Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

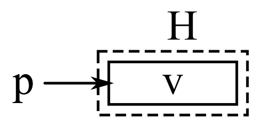
16.7.2013

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
  - Advanced: garbage collectors [4]

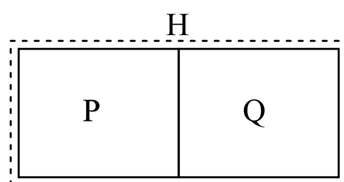
Recap: Spatial Assertions

- Points-to: values in the heap



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

- Star: disjoint parts of the heap



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

- Records, e.g. struct node { int data; struct node *next; }

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

Symbolic Heaps

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

- Fragment of symbolic heaps

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

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

- In Isabelle

$$\mathcal{E} x_1 \ldots x_n.\, \mathsf{P}_1 \star \ldots \star\ \mathsf{P}_m \cdot \mathsf{Q}_{\mathbf{1}} \wedge \ldots \wedge\ \mathsf{Q}_{\boldsymbol{r}}$$

- In goal format

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

Symbolic Execution

- Hoare rule for forward reasoning

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

- In SimplC: rule for heap-put

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

  formulated for goal format

$$\bigwedge x_1 \ldots x_n.\,\llbracket\, (\mathsf{P}_1 \star \ldots \star \mathsf{P}_m)\,\mathsf{heap};\, \mathsf{Q}_1; \ldots\; \mathsf{Q}_r \,\rrbracket \Longrightarrow \ldots$$

- Heap-get: lookup "the value" at p

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

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

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

Foundation: Heap Matching

- Matching heap
  - Re-arrange compartments of current heap to match
  - Star $\star$ is associative & commutative $\Rightarrow$ order
- Structural unfolding struct node { int data; struct node *next; }

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

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

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)"

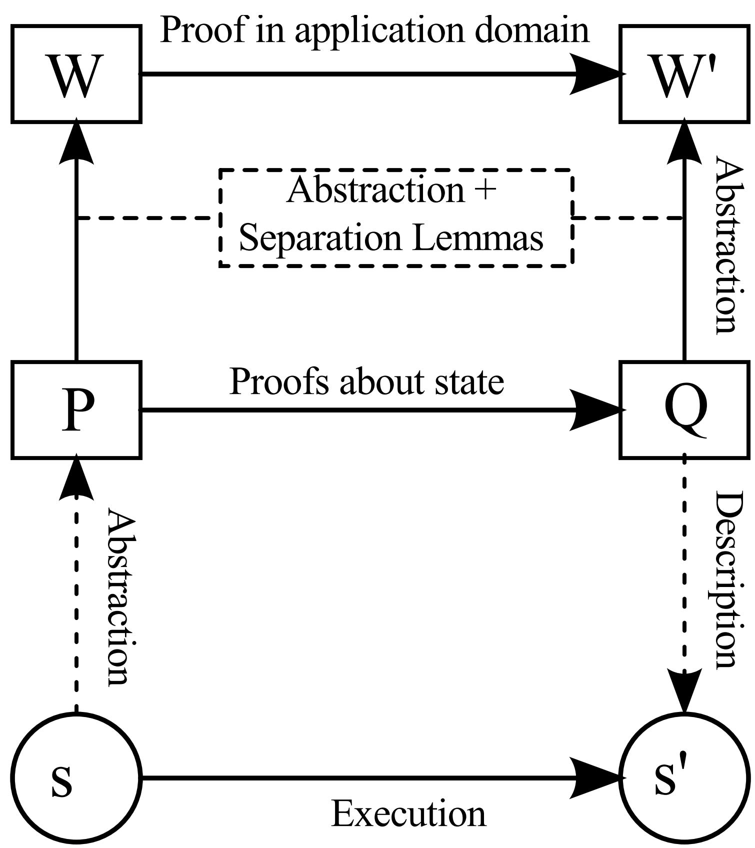
- Unfolding the first node

  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'))"

- Splitting lists in general (with ghost variables)

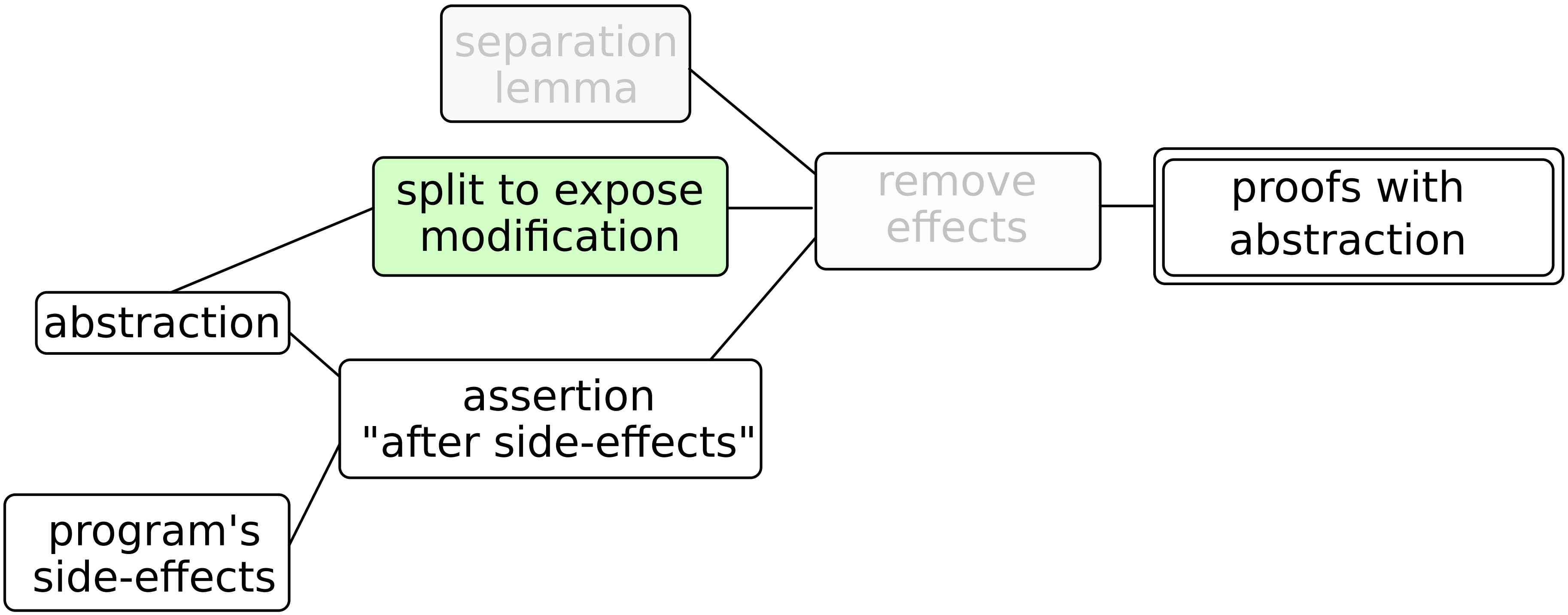
  list p (xs @ ys) q = (ℰ r. list p xs r ⋆ list r ys q)

Recap: Abstraction Layers



Note: “separation” implicit & automatic in separation logic

Recap: Cheat Sheet

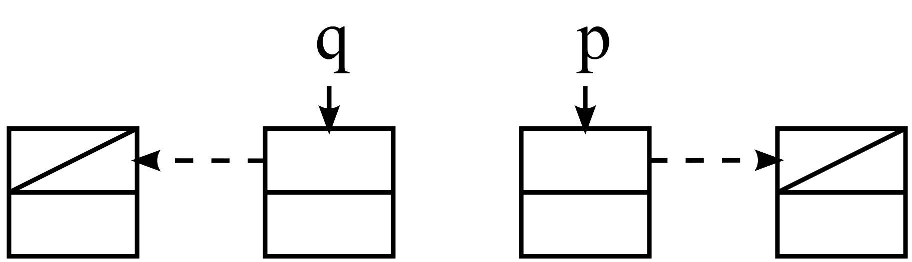


Note: “separation” implicit & automatic in separation logic

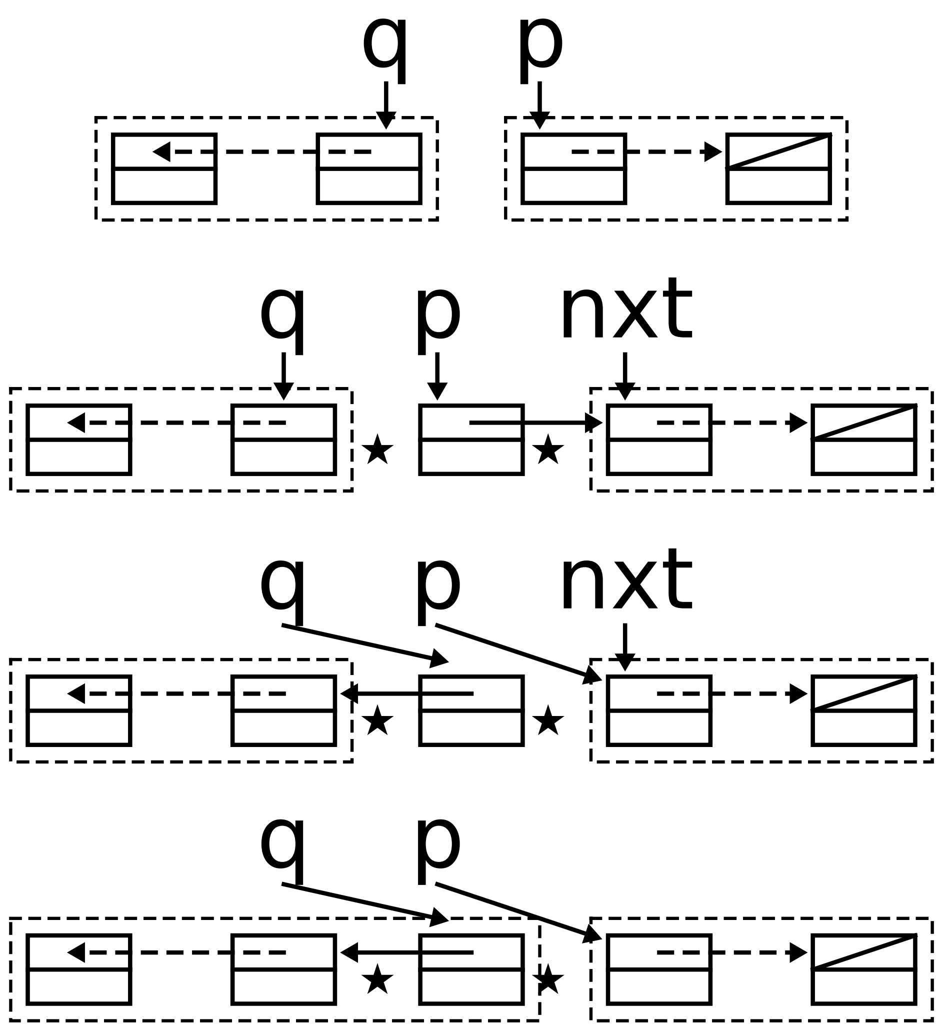
Proofs about Lists

Example: List-Reversal

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

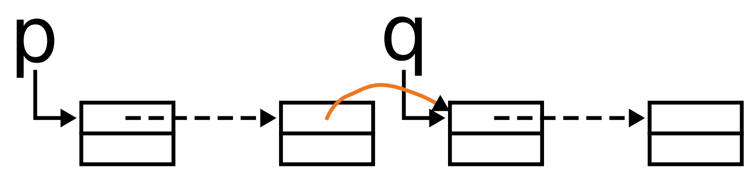


Cheat Sheet & List-Rev



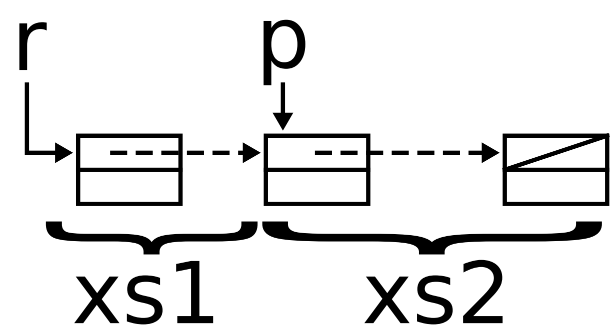
Destructive List-Append

- Goal: concatenate lists by linking at last node



- Code: search for the last node in first list

```
if (p == null) r = q;
else {
  r = p;
  while (p->next != null) {
    p = p->next;
  }
  p->next = q;
}
```

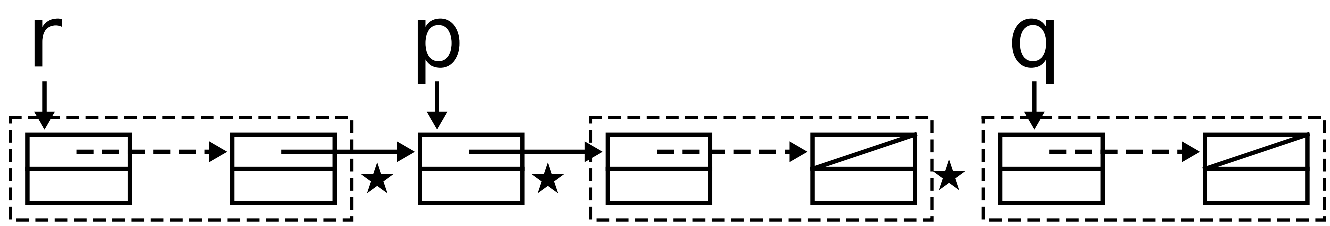


Invariant in Separation Logic

- Invariant in Burstall's model

  $$\begin{array}{l} \text{list node-alloc node-next r xs1 p} \wedge \\ \text{list node-alloc node-next p xs2 NULL} \wedge \\ \text{list node-alloc node-next q YS NULL} \wedge \\ \text{XS} = \text{xs1 @ xs2} \wedge \\ \text{distinct XS} \wedge \text{p} \neq \text{NULL} \wedge \text{set XS} \cap \text{set YS} = \{\} \end{array}$$

- Idea for separation logic



- Formalize picture directly

  $$\begin{array}{l} \mathcal{E}\ \text{xs1 xs2 v. list r xs1 p} \star \text{node p v} \star \text{list (node-next v) xs2 NULL} \star \\ \qquad \text{list q YS NULL} \cdot (\text{XS} = \text{xs1 @ node-data v \# xs2})) \end{array}$$

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)
  - `run` and `heap` do the job
  - Just a bit of help for quantifiers
  - In particular, `list p xs p` does not imply `xs=[]`

$\Rightarrow$ Possibly use ghost variables (➪ demo)

More advanced: Sorted Insertion

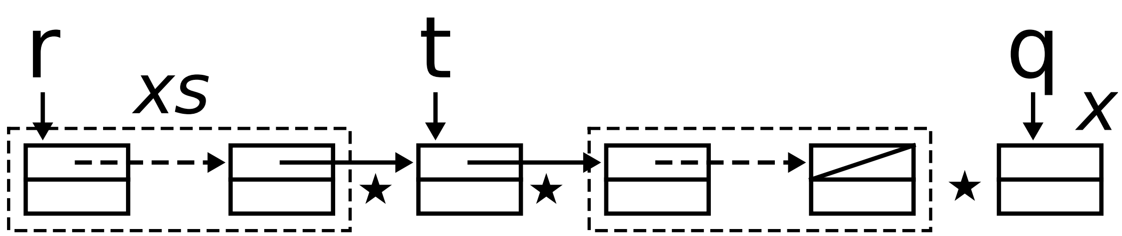
- Given: sorted list & new node

  $\mathsf{list\ p\ XS\ NULL} \star \mathsf{node\ q\ v} \cdot \mathsf{sorted\ XS}$

- . . . insert the given element to obtain a new sorted list

  $\mathcal{E}\,\mathsf{YS.\ list\ p\ YS\ NULL} \cdot \mathsf{YS} = \mathsf{insort\ (node\text{-}data\ v)\ XS}$

- Idea: find insertion point, have all xs less than x

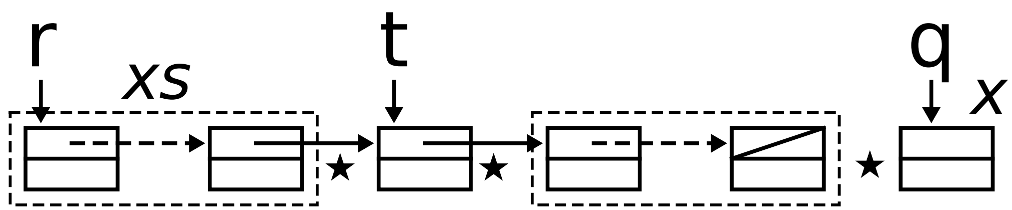


- Stop if ever `q->data < t->next->data`

- Special case: insertion as first element

Code

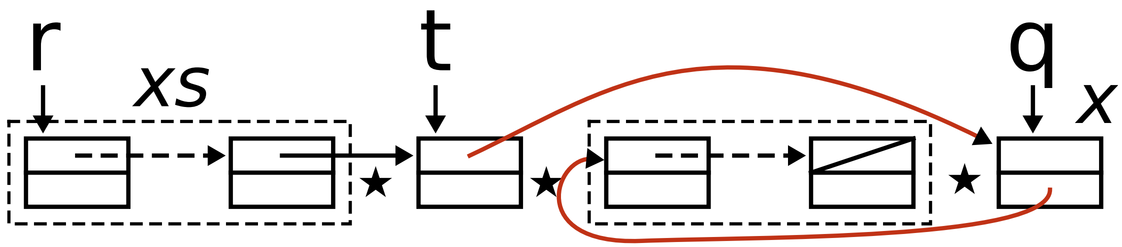
```
if (p == null || q->data <= p->data) {
  q->next = p;
  p = q;
} else {
  t = p;
  while (t->next != null && t->next->data < q->data) {
    t = t->next;
  }
  q->next = t->next;
  t->next = q;
}
```



- 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

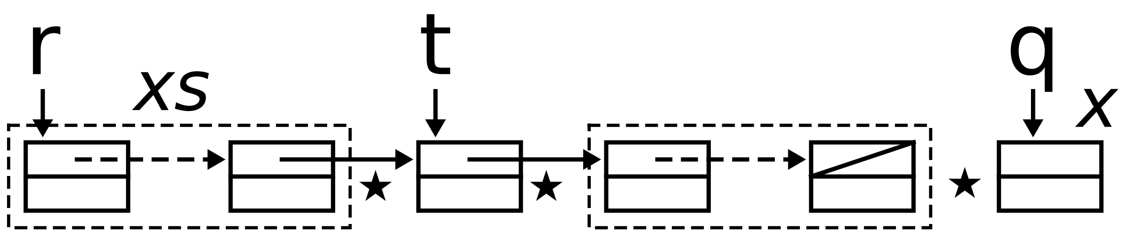
```
q->next = t->next;
t->next = q;
```



- Runs if `t->next == null` or q comes before `t->next`
- First of these handled by same code

Invariant

- Again: idea



- Formulation

$$\mathcal{E}\,\mathsf{xs}\,\mathsf{ys}\,\mathsf{u}.\ \mathsf{list}\ \mathsf{p}\ \mathsf{xs}\ \mathsf{t} \star \mathsf{node}\ \mathsf{t}\ \mathsf{u} \star \mathsf{list}\ (\mathsf{node\text{-}next}\ \mathsf{u})\ \mathsf{ys}\ \mathsf{NULL} \star \mathsf{node}\ \mathsf{q}\ \mathsf{v}$$
$$\cdot\ (\mathsf{XS} = \mathsf{xs}\ @\ \mathsf{node\text{-}data}\ \mathsf{u}\ \#\ \mathsf{ys} \wedge \mathsf{sorted}\ \mathsf{XS} \wedge \mathsf{node\text{-}data}\ \mathsf{u} < \mathsf{node\text{-}data}\ \mathsf{v} \wedge (\forall\, \mathsf{x} \in \mathsf{set}\ \mathsf{xs}.\ \mathsf{x} < \mathsf{node\text{-}data}\ \mathsf{v}))$$

  - Original list
  - Information gained by search

Proof

- Cases of insertion at the front
  - Heap matching with explicit witness immediate
  - `insort` definition/simps “compute” that result is correct
- The case of the loop
  - Invariant holds initially (with witness `xs=[]`)
  - Invariant preserved: shift node as before; transitivity of $<$
  - Invariant finally proves post-condition ⇨ next slide

Proof: Insertion at the End

- Loop terminates because `p->next` is null

- Main proof obligation after heap matching

  $$\mathsf{xs} \mathbin{@} [\mathsf{u}, \mathsf{v}] = \mathsf{insort}\, \mathsf{v}\, (\mathsf{xs} \mathbin{@} [\mathsf{u}])$$

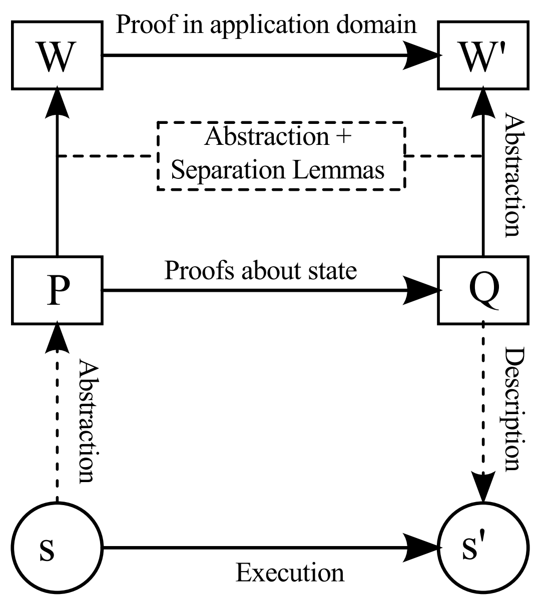
  where

  $$\begin{array}{l} (\mathsf{list}\, \mathsf{p}\, \mathsf{xs}\, \mathsf{t} \star \\ \mathsf{t} \oplus \mathsf{fo\text{-}node\text{-}data} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{u} \star \\ \mathsf{t} \oplus \mathsf{fo\text{-}node\text{-}next} \mapsto \mathsf{cty\text{-}ptr}.\, \mathsf{q} \star \\ \mathsf{q} \oplus \mathsf{fo\text{-}node\text{-}data} \mapsto \mathsf{cty\text{-}int}.\, \mathsf{v} \star \\ \mathsf{q} \oplus \mathsf{fo\text{-}node\text{-}next} \mapsto \mathsf{cty\text{-}ptr}.\, \mathsf{NULL}) \end{array}$$

- Need auxiliary result (proof by induct & auto)

  $$\begin{array}{l} [\![\, \forall\, \mathsf{x} {\in} \mathsf{set}\, \mathsf{xs}.\, \mathsf{x} < \mathsf{d} \\ ]\!] \Longrightarrow \mathsf{insort}\, \mathsf{d}\, \mathsf{xs} = \mathsf{xs} \mathbin{@} [\mathsf{d}] \end{array}$$

Cf. Overview



- Level “application domain” in auxiliary

  $\llbracket \forall x \in \mathsf{set}\, xs.\ x < d$
  $\rrbracket \Longrightarrow \mathsf{insort}\, d\, xs = xs\, @\, [d]$

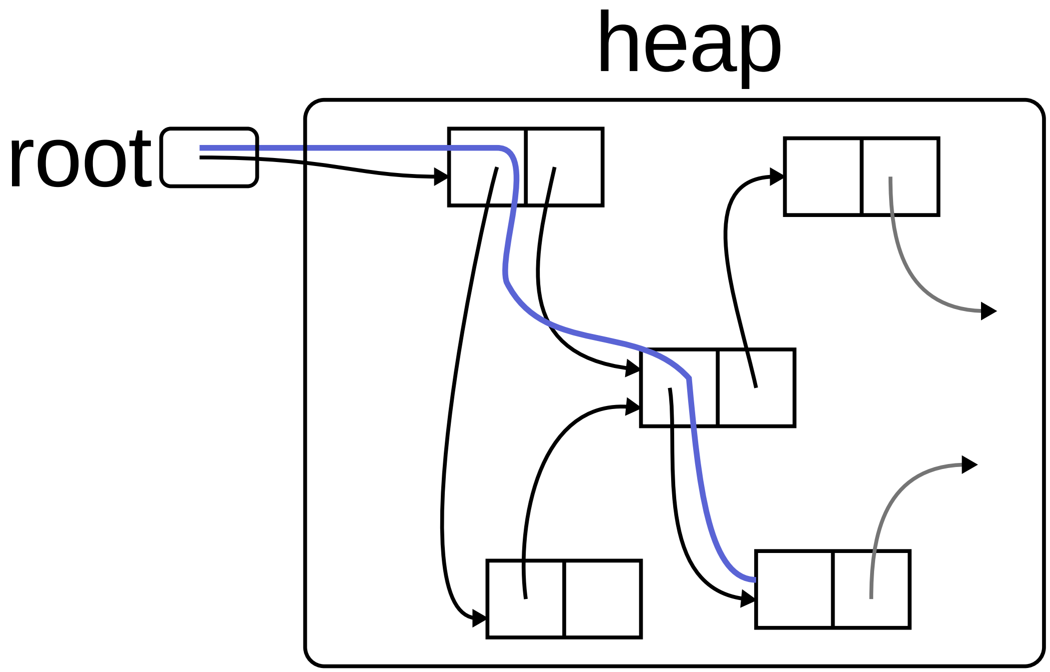
$\Rightarrow$ Non-trivial (well . . .) proofs about abstract values

Garbage Collectors

Garbage Collectors

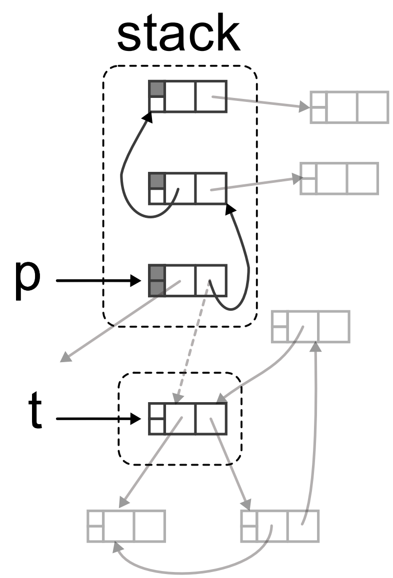
- Problem: clean up no-longer-used memory ⇨ re-use
- Idea: "collect" objects that will certainly not be accessed
- Program *can* access
  - Variables directly
  - Heap objects indirectly by following pointers
- Goal
  - Determine all reachable objects
  - Delete all unreachable objects
- Follow [4]

Idea: Reachability

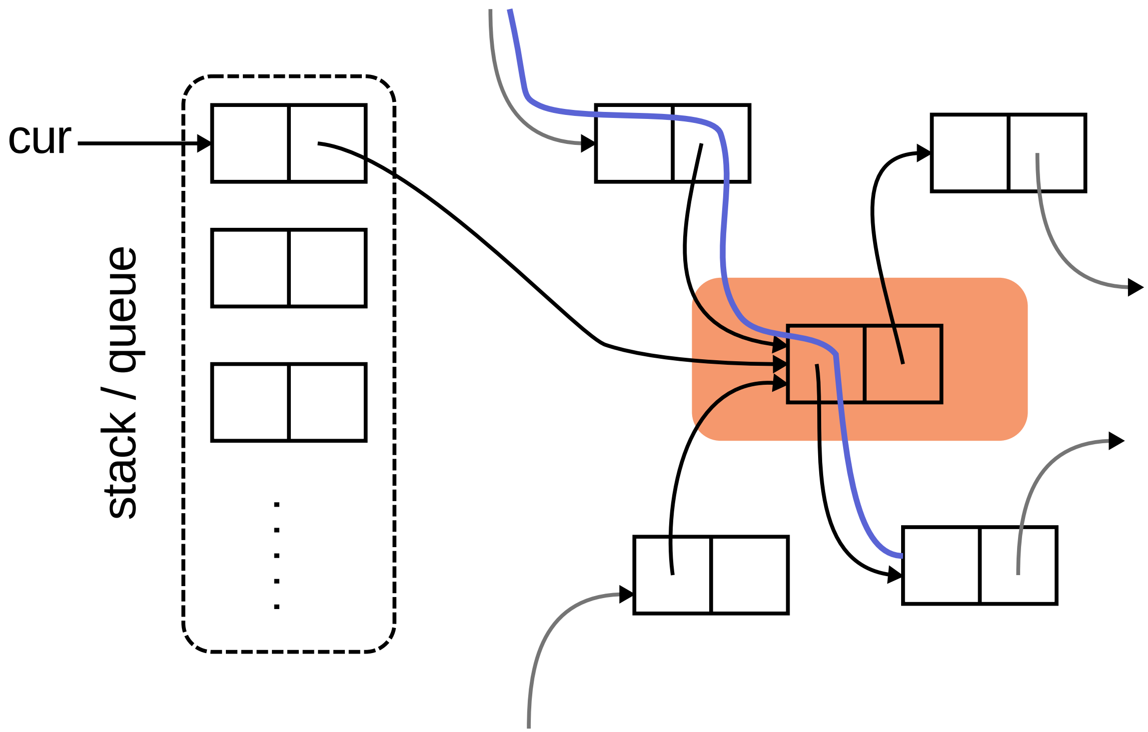


How Collectors Work

- 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

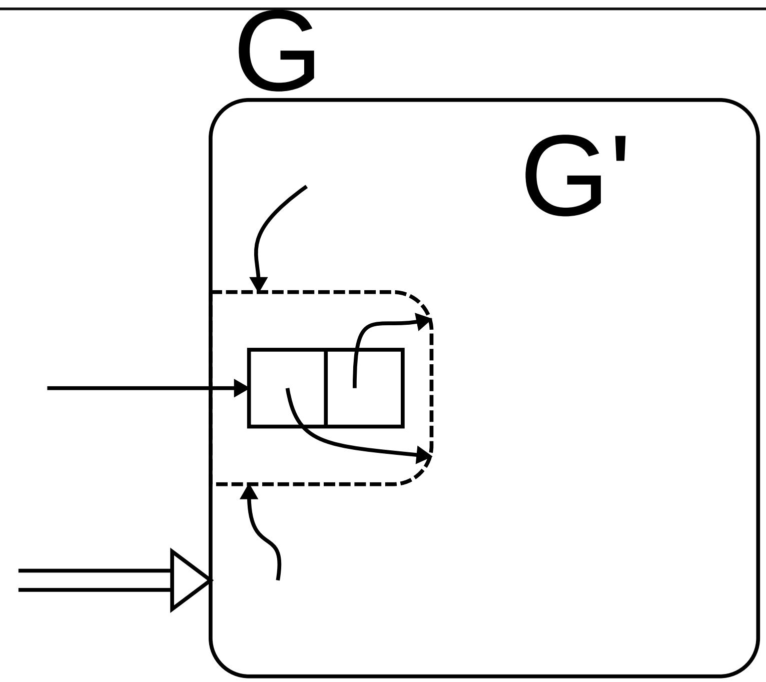


The Challenge



➪ 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?

Yes, we can!

- Define reachability "up to point" ➪ same as lists

$$\mathsf{reachable}\ G\ P\ Q$$

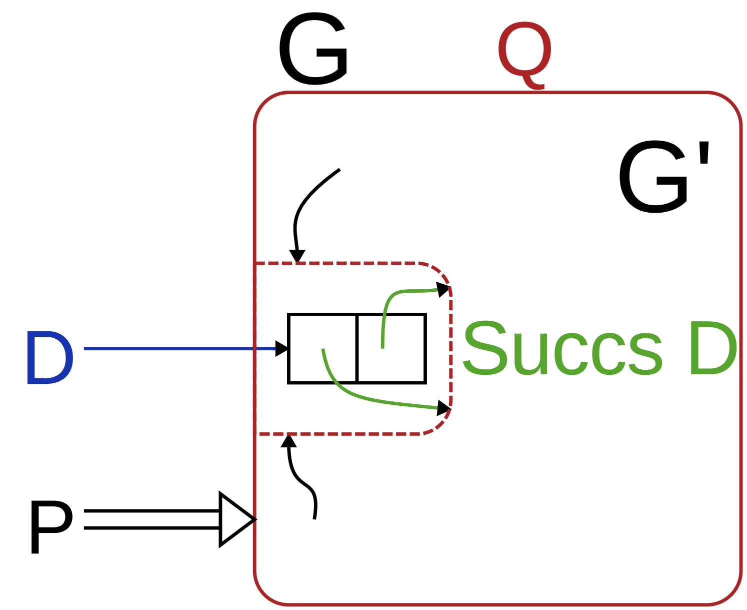
- Objects graphs by reachability

$$\mathsf{graph}\ P\ R\ G\ Q \equiv \mathsf{objs}\ R\ G \cdot R = \mathsf{reachable}\ G\ P\ Q$$

$\Rightarrow$ Have atomic predicate for entire object graphs

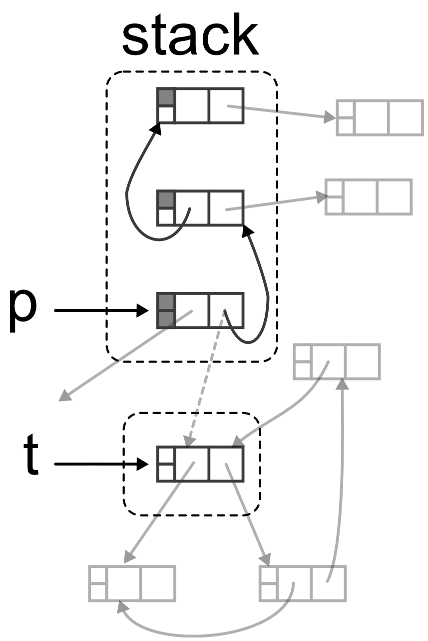
$\Rightarrow$ Can use for symbolic heaps

Unfolding Rule for Graphs



$$\frac{D \subseteq R}{\begin{array}{l}\mathrm{graph}\ P\ R\ G\ Q = (\mathcal{E}\ G'\ G''.\ \mathrm{objs}\ D\ G'' \\ \qquad \star\ \mathrm{graph}\ (P \cup \mathrm{Succs}\ (G|_D) - (Q \cup D)) \\ \qquad\qquad (R - D)\ G' \\ \qquad\qquad (Q \cup D) \\ \qquad \cdot\ G' ++ G'' = G \wedge D \subseteq \mathrm{reachable}\ G\ P\ Q)\end{array}}$$

Schorr-Waite – Invariant



$\mathsf{s.list\ p\ S\ NULL}$	*stack (linked list)*
$\star\ \mathsf{g.objs\ M\ B}$	*already marked outside stack*
$\star\ \mathsf{g.objs\ U\ D}$	*unreachable nodes from input*
$\star\ \mathsf{g.graph}$	*entry: r-pointers of stack nodes*

$\mathsf{((\{\,t\,\} \cup obj\text{-}r\,`\,(snd\,`\,set\,S - is\text{-}C)) - is\text{-}NULL - M - s.nodes\,S)}$

$\mathsf{R\ C}$	*still unmarked reachable nodes*
$\mathsf{(M \cup s.nodes\,S)}$	*boundary: marked & stack*

$\cdot\ \mathsf{(t \in \{NULL\} \cup s.nodes\,S \cup R \cup M \wedge \cdots)}$

progress of marking &

safety of pointer dereferencing

- ➪ 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

$\Rightarrow$ Simplicity of symbolic heaps deceptive / no direct limitation

Summary

- Recap: strategies for dealing with heap obbjects
- Application to symbolic execution
  - List reversal
  - List append
  - List insert
  - (Garbage collectors)

$\Rightarrow$ More examples of invariants & proofs

References

[1] 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.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.