Today

- So far: basics of symbolic execution [1, 7, 2, 3, 4]
  - Symbolic heaps
  - Techniques
    - VCG & reconstructing the execution order
    - Heap matching & structural unfolding
    - Heap normalization

- Today: reasoning & specification techniques
  - The “abstraction layers” return
  - The “cheat sheet” returns
  - More examples on lists
Recap: Spatial Assertions

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

- Star: disjoint parts of the heap
  \[ P \star Q \equiv \lambda h0. \exists h \ h'. h \perp h' \land h0 = h \uplus h' \land P \ h \land Q \ h' \]

- Records, e.g. struct node
  \[
  \begin{align*}
  \text{node } p v & \equiv \mathcal{E} \text{data next. } p \uplus \text{fo-node-data} \mapsto \text{cty-int. } \text{data } \star \\text{p } \uplus \text{fo-node-next} \mapsto \text{cty-ptr. } \text{next } \cdot \\
  v & = (| \text{node-data } = \text{data}, \text{node-next } = \text{next} |)
  \end{align*}
  \]
Symbolic Heaps

- General manipulation of sep-logic formulae hard [5, 6]

- Fragment of symbolic heaps
  \[ \exists x_1 \ldots x_n. P_1 \star \cdots \star P_m \land (Q_1 \land \cdots Q_r) \]

  - spatial part used to “read” values from heap
  - pure part specifies relationships between values

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

- In goal format
  \[ \wedge x_1 \ldots x_n. [(P_1 \star \cdots \star P_m) \text{ heap}; Q_1; \ldots; Q_r] \implies \ldots \]
Symbolic Execution

• Hoare rule for forward reasoning

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

• In SimplC: rule for heap-put

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

formulated for goal format

\[
\land x_1 \ldots x_n. \llbracket (P_1 \ast \ldots \ast P_m) \text{heap}; Q_1; \ldots; Q_r \rrbracket \Longrightarrow \ldots
\]

• Heap-get: lookup “the value” at p

\[
(p \mapsto \text{ct. } v \ast R) h \Longrightarrow \text{heap-get ct p h = v}
\]

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

\[
(p \mapsto \text{ct. } a \ast H) h \Longrightarrow \text{heap-acc-cond ct p h}
\]
Foundation: Heap Matching

- Matching heap
  - Re-arrange compartments of current heap to match
  - Star $\star$ is associative & commutative $\circlearrowright$ order

- Structural unfolding
  \[
  \text{struct node} \{ \text{int data; struct node}^* \text{next; } \}
  \]
  \[
  \begin{align*}
  \text{node p v} &= \quad (\mathcal{E}\text{data next.} \\
  &\quad \quad p \oplus \text{fo-node-data} \mapsto \text{cty-int. data} \star p \oplus \text{fo-node-next} \mapsto \text{cty-ptr. next} \cdot \\
  &\quad \quad v = (|\text{node-data = data, node-next = next|})
  \end{align*}
  \]

- Normalization by rewriting
  - Pull out operators, e.g. $(\mathcal{E} \times. P \times) \star Q = (\mathcal{E} \times. P \times \star Q)$
  - Linearize $\star$-terms by assoc & commute
Lists in Separation Logic

- Recursive definition: enumerate nodes
  \[
  \textbf{fun} \text{ list :: } \text{addr } \Rightarrow \text{int list } \Rightarrow \text{addr } \Rightarrow \text{hassn}
  \]
  \[
  \text{where}
  \]
  \[
  \begin{align*}
  & \text{list } p \; [] \; q = (\text{emp } \land (p = q)) \quad \text{■} \\
  | & \text{list } p \; (x \; \# \; xs') \; q = \left( \mathcal{E} \; v. \; \text{node } p \; v \star \text{list } (\text{node-next } v) \; xs' \; q \right. \\
  & \left. \quad \cdot \; x = \text{node-data } v \right)
  \end{align*}
  \]

- Unfolding the first node
  \[
  \textbf{lemma} \text{ list-step[sep-unfold]:}
  \]
  \[
  \begin{align*}
  & p \neq q \implies \text{list } p \; xs \; q = \left( \mathcal{E} \; v \; xs'. \; \text{node } p \; v \star \text{list } (\text{node-next } v) \; xs' \; q \right) \\
  & \quad \cdot \; (xs = \text{node-data } v \; \# \; xs')
  \end{align*}
  \]

- Splitting lists in general (with ghost variables)
  \[
  \text{list } p \; (xs \; \@ \; ys) \; q = \left( \mathcal{E} \; r. \; \text{list } p \; xs \; r \star \text{list } r \; ys \; q \right)
  \]
Recap: Abstraction Layers

Note: “separation” implicit & automatic in separation logic
Recap: Cheat Sheet

Note: “separation” implicit & automatic in separation logic
Proofs about Lists
Example: List-Reversal

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

```
```

```
```
Destructive List-Append

- Goal: concatenate lists by linking at last node

- Code: search for the last node in first list

```c
if (p == null) r = q;
else {
    r = p;
    while (p->next != null) {
        p = p->next;
    }
    p->next = q;
}
```
Invariant in Separation Logic

- Invariant in Burstall’s model
  
  \[
  \text{list node-alloc node-next r xs1 p} \land \\
  \text{list node-alloc node-next p xs2 NULL } \land \\
  \text{list node-alloc node-next q YS NULL } \land \\
  \text{XS }=\text{xs1 @ xs2} \land \\
  \text{distinct XS }\land \text{p }\neq\text{ NULL }\land \text{set XS }\cap\text{ set YS }=\{\}
  \]

- Idea for separation logic

- Formalize picture directly
  
  \[
  \mathcal{E} \ xs1 \ xs2 \ v. \ \text{list r xs1 p }\star\text{ node p v }\star\text{ list (node-next v) xs2 NULL }\star \\
  \text{list q YS NULL }\cdot\text{ (XS }=\text{xs1 @ node-data v }\neq\text{ xs2)}
  \]
Proof

- Idea for each step
  - \( p \) moves to next node
  - Extract that node from the remainder of the list
  - Add it to the end of list fragment at \( r \)

- Proof (demo during lecture)
  - \( \text{run} \) and \( \text{heap} \) do the job
  - Just a bit of help for quantifiers
  - In particular, \( \text{list} \ p \ xs \ p \) does not imply \( xs = [] \)

\[ \Rightarrow \] Possibly use ghost variables (▷ demo)
More advanced: Sorted Insertion

- Given: sorted list & new node
  \[ \text{list } p \ X S \ \text{NULL} \ \star \ \text{node } q \ v \ \cdot \ \text{sorted } X S \]

- . . . insert the given element to obtain a new sorted list
  \[ \exists \ Y S. \ \text{list } p \ Y S \ \text{NULL} \ \cdot \ Y S = \text{insort} (\text{node-data } v) \ X S \]

- Idea: find insertion point, have all \( xS \) less than \( x \)

- Stop if ever \( q->\text{data} < t->\text{next->data} \)

- Special case: insertion as first element
if (p == null || q->data <= p->data) {
    q->next = p;
    p = q;
} else {
    t = p;
    while (t->next != null && t->next->data < q->data) {
        t = t->next;
    }
    q->next = t->next;
    t->next = q;
}

- As first node if list is empty or new element is so small
- Note: short-circuit semantics necessary for p->data
- Loop: walk on if t->next comes before q
The Final Insertion

\[ q\rightarrow{\text{next}} = t\rightarrow{\text{next}}; \]
\[ t\rightarrow{\text{next}} = q; \]

- Runs if \( t\rightarrow{\text{next}} == \text{null} \) or \( q \) comes before \( t\rightarrow{\text{next}} \)
- First of these handled by same code
Invariant

- Again: idea

\[ \begin{align*}
  \varepsilon \text{ xs ys u. list } p \text{ xs t } * \text{ node } t u * \text{ list } (\text{node-next } u) \text{ ys NULL } * \\
  \text{ node q v} \\
  \cdot (\text{XS } = \text{ xs } @ \text{ node-data } u \neq \text{ ys} \land \text{sorted } \text{XS} \land \\
  \text{node-data } u < \text{ node-data } v \land \\
  (\forall x \in \text{set } \text{xs. } x < \text{ node-data v}))
\end{align*} \]

- Formulation

- Original list

- Information gained by search

---

19 — H.Gast gasth@in.tum.de

Interactive Software Verification, 16.7.2013
Proof

- Cases of insertion at the front
  - Heap matching with explicit witness immediate
  - \texttt{insort} definition/simps “compute” that result is correct

- The case of the loop
  - Invariant holds initially (with witness \( \texttt{xs} = [] \))
  - Invariant preserved: shift node as before; transitivity of <
  - Invariant finally proves post-condition \( \triangleright \) next slide
Proof: Insertion at the End

- Loop terminates because \( p->next \) is null

- Main proof obligation after heap matching

\[
xs @ [u, v] = \text{insort} \; v \; (xs @ [u])
\]

where

\[
(
\begin{align*}
\text{list} & \; p \; xs \; t^* \\
\text{t} & \oplus \text{fo-node-data} \mapsto \text{cty-int.} \; u \; ^* \\
\text{t} & \oplus \text{fo-node-next} \mapsto \text{cty-ptr.} \; q \; ^* \\
\text{q} & \oplus \text{fo-node-data} \mapsto \text{cty-int.} \; v \; ^* \\
\text{q} & \oplus \text{fo-node-next} \mapsto \text{cty-ptr.} \; \text{NULL}
\end{align*}
)\]

- Need auxiliary result (proof by induct & auto)

\[
\left[ \forall \; x \in \text{set} \; xs. \; x < d \right] \implies \text{insort} \; d \; xs = xs @ [d]
\]
Level “application domain” in auxiliary

\[
\forall x \in \text{set} \; xs. \; x < d
\]
\[
\implies \; \text{insort} \; d \; xs = xs \oplus [d]
\]

\( \implies \) Non-trivial (well ...) proofs about abstract values
Garbage Collectors
Garbage Collectors

- Problem: clean up no-longer-used memory \( \Rightarrow \) re-use

- Idea: “collect” objects that will certainly not be accessed

- Program *can* access
  - Variables directly
  - Heap objects indirectly by following pointers

- Goal
  - Determine all reachable objects
  - Delete all unreachable objects

- Follow [4]
Idea: Reachability
How Collectors Work

- Goal: visit (copy, mark) all reachable object
- Working queue or stack
  - Keep pointers of (possibly) unvisited objects
  - Loop: pop one object, treat object, push successors
- Instance: Schorr-Waite Graph Marking
The Challenge

▷ Can we “expose the object” according to cheat sheet?
An idea

- Single out reachable node from the graph
- "Clip" reachability paths before node
- "Restart" reachability with node successors

⇒ Can we formalize this?
• Define reachability “up to point” \( \Rightarrow \) same as lists

\[
\text{reachable } G P Q
\]

• Objects graphs by reachability

\[
\text{graph } P R G Q \equiv \text{objs } R G \cdot R = \text{reachable } G P Q
\]

\( \Rightarrow \) Have atomic predicate for entire object graphs

\( \Rightarrow \) Can use for symbolic heaps
Unfolding Rule for Graphs

\[
\text{graph } P \cup \text{Succs}(G|_D) - (Q \cup D) \cup (R - D) \cup (Q \cup D) \ni G' + G'' = G \land D \subseteq \text{reachable } G P Q
\]
Schorr-Waite – Invariant

\[
\begin{align*}
\text{s.list} \, p \, S \, \text{NULL} & \quad \text{stack (linked list)} \\
\star \, \text{gobjs} \, M \, B & \quad \text{already marked outside stack} \\
\star \, \text{gobjs} \, U \, D & \quad \text{unreachable nodes from input} \\
\star \, \text{g.graph} & \quad \text{entry: r-pointers of stack nodes} \\
& \quad (\{ \, t \, \} \cup \, \text{obj-r} \, '(\, \text{snd} \, ' \, \text{set} \, S \, \text{-is-C}) \, \text{-is-NUL}-\, M \, \text{-s.nodes} \, S) \\
R \, C & \quad \text{still unmarked reachable nodes} \\
(M \cup \, \text{s.nodes} \, S) & \quad \text{boundary: marked \& stack} \\
\cdot \, (t \, \in \, \{ \, \text{NULL} \, \} \cup \, \text{s.nodes} \, S \cup \, R \cup \, M \, \wedge \, \cdots) & \quad \text{progress of marking \&} \\
& \quad \text{safety of pointer dereferencing}
\end{align*}
\]

▷ Unfolding of graph-component automatic in proofs
▷ Proof is semi-automatic, only pure proof obligations
Result

- Unfolding fits symbolic execution framework
- Proofs are substantially smaller than previously known
- Atomic predicate for rather complex memory structure

⇒ Simplicity of symbolic heaps deceptive / no direct limitation
Summary

- Recap: strategies for dealing with heap objects

- Application to symbolic execution
  - List reversal
  - List append
  - List insert
  - (Garbage collectors)

⇒ More examples of invariants & proofs
References


