Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

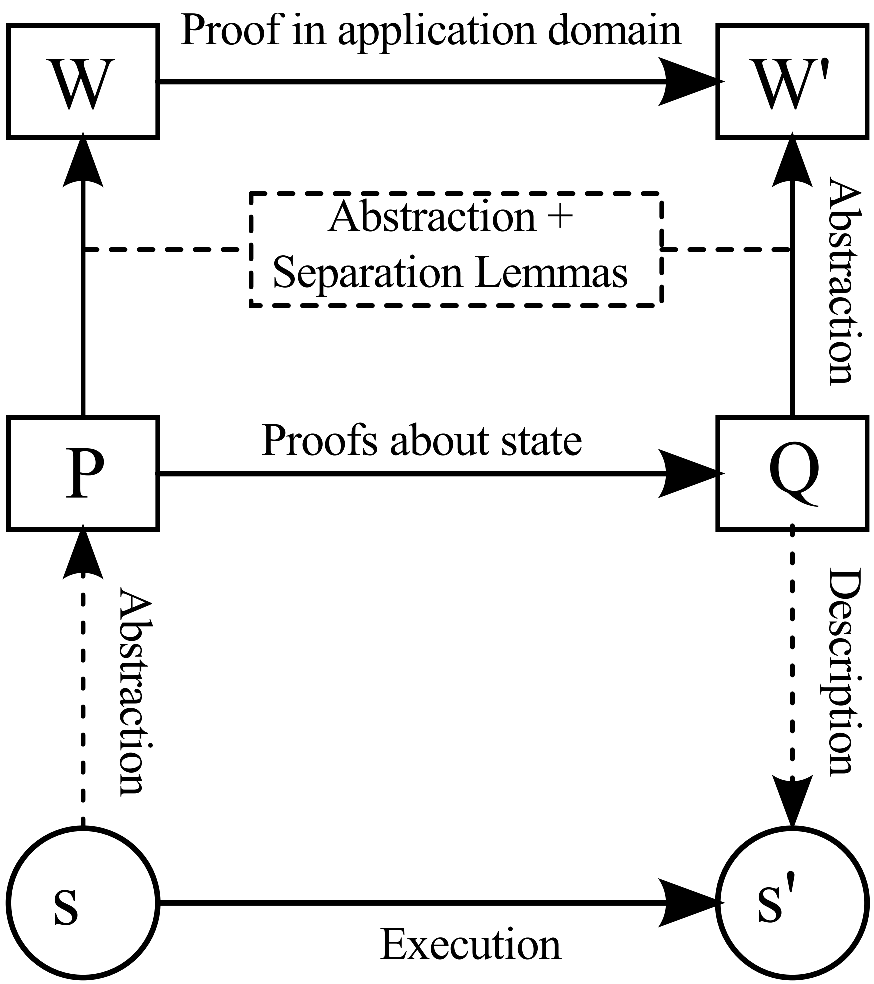
25.6.2013

Today

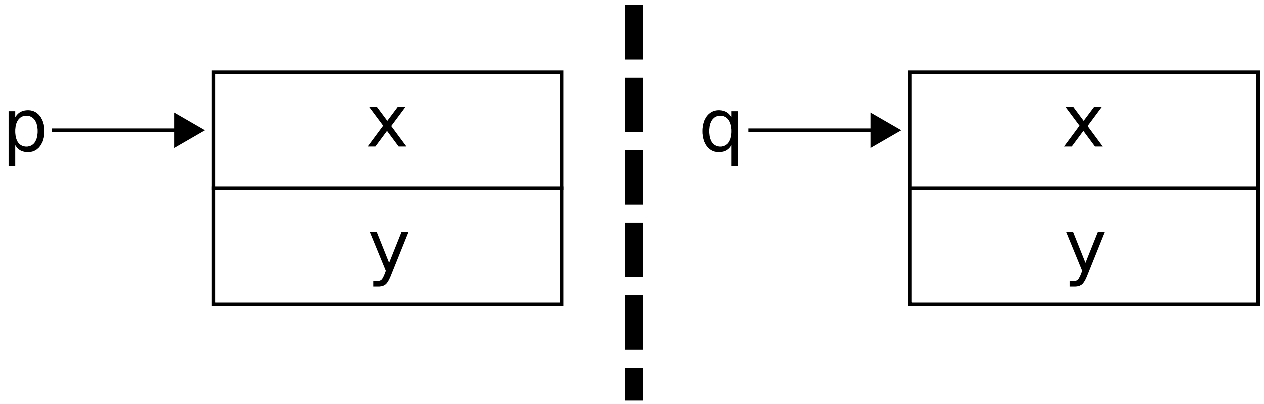
- Manipulating lists on the heap
- Ghost variables as a specification tool

Recap

Correctness with Abstractions



The Problem of Aliasing

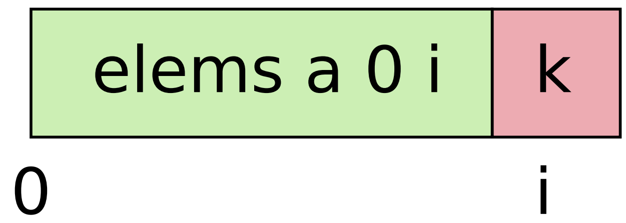


- $\{\mathsf{r0} = \mathsf{q} \rightarrow \mathsf{point\text{-}y} \wedge \mathsf{point\text{-}alloc}\ \mathsf{p} \wedge \mathbf{p} \neq \mathbf{q}\}$`p->y = 0;`$\{\mathsf{r0} = \mathsf{q} \rightarrow \mathsf{point\text{-}y}\}$

  Proof obligation: exclude aliasing

  $\mathsf{field\text{-}get}\ \mathsf{point\text{-}y}\ \mathsf{q} = \mathsf{field\text{-}get}\ (\mathsf{field\text{-}upd}\ \mathsf{point\text{-}y}\ \mathsf{p}\ 0)\ \mathsf{q}$

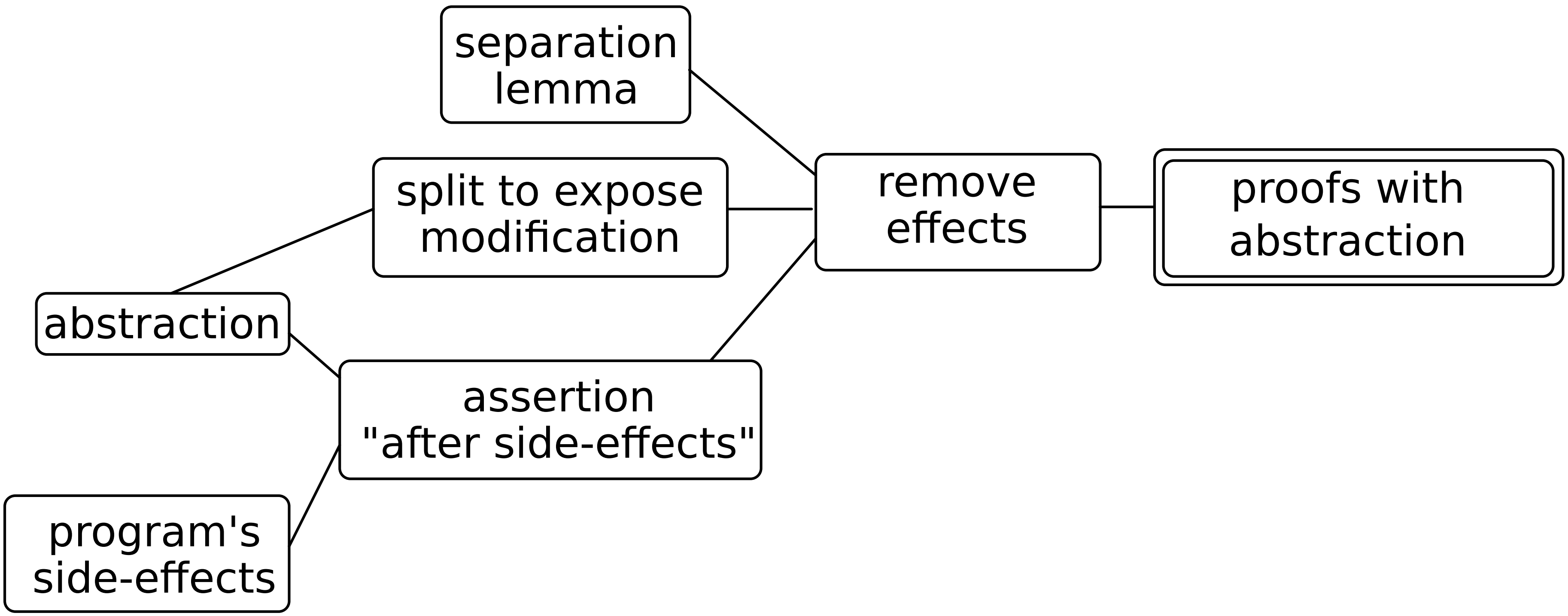
Separation Lemmas & Split



$$k \notin \{i..<j\} \Longrightarrow \text{elems}\,(\text{aset}\;k\;y\;a)\;i\;j = \text{elems}\;a\;i\;j$$

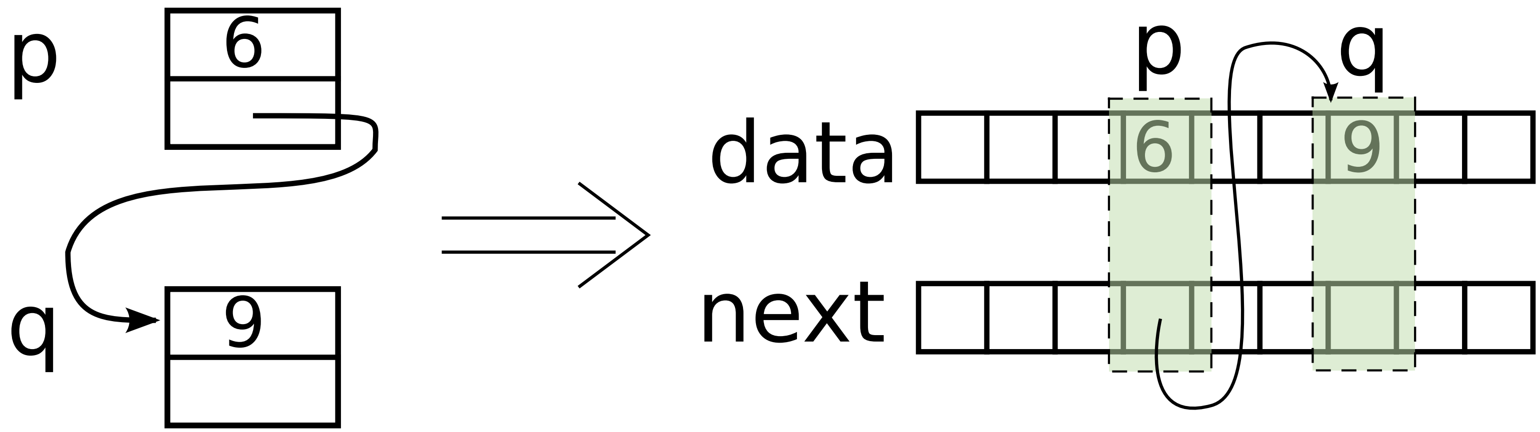
➩ Lift (non-)aliasing proofs to abstractions

Cheat Sheet

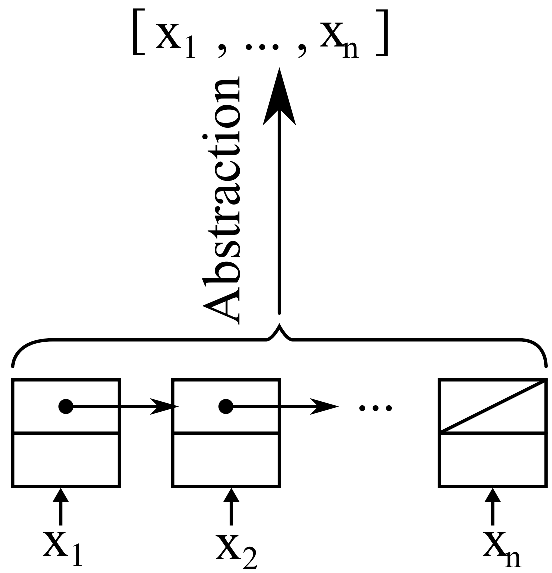


separation lemma
split to expose modification
remove effects
proofs with abstraction
abstraction
assertion "after side-effects"
program's side-effects

Burstall's Heap Model



Lists on the Heap



Using Burstall's heap model

inductive list :: "alloc ⇒ nxt ⇒ addr ⇒ addr list ⇒ addr ⇒ bool"
where
emptyI: "⟦ p = q; xs = [] ⟧ ⟹ list alloc nxt p xs q"
| consI: "⟦ alloc p; p ≠ NULL; list alloc nxt (p→nxt) xs' q
⟧ ⟹ list alloc nxt p (p # xs') q"

Example: Counting the List Nodes

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

pre: $''$ list node-alloc node-next p XS NULL$''$

post: $''$ nat n = length XS$''$

```
    n = 0;
/*@ ∃ys. list node-alloc node-next p ys NULL ∧
           length XS = nat n + length ys ∧ 0 ≤ n  */
 while (p != null) {
     n = n + 1;
     p = p->next;
 }
```

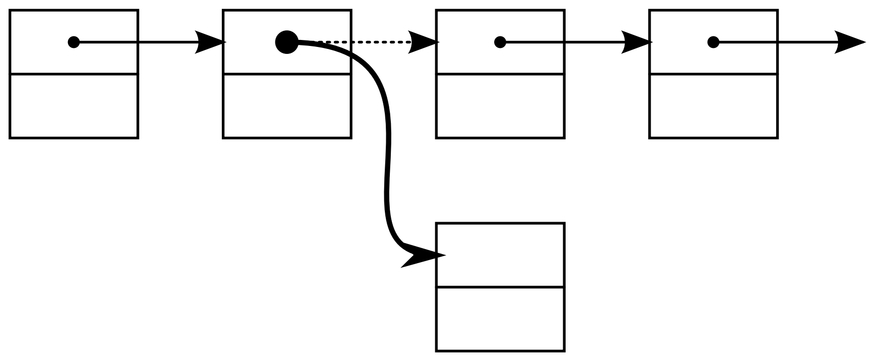
Example: Sum over List Data

```
pre:  "list node-alloc node-next p XS NULL ∧ p = P"
post:  "s = listsum (list-data node-data XS)"
s = 0;
/*@  ∃xs ys. list node-alloc node-next P xs p ∧
          list node-alloc node-next p ys NULL ∧
          XS = xs @ ys ∧
          s = listsum (list-data node-data xs)  */
while (p != null) {
  s = s + p->data;
  p = p->next;
}
```

Manipulating Lists

Side-effects and Separation Lemmas

- Problem: side-effects may destroy list structure



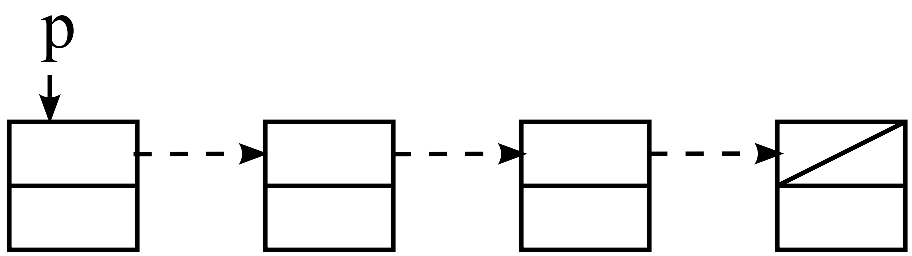
- Insight: result of `list` depends only of nodes in fragment

  lemma list-sep[simp]:
  "$r \notin \mathrm{set}\ xs \Longrightarrow \mathrm{list}\ \mathrm{alloc}\ (\text{field-upd}\ nxt\ r\ y)\ p\ xs\ q = \mathrm{list}\ \mathrm{alloc}\ nxt\ p\ xs\ q$"

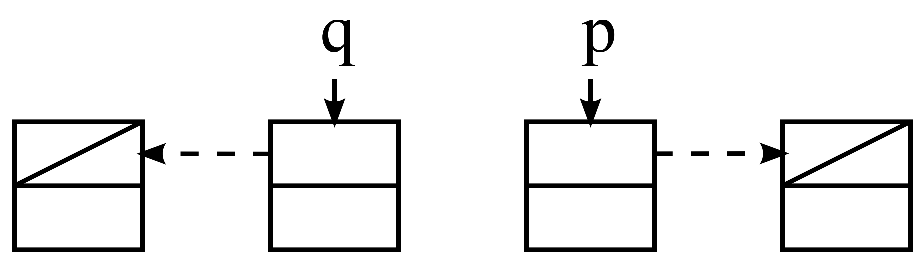
$\Rightarrow$ Reduced aliasing problem to proving set properties

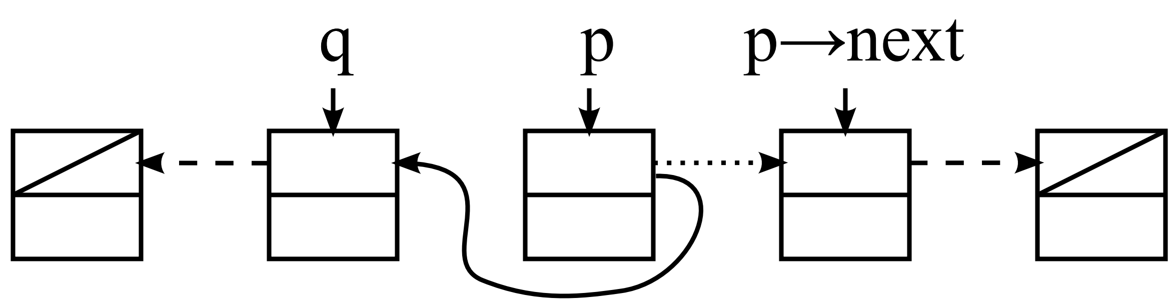
$\Rightarrow$ Isabelle's provers solve such proof obligations

Example: List Reversal

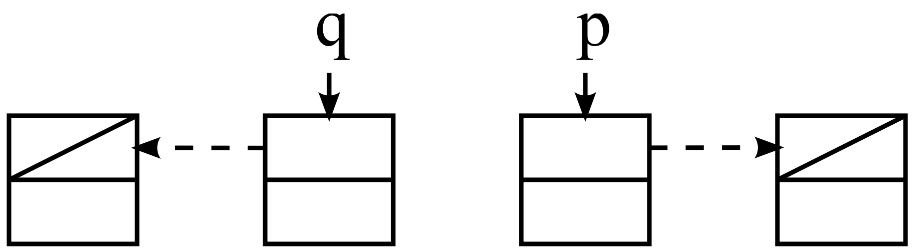



```
q = NULL;
while (p ≠ NULL) {
  t = p→nxt;
  p->nxt = q;
  q = p;
  p = t;
}
```





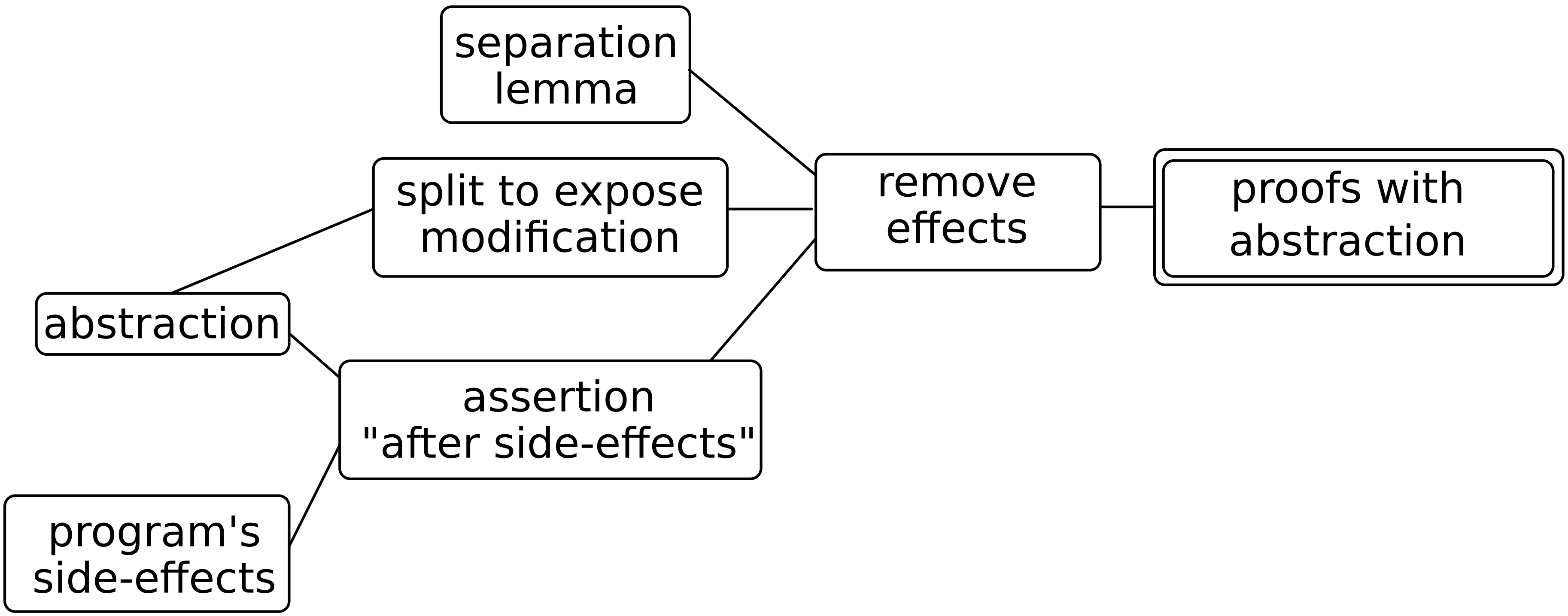
Invariant of List Reversal



$$\exists\, \mathsf{ys}\; \mathsf{zs}.\ \mathsf{list\;alloc\;nxt\;p\;zs\;NULL} \wedge \mathsf{list\;alloc\;nxt\;q\;ys\;NULL} \wedge$$
$$\mathsf{XS} = \mathsf{rev\;ys} \mathbin{@} \mathsf{zs} \wedge \mathsf{set\;zs} \cap \mathsf{set\;ys} = \{\} \wedge \mathsf{distinct\;zs}$$

- Initial list is `XS`
- During loop: two lists starting at p and q
- Invariant strategy “partial result”:
  `XS` has already been partially reversed
- State disjointness to exclude alias
- Holds initially, since `q=NULL`, i.e. `ys=[]`

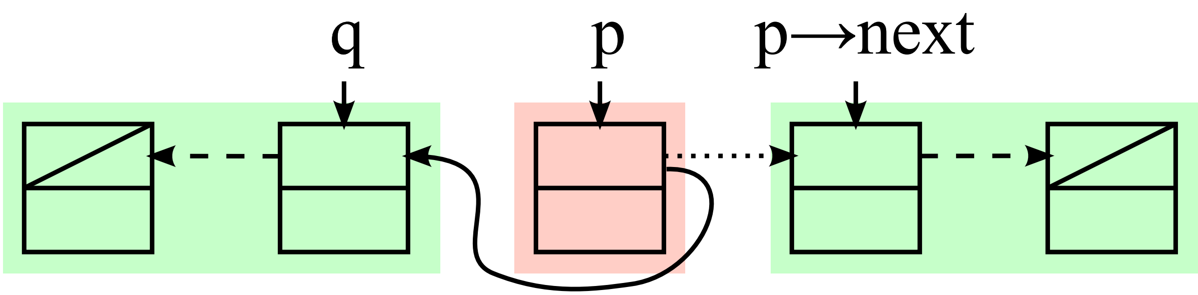
Cheat Sheet



separation lemma
split to expose modification
remove effects
proofs with abstraction
abstraction
assertion "after side-effects"
program's side-effects

Preserving the Invariant

$\exists$ ys zs. list alloc nxt p zs NULL $\wedge$ list alloc nxt q ys NULL $\wedge$
XS = rev ys @ zs $\wedge$ set zs $\cap$ set ys = {} $\wedge$ distinct zs



- We write to p, such that we must
  - Extract p from the list abstraction
  - Prove that green nodes are not changed
    - ➪ Apply separation lemma for lists
  - Finally add node p to list at q

Befor reading on, look at the "cheat sheet" on p. 7

Looking at the Single Steps

- Extract p from the list abstraction

$$\llbracket \mathsf{p} \neq \mathsf{NULL};\ \frac{\exists\, \mathsf{xs'}.\ \mathsf{zs} = \mathsf{p} \# \mathsf{xs'} \wedge \mathsf{alloc\ p} \wedge \mathsf{list\ alloc\ nxt\ (field\text{-}get\ nxt\ p)\ xs'\ NULL};}{\mathsf{list\ alloc\ nxt\ q\ ys\ NULL};\ \mathsf{set\ zs} \cap \mathsf{set\ ys} = \{\};\ \mathsf{distinct\ zs} \rrbracket}$$

$$\Longrightarrow \exists\, \mathsf{ysa\ zsa}.\ \mathsf{list\ alloc\ (field\text{-}upd\ nxt\ p\ q)\ (field\text{-}get\ nxt\ p)\ zsa\ NULL} \wedge \frac{(\exists\, \mathsf{xs'}.\ \mathsf{ysa} = \mathsf{p} \# \mathsf{xs'} \wedge \mathsf{alloc\ p} \wedge \mathsf{list\ alloc\ (field\text{-}upd\ nxt\ p\ q)\ q\ xs'\ NULL}) \wedge}{\mathsf{set\ zsa} \cap \mathsf{set\ ysa} = \{\} \wedge \mathsf{distinct\ zsa} \wedge \mathsf{rev\ ys} \mathbin{@} \mathsf{zs} = \mathsf{rev\ ysa} \mathbin{@} \mathsf{zsa}}$$

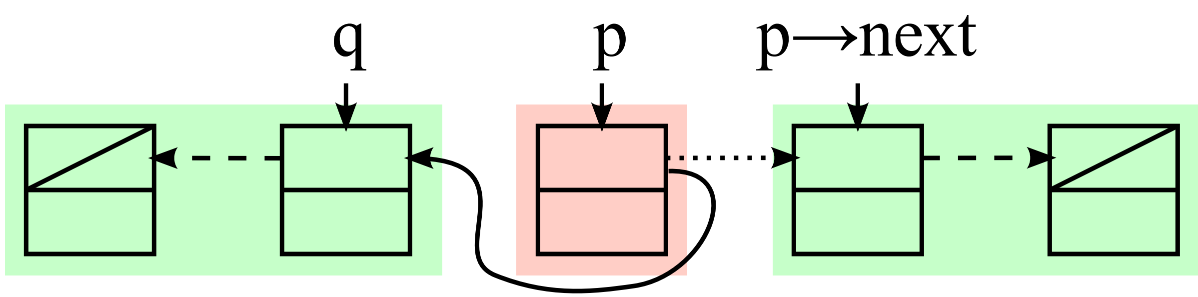
- $\exists$-quantifiers are annoying (`clarsimp & exI`)

$$\llbracket \mathsf{p} \neq \mathsf{NULL};\ \underline{\mathsf{list\ alloc\ nxt}}\ \mathsf{q\ ys\ NULL};\ \mathsf{alloc\ p};\ \mathsf{list\ alloc\ nxt\ (field\text{-}get\ nxt\ p)\ xs'\ NULL};\ \mathsf{p} \notin \mathsf{set\ ys};\ \mathsf{set\ xs'} \cap \mathsf{set\ ys} = \{\};\ \mathsf{p} \notin \mathsf{set\ xs'};\ \mathsf{distinct\ xs'} \rrbracket$$

$$\Longrightarrow \mathsf{list\ alloc\ (field\text{-}upd\ nxt\ p\ q)\ (field\text{-}get\ nxt\ p)\ xs'\ NULL} \wedge \underline{\mathsf{list\ alloc\ (field\text{-}upd\ nxt\ p\ q)}}\ \mathsf{q\ ys\ NULL} \wedge \mathsf{set\ xs'} \cap \mathsf{set\ (p \# ys)} = \{\} \wedge \mathsf{rev\ ys} \mathbin{@} \mathsf{p} \# \mathsf{xs'} = \mathsf{rev\ (p \# ys)} \mathbin{@} \mathsf{xs'}$$

- Only remaining problem: side-effects

Rewriting with the Separation Lemma



- Simplify: list alloc (field-upd nxt p q) (field-get nxt p) xs' NULL

⇒ Pre-condition of `list_sep` yields goal p $\notin$ set xs'

- Nice: given by the `distinct` clause of the invariant
- Apply same ideas to second list abstraction
- In this case we could also use a lemma on the special case [3]:

  lemma list-hd-not-in-tl:
  "list alloc nxt (field-get nxt p) xs NULL $\Longrightarrow$ p $\notin$ set xs"

Slightly More Automatic Proofs

- Have understood outline of proof
  - Unfold lists by `list-step`
  - Remove side-effects by `list-sep`
  - All side-conditions are proven by `auto`

⇒ Isabelle should be of more assistance

```
apply vcg
 apply (auto simp add: list-step)
 apply (rule-tac x="p # ys" in exI)
 apply auto
 done
```

- Only a little help needed on *one* ∃-quantifier

⇒ Can be get rid of this remaining piece of interaction?

Ghost Variables

Ghost-Variablen

- Idea: witnesses for $\exists$-quantifiers follow computation

$\Rightarrow$ Write down witnesses as "assignments"

- Ghost variables (e.g. [1])
  - Can be written to like ordinary variables
  - Can be read in assertions & ghost statements
  - Cannot be read in non-ghost statements

  $\Rightarrow$ Will not be required during execution
  - Content: any HOL values (lists, trees, functions, . . .)
- Example: list-sum with ghost variables (interactive)
- Beware: $\neq$ auxiliary variables
  - Those are $\forall$-quantified
  - They cannot be modified
  - Goal: connect pre-/post-condition & invariants

List Reversal with Ghost Variables

```
q = null;
//@ ys = []
//@ zs = XS
/*@ list node-alloc node-next p zs NULL ∧
      list node-alloc node-next q ys NULL ∧
      set zs ∩ set ys = {} ∧ distinct zs ∧
      XS = rev ys @ zs */
while (p != null) {
  t = p→next;
  p→next = q;
  q = p;
  p = t;
  //@ ys = (hd zs) # ys
  //@ zs = tl zs
}
. . .
by vcg (fastforce simp add: list-step)+ // that's it!
```

Looking at the Details

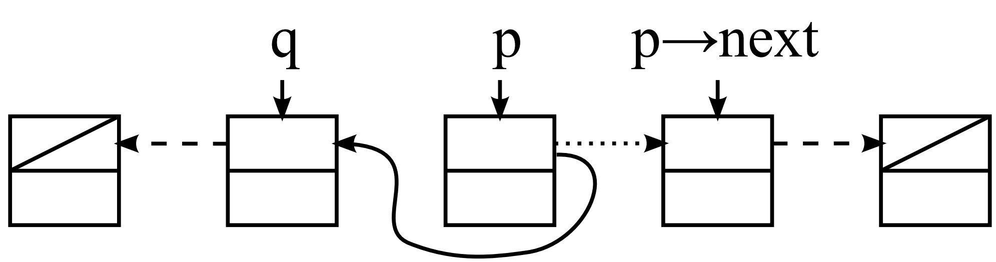
- Loop updates ghots state

  list node-alloc node-next p **zs** NULL $\wedge$
  list node-alloc node-next q **ys** NULL $\wedge$

  . . .

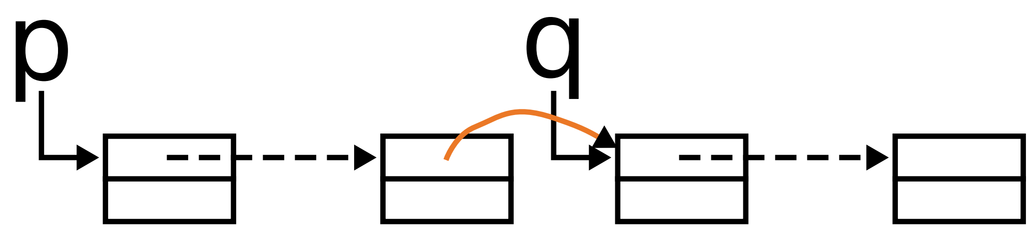
  //@ ys = (hd zs) # ys
  //@ zs = tl zs

- Compare to idea of code



Example: Destructive List Append

```
if (p == null)
  r = q;
else {
  r = p;
  //@ xs1 = []
  //@ xs2 = XS
  while (p->next != null) {
    p = p->next;
    //@ xs1 = xs1 @ [ hd xs2 ]
    //@ xs2 = tl xs2
  }
  p->next = q;
}
```



The Proof Structure with Ghost State

- Previously: $\exists\, \mathbf{ys}.\ \mathsf{list\ node\text{-}alloc\ node\text{-}next\ p}\ \mathbf{ys}\ \mathsf{NULL} \wedge \cdots$

⇒ Must fill ys while proving

- Ghost state: "result" of reading a list from a heap is given
- Can simplify proving proving by classical reasoners
- Introduce rewrite rules

$$
\begin{array}{l}
\mathsf{list\ alloc\ nxt\ p\ []\ q} = (\mathsf{p} = \mathsf{q}) \\
\mathsf{list\ alloc\ nxt\ p\ (x \# xs)\ q} = \\
\qquad\qquad (\mathsf{p} = \mathsf{x} \wedge \mathsf{alloc\ p} \wedge \mathsf{p} \neq \mathsf{NULL} \wedge \mathsf{list\ alloc\ nxt\ (p \rightarrow nxt)\ xs\ q}) \\
\mathsf{list\ alloc\ nxt\ p\ (xs\ @\ ys)\ q} \\
\qquad\qquad = (\exists\, \mathsf{r}.\ \mathsf{list\ alloc\ nxt\ p\ xs\ r} \wedge \mathsf{list\ alloc\ nxt\ r\ ys\ q})
\end{array}
$$

The Core Assertions

- Precondition

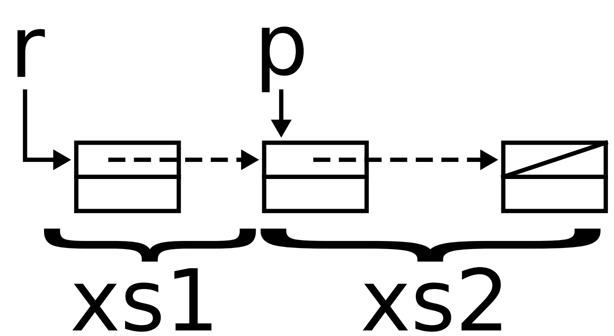
  list node-alloc node-next p XS NULL $\wedge$
  list node-alloc node-next q YS NULL

- Postcondition

  list node-alloc node-next r (XS @ YS) NULL

- Invariant

  list node-alloc node-next r xs1 p $\wedge$
  list node-alloc node-next p xs2 NULL $\wedge$
  list node-alloc node-next q YS NULL $\wedge$
  XS = xs1 @ xs2 $\wedge$ p $\neq$ NULL



Adding Disjointness Assertions

- Clearly the two input lists are disjoint

  $\text{set}\,XS \cap \text{set}\,YS = \{\}$

- Furthermore, we find that the first list must not be cyclic

  $\text{distinct}\,XS$

- Both are maintained (obviously) by the loop

- And then enable us to reason about the final

```
p->next = q;
```

Conclusion

- Concept: ghost state
  - Store HOL values in mutable locations
  - Cannot access from usual program

  ⇒ Not present during execution

- Role in proofs: provide witnesses for existentials

⇒ Proofs can "compute" abstractions

⇒ Proofs become more automatic

⇒ The (efficient) simplifier can do more

⇒ SMT-Solvers can do more [2, 4]

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects*, volume 4111 of *LNCS*. Springer, 2006.

[2] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-16)*, 2010.

[3] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1–2):200–227, 2005.

[4] MichałMoskal. Programming with triggers. In Bruno Dutertre and Ofer Strichman, editors, *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. ACM, 2009.