Interactive Software Verification

Spring Term 2013

Holger Gast
gasth@in.tum.de

2.7.2013
Today: Introduction to Separation Logic

- Limits of Burstall’s memory model
- Basics of Separation Logic [13, 16]
- Next week: verifying list algorithms
Recap: Loop Invariants ⇒ List Insert

- Generalize the desired post-condition
  - Loop test is $a \neq b$, then post-condition from $a = b \land I$
  - If loop test is $a < b$, then add $a \leq b$ to invariant
- Describe (precisely) the achieved partial result
  - Consider the variables that are modified in the body
  - And capture the relationship between them
  - . . . and the relationship to the input values (logical vars)
- Add safety assertions
  - About the range of index variables
  - About pointers not being null
- Preserve information from before loop

⇒ Apply to Exercise 8.3 (interactive)
Recap: Correctness & Abstraction

Diagram:

- **W** to **W'** via Proof in application domain
- **P** to **Q** via Abstraction + Separation Lemmas
- **S** to **S'** via Execution
- **W** to **P** via Abstraction
- **Q** to **S'** via Description

Diagram Components:

- **W**
- **W'**
- **P**
- **Q**
- **S**
- **S'**
Recap: Cheat Sheet

- separation lemma
- split to expose modification
- remove effects
- proofs with abstraction
- abstraction
- assertion "after side-effects"
- program's side-effects
Limits of Burstall’s Memory Model
Burstall’s Memory Model

- The heap model
  - Introduce a separate array for each struct field
  - Index arrays by object pointers: $p \rightarrow f$ becomes $f[p]

- Variant: explicit heap as two-dimensional array
  - $p \rightarrow_h f$ becomes $h(f, p)$

- Example: lists

![Diagram of linked list structure]
Linked Lists on the Heap

• Definition

\textbf{inductive} list :: ” alloc ⇒ nxt ⇒ addr ⇒ addr list ⇒ addr ⇒ bool”

\textbf{where}

\textbf{emptyI}: ” [ p = q; xs = [] ] ⇒ list alloc nxt p xs q”

\textbf{consI}: ” [ alloc p; p \neq \text{NULL}; list alloc nxt (field-get nxt p) xs' q ] ⇒ list alloc nxt p (p \neq xs’) q”

• Separation lemma

\textbf{lemma} list-sep[simp]:

” r \not\in \text{set xs} \implies \text{list alloc (field-upd nxt r y) p xs q = list alloc nxt p xs q}”
Example: Invariant of List Reversal

\[ \exists \ y, \ z. \ \text{list alloc} \ \text{nxt} \ p \ z \ \text{NULL} \land \ \text{list alloc} \ \text{nxt} \ q \ y \ \text{NULL} \land \ X_S = \text{rev} \ y \ \circ \ z \land \ \text{set} \ z \ \cap \ \text{set} \ y = \{\} \land \ \text{distinct} \ z \]

- Auxiliary variable \( X_S \) for initial list
- Two partial lists starting at \( p \) and \( q \)
- So far \( X_S \) has already been reversed partially
- Disjointness conditions to exclude aliasing
Disjointness in the Proof

\[ p \neq \text{NULL}; \text{list alloc} \text{ nxt} \ q \ y s \ \text{NULL}; \text{alloc} \ p; \]
\[ \text{list alloc} \text{ nxt} (\text{field-get} \text{ nxt} \ p) \ x s' \ \text{NULL}; \]
\[ p \notin \text{set} \ y s; \text{set} \ x s' \cap \text{set} \ y s = \{\}; \ p \notin \text{set} \ x s'; \text{distinct} \ x s' \]
\[ \implies \text{list alloc} (\text{field-upd} \text{ nxt} \ p \ q) (\text{field-get} \text{ nxt} \ p) \ x s' \ \text{NULL} \land \]
\[ \text{list alloc} (\text{field-upd} \text{ nxt} \ p \ q) q \ y s \ \text{NULL} \land \]
\[ \text{set} \ x s' \cap \text{set} (p \neq y s) = \{\} \land \text{rev} \ y s \circ p \neq x s' = \text{rev} (p \neq y s) \circ x s' \]

**Lemma** list-sep[simp]:

"\( r \notin \text{set} \ x s \implies \text{list alloc} (\text{field-upd} \text{ nxt} \ r \ y) p \ x s \ q = \text{list alloc} \text{ nxt} \ p \ x s \ q \)"
Limits of Burstall’s Model

- Specifications bloated with disjointness conditions
- Proof effort to exclude aliasing
  - Sets of addresses
  - Must axiomatize & automate set theory
- Abstraction predicates must yield addresses for sep. lemma
  - "Abstraction" is not very strong
  - Requires extra list-data function
- Supports limited range of language features [12]
  - No address operator (& in C)
  - No low-level, byte-addressed access (common in C)
Separation Logic
Separation Logic

- Idea: capture disjointness in syntactic structure of assertions
  - No need to mention addresses
  - No need for complex proofs in set theory
- Approach
  - State consists of heap and store (local variables)
  - Heap is mapping addr \( \rightarrow \) val (generalization to bytes immediate)
  - Assertions are predicates on \((\text{heap} \times \text{store})\)
- Introduce special connectives for assertions
  - \(P \star Q\): \(P\) and \(Q\) hold on disjoint parts of the heap
  - \(P \rightarrow Q\): \(Q\) holds on (disj.) extension of heap validating \(P\)
  - \(p \mapsto v\): heap consists of single address \(p\) with value \(v\)

\(\Rightarrow\) Capture heap layout & -content in simultaneously
A Success Story

- First proposed 2001/02 [13, 11]
- Applications to algorithms
  - Baum/DAG copies [3]
  - Schorr-Waite Graph Marking [17]
- Semi-Automatic proofs on fragment of symbolic heaps
  - Shape analysis [2, 6, 1]
  - Data structures (lists, etc.) [15]
  - Low-level C programs [4]
  - Garbage collectors [7]
- Interactive proofs
  - Memory manager [14]
  - Garbage collectors [9]
Our Goal

- Implement core of automatic methods [2, 15, 4]
- Retain Simpl as a basis [7]

⇒ Obtain straightforward proofs about list algorithms

pre: "(list p XS NULL) heap"
post: "(list ret (rev XS) NULL) heap"
q ≡ null;
/*@inv
(∀ 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;
}

- Demo: stepping through assignments
- Today: basics of separation logic
Basics in Isabelle: the Heap

- Make heap contain typed values (extension to bytes straightforward [7])

  ```
  datatype heap-val =
  VBool bool | VInt int | VNat nat | VPtr addr
  ```

- Heap is simply the expected mapping

  ```
  datatype addr = Addr nat
  datatype heap = Heap "addr → heap-val"
  ```

- Define accessors for domain and content

  ```
  hcnt (Heap c) = c
  hdom (Heap c) = dom c
  ```

- Program logic: introduce global variable heap
A type descriptor packages the heap representation of values

record 'a cty =
  ty-size :: "nat"
  ty-rep :: "'a ⇒ heap-val list"
  ty-val :: "heap-val list ⇒ 'a"

• The size (i.e. consecutive heap addresses needed)
• The representation function
• The value function
• Only point of “encoding”: might use byte sequences [7]

Consistency of type descriptors

\[ ty-ok ct \equiv (\forall v. \text{length (ty-rep ct v)} = \text{ty-size ct} \land \text{ty-val ct (ty-rep ct v)} = v) \]

⇒ Writing a value and then reading it yields the value
Accessing the Heap

- Only access heap through read, write, check-allocatedness
- Parametrize by type descriptor
  
  \[
  \text{heap-get } ct \ p \ h \equiv \text{ty-val } ct \ (\text{map} \ (\text{the} \circ \text{hcnt } h) \ (\text{addr-ivl } p \ (\text{ty-size } ct)))
  \]

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

  \[
  \text{heap-acc-cond } (ct :: 'a \ cty) \ p \equiv \\
  \lambda h. \text{set} \ (\text{addr-ivl } p \ (\text{ty-size } ct)) \subseteq \text{hdom } h
  \]

- “put”: if address is “overwritten”, take local value
- Note: abstraction layer in theory
Points-To

- Points-to \( p \mapsto v \) (or: singleton heap)
  - Heap contains \( v \) at address \( p \)
  - Heap allocates only addresses at \( p \) to store \( v \)

- Idea (\( H \) is the overall heap)

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

\( \Rightarrow \) Assertions also capture allocated addresses
• $P \star Q$: spatial conjunction

• Idea ($H$ is the overall heap)

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

• Definition ($\perp$ is “disjoint”)

• Meaning similar to $\land$, but with spatial side-conditions
Further Spatial Assertions

- **emp**: holds on empty heap
- **true**: holds for any heap

- Extend classical connectives to heaps
  - \( P \land Q \): \( P \) and \( Q \) hold on the same heap
  - \( P \lor Q \): analogously
  - \( \forall x. P \ x \): \( P \) holds on heap for all values \( x \)
  - \( \exists x. P \ x \): \( P \) holds on heap for some value \( x \)

- Pure assertions do not concern the heap
  - Capture local variables just as before
Defining Heap Predicates

• Heap predicates are simple: \( \text{hassn} = \text{heap} \Rightarrow \text{bool} \)

• Infrastructure: predicates for structs

\[
\text{structdef}
= \text{"struct node \{ int data; struct node *next; \}"}
\]

⇒ Yields spatial predicate \( \text{node p v} \), with HOL-record \( v \)

• Build lists inductively

\[
\text{fun list :: "addr} \Rightarrow \text{Word list} \Rightarrow \text{addr} \Rightarrow \text{hassn}"
\]

where

\[
\begin{align*}
\text{"list p [] q} & = \text{(emp} \cdot \text{(p = q)})'' \\
\text{\| \"list p (x \# xs') q} & = \text{(<E v. node p v \Rightarrow list (node-next v) xs'} q} \\
& \cdot x = \text{node-data v)'"}
\end{align*}
\]

• Operator \("\cdot\): classical conjunction with pure assertion

• Note: \( xs \) are data values stored in list
Derived Connectives

- Value stored at address $p$
  \[ p \mapsto v = p \mapsto v \star \text{true} \]

- “Some value”: implicit existential quantification
  \[ p \mapsto \_ = \exists v. \ p \mapsto v \]
  \[ p \mapsto \_ = \exists v. \ p \mapsto v \star \text{true} \]

- Records/structs
  \[ p \mapsto [v_1 \ldots v_n] \]
  means
  \[ p \mapsto v_1 \star \cdots \star p \oplus (n - 1) \mapsto v_n \]
  connectives will not be further discussed
The “Magic Wand”

- $P \rightarrow* Q$: separating implication
  - If $P$ holds on a disjoint extension of the current heap
  - Then $Q$ holds for the extended heap

- Idea

\[ P \rightarrow* Q \equiv \lambda h. \exists h'. h \perp h' \land P h' \implies Q(h \uplus h') \]

- Link to classical implication $P \rightarrow Q$ (not formal, only for pedagogic purposes!)
  - Using additional assumption $P$
  - ... statement $Q$ holds

24 — H.Gast gasth@in.tum.de

Interactive Software Verification, 2.7.2013
Application of Magic Wand

- Hoare rule for writing to the heap [13]
  \[ \{ (p \rightarrow -) \star (p \rightarrow e \ast Q) \} \ast p := e \{ Q \} \]

- Interaction with \( \ast \)
  - Split heap \( Q \) into disjoint parts
  \( \Rightarrow \) Extract the address to be modified
  \( \ldots \) and re-insert it with a new assumption

- Allocatedness: pre-condition asserts that \( p \) must be in the domain of the overall heap

- Likewise: reading from the heap (compare to Hoare's assignment axiom)
  \[ \{ \exists v. (p \rightarrow v) \star (p \rightarrow v \ast Q[v/x]) \} \; x := \ast p \{ Q \} \]
So far

- Basics of spatial assertions
- Can define heap predicates (also inductively)
- Attained main goals
  - Addresses hidden from abstraction predicates
  - Disjointness syntactically clear \( \vdash \) no need for proofs
- Open issues
  - A Hoare logic for separation logic
  - Proofs within separation logic
Lemmas about Connectives

- Goal: proofs by syntactic formula manipulation
- Useful basic properties [13]

\[ P \star (Q \star R) = (P \star Q) \star R \]
\[ P \star Q = Q \star P \]
\[ P \star \text{emp} = P \]
\[ (P \lor Q) \star R = (P \star R) \lor (Q \star R) \]
\[ (\exists x. P \ x) \star Q = \exists x. (P \ x \star Q) \]

- Wand & star

\[
\begin{align*}
P \star Q & \quad \Longrightarrow \quad R \\
\hline
P & \quad \Longrightarrow \quad (Q \rightarrow \star R) \\
\hline
P \star Q & \quad \Longrightarrow \quad R
\end{align*}
\]
Surprises

- We have \([13]\)
  \[ (P \land Q) \star R \implies (P \star R) \land (Q \star R) \]
  \[ (\forall x. P x) \star Q \implies \forall x. (P x \star Q) \]

- But not the reverse directions
  - Address set of \(R\) is not fixed
  - Split of address set by \(\star\) not fixed
  - Similar: different choices for \(x\)

- Unfortunately: no complete proof system [5]

- But: fragment without \(-\star\)
  - Enables efficient proofs
  - Sufficient for many programs [2, 1, 10, 6, 15, 4, 7]
Hoare Logic

- Already seen: dereferencing

\[
\begin{align*}
\{ (p \mapsto -) \star (p \mapsto e \star Q) \} & \quad * p := e \{ Q \} \\
\{ \exists v. (p \mapsto v) \star (p \mapsto v \star Q[v/x]) \} & \quad x := *p \{ Q \}
\end{align*}
\]

- Allocation & deallocation

\[
\begin{align*}
\{ \forall v'. (v' \mapsto [\bar{e}]) \star Q[v'/p] \} & \quad p := \text{alloc}(\bar{e}) \{ Q \} \\
\{ (p \mapsto -) \star Q \} & \quad \text{dispose}(p) \{ Q \}
\end{align*}
\]

⇒ All heap manipulation have concise rules

- Assignments to local variables: as before
The Frame Rule

\[
\frac{\{ P \} \ s \ \{ Q \}}{\{ P \star R \} \ s \ \{ Q \star R \}}
\]

if \( s \) does not change variables occurring in \( R \)

- **Tight specifications**: pre- and post-conditions mention only the used heap fragment, remainder implicitly unchanged
  - Hoare rules check for allocatedness
    ⇒ Implicit statement about used fragment

- **Special case**: function call
  - Specification mentions accessed heap fragment
  - Remaining heap unmodified

- **Note**: also treat allocation/deallocation

- **Refined version**: fractional (read/write) permissions
Forward Reasoning

- Developers explain programs by mental execution (forward)
- But: classical Hoare logic works backwards
- Separation logic enables forward rules

\[
\{ p \mapsto e \} \; p := e \; \{ p \mapsto e \star Q \}
\]

\[
\{ P \} \; p := \text{alloc}(\bar{e}) \; \{ \exists p'. \; p \mapsto [\bar{e}] \star P[p'/p] \}
\]

\[
\{ p \mapsto \neg \star P \} \; \text{dealloc}(p) \; \{ P \}
\]

- Application of such rules
  - Re-arrange given pre-condition to match rule
  - Apply rule
  - Obtain post-condition (final heap state)
  - Basis for automated methods
Local Reasoning

- Frame rule enables **Local reasoning**
  - Prove correctness first for heap actually used
  - Generalize to overall heap
- Examples: garbage collectors [8], Schorr-Waite [17]
- Can formulate alternative Hoare logic, e.g.
  \[
  \{ p \mapsto - \} * p := e \{ p \mapsto e \}
  \]
- Benefits
  - Concentrate on essential statements about programs
    ⇒ Smaller assertions in proofs
  - Possibly re-use proofs of repeated code
Summary

• Introduction to separation logic
  • Captures disjointness syntactically
  • True abstraction predicates (without internal addresses)

• Essential points
  • Spatial connectives for heaps
    ⇒ Capture individual heap fragments & compose
  • Local reasoning
  • Tight specifications
  • Forward reasoning

• Next week: symbolic execution in separation logic
References


