recap: Heaps for Separation Logic

- Keep memory in global variable heap
  
  \textbf{datatype} addr = Addr \text{ nat} \\
  \textbf{datatype} heap = Heap \text{"addr} \mapsto \text{heap-val"}

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

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

  heap-acc-cond ct p \equiv
  
  \lambda h. \text{set} (addr-ivl p (ty-size ct)) \subseteq \text{hdom h}

- Parameterized by \textbf{type descriptors}: representation of values
Recap: Spatial Assertions

- Points-to: values in the heap

\[ p \mapsto \text{ct. } v \equiv \lambda h. \text{hdom } h = \text{set (addr-ivl } p \text{ (ty-size ct))} \land \text{heap-get ct p h } = v \land p \neq \text{NULL} \]

- Star: disjoint parts of the heap

\[ P \star Q \equiv \lambda h_0. \exists h \ h'. h \perp h' \land h_0 = h \cup h' \land P h \land Q h' \]

- Core idea: tight specifications
Extension: Records

- C Structs
  ```c
  define-struct
  "struct node { int data; struct node *next; }"
  ```
- HOL value is an Isabelle record
  
  \[
  (\text{node-data} = \text{data}, \text{node-next} = \text{next})
  \]
- Introduce constants for field offsets (fo- . . .)
- Define spatial abstraction predicate
  
  \[
  \text{node}\ p\ v \equiv E\ \text{data}\ \text{next}.\ p \oplus \text{fo-node-data} \mapsto \text{cty-int. data} \ast
  
  p \oplus \text{fo-node-next} \mapsto \text{cty-ptr. next} \cdot
  
  v = (\text{node-data} = \text{data}, \text{node-next} = \text{next})
  \]

⇒ 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
  \[
  \{ p \mapsto - \ast Q \} \ast p := e \{ p \mapsto e \ast Q \}
  \]
- 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

\[ \exists x_1 \ldots x_n. \ P_1 \star \cdots \star P_m \land (Q_1 \land \cdots Q_r) \]

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)

\[ \mathcal{E} x_1 \ldots x_n. P_1 \star \cdots \star P_m \cdot Q_1 \land \cdots \land Q_r \]

- As premise: \( \land x_1 \ldots x_n. \left[ (P_1 \star \cdots \star P_m) \text{heap}; Q_1; \ldots; Q_r \right] \Rightarrow \ldots \)
Towards Symbolic Execution in SimpI

• Consider example

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

• SimpI generates

\( (p \mapsto \text{cty-int.} \, X \, \star \, q \mapsto \text{cty-int.} \, Y) \, \text{heap} \implies \)

\( \text{heap-acc-cond} \, \text{cty-int} \, p \, \text{heap} \land \)
\( \text{heap-acc-cond} \, \text{cty-int} \, q \, (\text{heap-put} \, \text{cty-int} \, p \, 1 \, \text{heap}) \land \)
\( (p \mapsto \text{cty-int.} \, 1 \, \star \, q \mapsto \text{cty-int.} \, 2 \cdot \)
\( \text{heap-get} \, \text{cty-int} \, q \, (\text{heap-put} \, \text{cty-int} \, p \, 1 \, \text{heap}) = Y) \)
\( (\text{heap-put} \, \text{cty-int} \, q \, 2 \, (\text{heap-put} \, \text{cty-int} \, p \, 1 \, \text{heap})) \)

\( \implies \) Where is the “execution” in this mess?
Generated Verification Conditions

- Recall side-effects: \( a[i] = 42 \); with post: \( aget a i = 42 \)
  
  \[ 0 \leq i; i < \text{alength} a \] \( \implies \) \( \text{acheck} i a \wedge \text{aget} (\text{aset} i 42 a) i = 42 \)

- Analogous: \( *p = 1; r = *q; *q = 2; \)
  
  \((p \mapsto \text{cty-int.} 1 \star q \mapsto \text{cty-int.} 2 \cdot \\
  \text{heap-get cty-int} q (\text{heap-put cty-int} p 1 \text{heap}) = Y) \)
  
  \((\text{heap-put cty-int} q 2 (\text{heap-put cty-int} p 1 \text{heap})) \)

\[ \implies \text{Obtain assertion about “changed heap”, as usual} \]

- Central insight [6]
  
  The nesting of the heap access conditions reflects the original execution order.
Re-constructing the Execution Order

- Consider again \( *p = 1; *r = *q; \)
  
  \[
  \text{heap-get cty-int } q \ (\text{heap-put cty-int } p \ 1 \ \text{heap})
  \]

- Plan of method prep_vc [6]
  - Name all heap accesses (get & put)
  \( \Rightarrow \) 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
  \( \Rightarrow \) Let-binding reflect execution order [1, 7, 9]

- Technical detail: use HeapOp \( \equiv \text{Let} \) as markup
Example

- Code \( *p = 1; \ r = *q; \ *q = 2; \)

- Result

\[
\begin{align*}
&\text{heap-acc-cond cty-int } p \text{ heap } \land \\
&\quad (\text{let } H-1 = \text{heap-put cty-int } p \ 1 \text{ heap} \\
&\quad \text{in heap-acc-cond cty-int } q \ H-1 \land \\
&\quad (\text{let } H-2 = \text{heap-get cty-int } q \ H-1; \\
&\quad \quad H-3 = \text{heap-put cty-int } q \ 2 \ H-1 \\
&\quad \quad \text{in } (p \mapsto \text{cty-int.} \ 1 \star q \mapsto \text{cty-int.} \ 2 \cdot H-2 = Y) \ H-3))
\end{align*}
\]

- Remark: technically

\[
\begin{align*}
&\text{heap-acc-cond cty-int } p \text{ heap } \land \\
&\quad (\text{heapop } H-1 \leftarrow \text{heap-put cty-int } p \ 1 \text{ heap}; \\
&\quad \text{heap-acc-cond cty-int } q \ H-1 \land \\
&\quad (\text{heapop } H-2 \leftarrow \text{heap-get cty-int } q \ H-1; \\
&\quad \quad \text{heapop } H-3 \leftarrow \text{heap-put cty-int } q \ 2 \ H-1; \\
&\quad \quad (p \mapsto \text{cty-int.} \ 1 \star q \mapsto \text{cty-int.} \ 2 \cdot H-2 = Y) \ H-3))
\end{align*}
\]
Symbolic Execution in Goal Format

• Idea: current state reflected in goal’s premises

• Example

\[(p \mapsto cty\text{-}int. X \star q \mapsto cty\text{-}int. Y) \text{ heap} \implies\]
heap-acc-cond cty\text{-}int p heap \land

(\text{heapop } H-1 \leftarrow \textbf{heap\text{-}put } cty\text{-}int p 1 \text{ heap};
heap-acc-cond cty\text{-}int q H-1 \land

• Next step

\[(p \mapsto \textbf{cty\text{-}int. 1} \star q \mapsto cty\text{-}int. Y) h \implies\]
heap-acc-cond cty\text{-}int q h \land

(\text{heapop } H-2 \leftarrow \textbf{heap\text{-}get } cty\text{-}int q h;
heapop H-3 \leftarrow \text{heap\text{-}put } cty\text{-}int q 2 h;
(p \mapsto cty\text{-}int. 1 \star q \mapsto cty\text{-}int. 2 \cdot H-2 = Y) H-3)\]

• Next

\[(q \mapsto cty\text{-}int. Y \star p \mapsto cty\text{-}int. 1) h \implies\]
\text{HeapOp (heap\text{-}put } cty\text{-}int q 2 h)
(p \mapsto cty\text{-}int. 1 \star q \mapsto cty\text{-}int. 2 \cdot Y = Y)\]
Handling Heap-Puts

• Recall classical rule

\[ \{ p \mapsto - \ast Q \} \ast p := e \{ p \mapsto e \ast Q \} \]

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

\[
\begin{align*}
\llbracket (p \mapsto \text{ct. } a \ast H) h; \\
\quad \text{ty-ok } \text{ct}; \\
\quad \land h. (p \mapsto \text{ct. } v \ast H) h \implies Q h \\
\rrbracket \implies &\text{HeapOp (heap-put ct p v h) Q}
\end{align*}
\]

• 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
  \[(p \mapsto \text{cty-int. } 1 \star q \mapsto \text{cty-int. } Y) \implies (\text{heapop-H-2} \leftarrow \text{heap-get cty-int } q \ h; \cdots)\]

- Basic insight: lookup “the value” at \(p\)
  \[(p \mapsto \text{ct. } v \star R) \implies \text{heap-get ct } p \ h = v\]

- Yields
  \[(p \mapsto \text{cty-int. } 1 \star q \mapsto \text{cty-int. } Y) \implies (\text{heapop-H-2} \leftarrow Y \cdots)\]

- Then unfold the let-binding ⇨ syntactic replacement
Handling Allocatedness

• Situation

\[(p \mapsto \text{cty-int.} \ X \ \star \ q \mapsto \text{cty-int.} \ Y) \ heap \implies \]
\[\text{heap-acc-cond cty-int p heap \land \cdots} \]

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

\[(p \mapsto \text{ct.} \ a \ \star \ H) \ h \implies \text{heap-acc-cond ct p h} \]

⇒ Simply apply rule to prove side-condition
Re-arranging Symbolic Heaps

- All three rules have premise \( p \mapsto \text{ct. a} \star H \)
- Re-arrange compartments of current heap to match
- Star \( \star \) is associative & commutative \( \Rightarrow \) order irrelevant
- Likewise: heap-matching in final proof obligation
- In example:
  
  \[
  (q \mapsto \text{cty-int. 2} \star p \mapsto \text{cty-int. 1}) \text{ ha} \Rightarrow
  (p \mapsto \text{cty-int. 1} \star q \mapsto \text{cty-int. 2}) \text{ ha}
  \]

\( \Rightarrow \) Method heap
Automatic Unfolding

• Next up: sub-structures & unfolding

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

• Example: \( r = p \rightarrow data; \)

  \[
  \begin{array}{ll}
  \text{node } p \ v \ \text{heap} & \implies \\
  \text{heap-acc-cond cty-int } (p \oplus \text{fo-node-data}) \ \text{heap} \land \\
  \text{node } p \ v \ \text{heap} \land \\
  (\text{heapop H-1} & \leftarrow \text{heap-get cty-int } (p \oplus \text{fo-node-data}) \ \text{heap}; \\
  \text{H-1} = \text{node-data } v) \\
  \end{array}
  \]

• Heap matching applies unfolding rules automatically \([3, 2]\)

• Example

  \[
  \begin{array}{ll}
  \text{node } p \ v = \\
  (E \text{ data next}. \\
  p \oplus \text{fo-node-data} \mapsto \text{cty-int. data} \ast p \oplus \text{fo-node-next} \mapsto \text{cty-ptr. next} \cdot \\
  v = (| \text{node-data} = \text{data}, \text{node-next} = \text{next}|)) \\
  \end{array}
  \]
Example Unfolding

- Example \( r = p\rightarrow\text{data}; \) requires \( p \oplus \text{fo-node-data} \mapsto \ldots \)

\[ \Rightarrow \text{Heap matching applies rules and leaves} \]
\[
[v = (\{ \text{node-data} = \text{data}, \text{node-next} = \text{next} \});
(p \oplus \text{fo-node-data} \mapsto \text{cty-int. data} \star
p \oplus \text{fo-node-next} \mapsto \text{cty-ptr. next})
\text{heap}
\] \[\mapsto \text{data} = \text{node-data} \, v \]

- Post-condition requires “re-assembled” heap
\[
(p \oplus \text{fo-node-data} \mapsto \text{cty-int. data} \star
p \oplus \text{fo-node-next} \mapsto \text{cty-ptr. next}) \, \text{heap}
\rightarrow \text{node p (} \{ \text{node-data} = \text{data}, \text{node-next} = \text{next} \} \, \text{heap}
\]

\[ \Rightarrow \text{Apply unfolding rules “backwards” in heap tactic} \]
Detail: Normalizing the Symbolic Heap

- Application of unfolding rules leaves $\mathcal{E}, \cdot$ within heap
  $\Rightarrow$ Apply rewriting rules to “pull out” these symbols
  
  $$\forall P \ Q. (\mathcal{E} x. P x) \star Q = (\mathcal{E} x. P x \star Q)$$
  $$\forall P \ Q. P \star (\mathcal{E} x. Q x) = (\mathcal{E} x. P \star Q x)$$
  $$\forall P \ Q. \llbracket \mathcal{E} x. P x \rrbracket = (\exists x. \llbracket P x \rrbracket)$$
  $$\forall P \ Q c. (P \cdot c) \star Q = ((P \star Q) \cdot c)$$
  $$\forall P \ Q c. P \star (Q \cdot c) = ((P \star Q) \cdot c)$$
  $$\forall P \ Q. \llbracket P \cdot c \rrbracket = (\llbracket P \rrbracket \land c)$$

- . . . and to make spatial fragment a linear list
  $\forall P \ Q \ R. (P \star Q) \star R = (P \star (Q \star R))$

  $\Rightarrow$ 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

  ```
  fun list :: "addr ⇒ int list ⇒ addr ⇒ hassn"
  where
    "list p [] q = (emp • (p = q))"
  | "list p (x # xs') q = (E v. node p v • list (node-next v) xs' q
    • x = node-data v)"
  ```

- Many programs access first node ⇨ auto-unfold

  ```
  lemma list-step[sep-unfold]:
    "p ≠ q ⟷ list p xs q = (E v xs'. node p v • (list (node-next v) xs' q)
    • (xs = node-data v ≠ xs'))"
  ```
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) \(\cdot\) 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 run
apply simp
apply heap
apply simp
done

• heap instantiates quantifiers

⇒ No need for apply (rule-tac x="..." in exl)
Variant with Ghost-Variables

\[ q = \text{null}; \]
\[ // @ xs = \text{XS} \]
\[ // @ ys = [] \]
\[ /* @ */ \]
\[ (\text{list } p \ \text{xs } \text{NULL } \star \ \text{list } q \ \text{ys } \text{NULL} \cdot (\text{XS } = (\text{rev } \text{ys } @ \text{xs}))) \text{ heap} \]
\[ */ \]
\[ \text{while } (p \neq \text{null}) \{ \]
\[ \quad t = p->\text{next}; \]
\[ \quad p->\text{next} = q; \]
\[ \quad q = p; \]
\[ \quad p = t; \]
\[ // @ ys = \text{hd } \text{xs } \# \text{ ys} \]
\[ // @ xs = \text{tl } \text{xs} \]
\[ \} \]

- 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


