Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

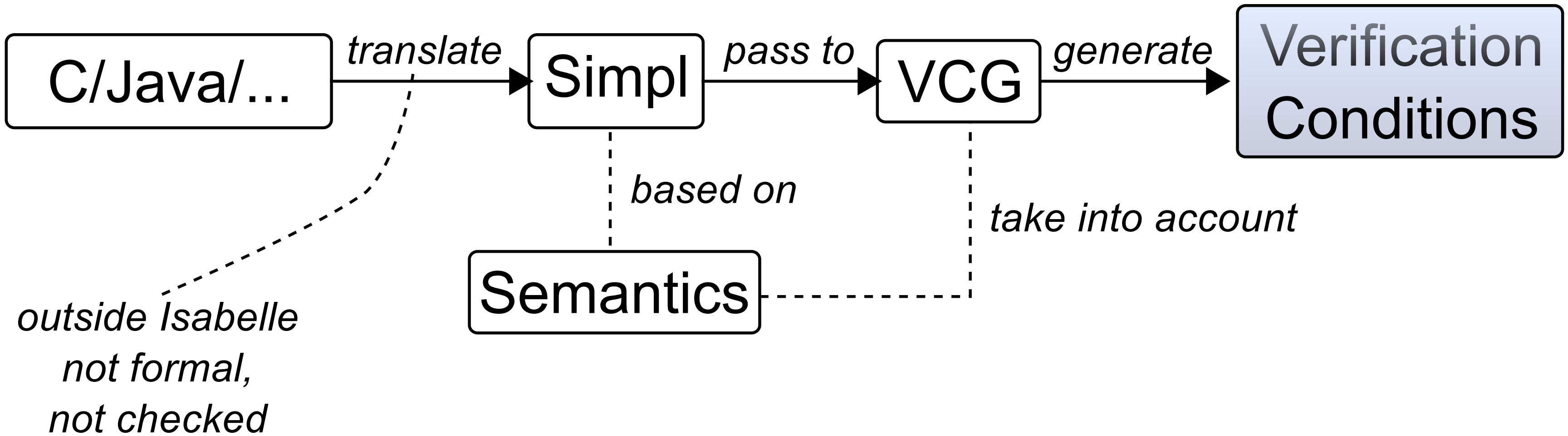
18.6.2013

Today: The Heap

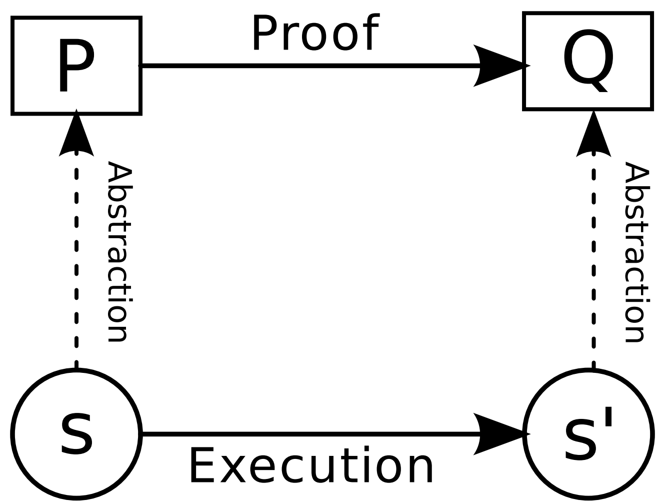
- Burstall's memory model for records on the heap
- Abstraction predicates for linked lists

Gathering the Bits & Pieces

Shallow Embedding

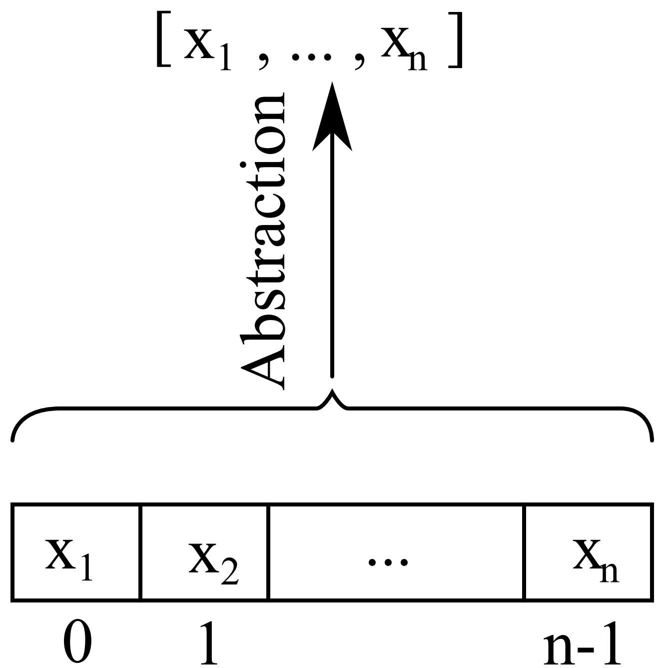


Partial Correctness



- Connection semantics with Hoare triples
- Soundness theorem for the VCG

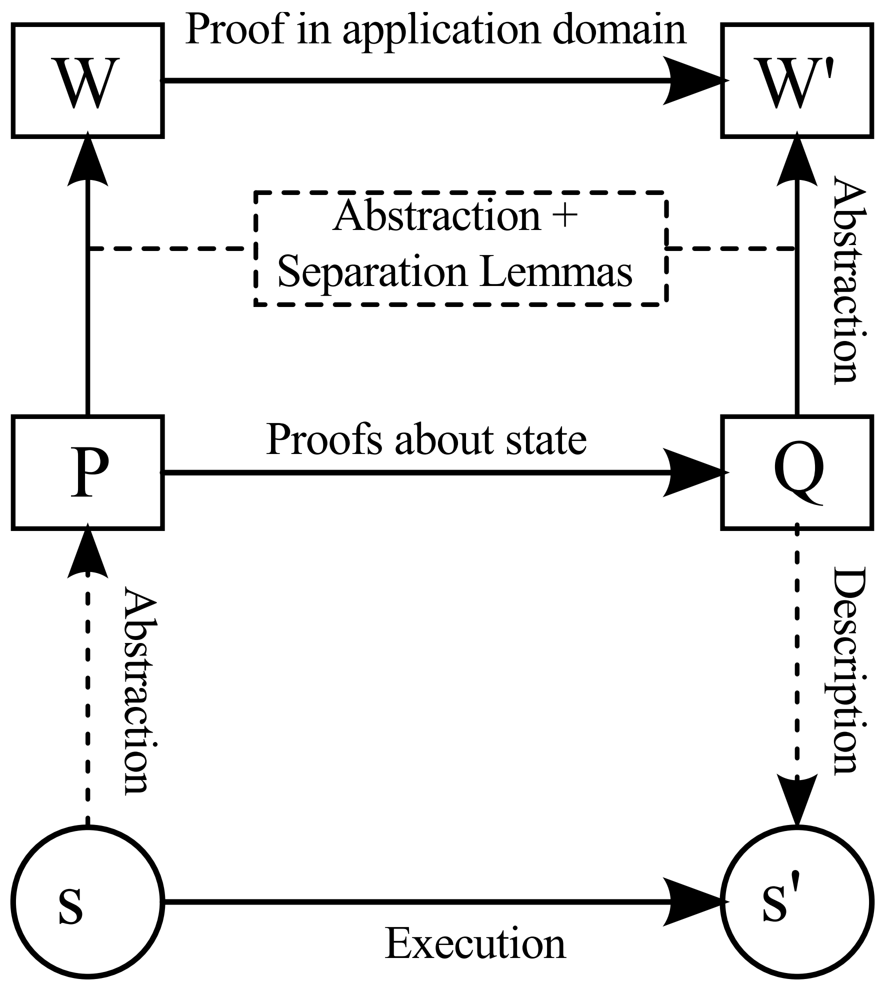
Abstraction



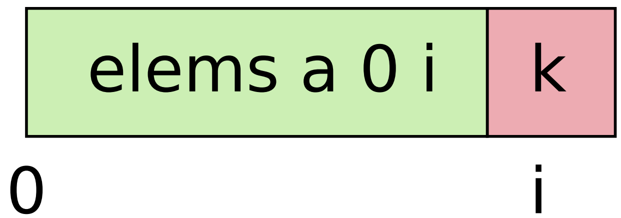
Definition: construct a list of all array elements

$$\text{elems}\ a\ i\ j \equiv \text{map}\ (\text{aget}\ a \circ \text{int})\ [\text{nat}\ i\ ..< \text{nat}\ j]$$

Correctness with Abstractions



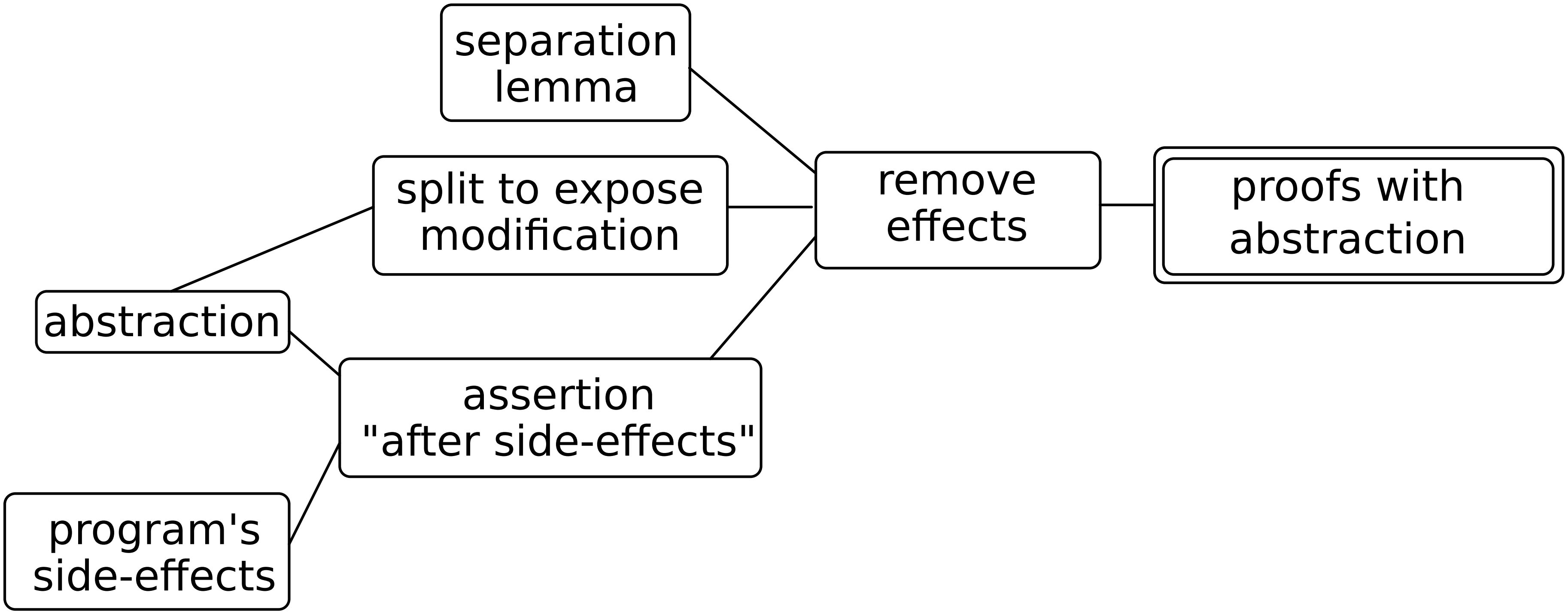
Separation Lemmas & Split



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

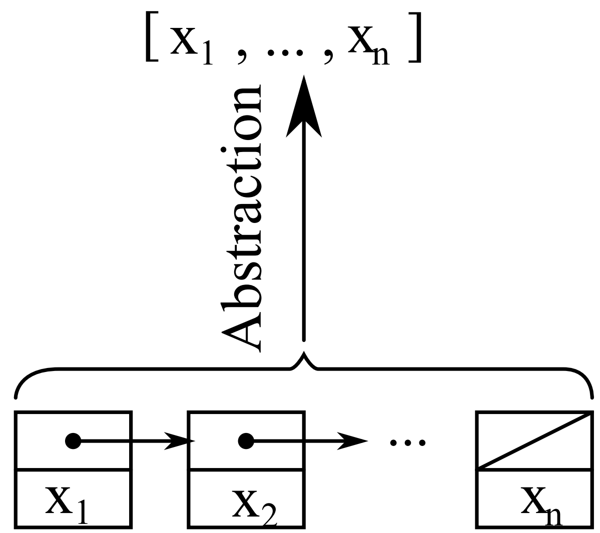
$$[\![\, 0 \leq i;\, i < j \,]\!] \Longrightarrow \text{elems}\,a\,i\,j = \mathbf{aget\,a\,i} \,\#\, \text{elems}\,a\,(i{+}1)\,j$$
$$[\![\, 0 \leq i;\, i < j \,]\!] \Longrightarrow \text{elems}\,a\,i\,j = \text{elems}\,a\,i\,(j\text{-}1)\,@\,[\,\mathbf{aget\,a\,(j\text{-}1)}\,]$$

Working with Abstractions



A Heap of Objects

Goal: Linked Lists on the Heap



- Linked list is stored on the heap using

```
struct node {
  int data;
  struct node *next;
}
```

- Interpret as sequence of `data` values

Basic Idea: Model the Memory

- So far: state = local variables
  - Access and modify by name, possibly index
  - No interaction between variables with different names
- Idea of "models"
  - Local variables are "really" values stored in memory
  - They behave as if they were stored in a map $\mathrm{name} \rightarrow \mathrm{value}$

  $\Rightarrow$ Model: state is tuple/record of current values
- Example: we can derive `x`=`x0` after

```
x0 = x;
y  = x + z;
```

Heaps in C

- Consider C definition

```
struct point { int x; int y; }
```

- Most programs are well-behaved (they use structs according to their definitions)

```
p->x = q->x + dx;
p->y = q->y + dy;
```

- However, some are more vicious

```
memcopy(p,q,sizeof(struct point));
```

  or

```
q  = &(p->y);
*q = y;
```

- Problem: pointer aliasing invalidates reasoning "by name"

How to Formalize the Heap?

- First approach: formalise the "reality"
  - Heap is mapping $\mathrm{addr} \to \mathrm{byte}$
  - Values map to byte sequences
  - Pointers are addresses
- This model enables low-level C programming
  - Pointer casts & untyped access
  - Pointer arithmetic & access to parts of heap object
- Problem: complexity of proofs
  - Cannot formulate separation lemmas directly
  - Prove non-aliasing / disjointness of memory regions [4, 5]

$\Rightarrow$ Later: separation logic [9, 8, 1, 10, 2, 6]

Towards a Simpler Memory Model

- Idea: restrict possible pointer operations
- . . . to obtain simpler proofs for the majority of programs
- Insight: in many programming languages (e.g. Java)
  - Pointers always point to the start of heap objects
  - Objects on the heap never overlap
  - Heap objects are records with known fields

$\Rightarrow$ Well-behaved access

```
p->x = q->x + dx;
p->y = q->y + dy;
```

Reasoning about Well-Behaved Programs

- Non-equal pointers refer to entirely different objects

```
x0   = q->x;
y0   = q->y;
p->x = 0;
```

  leaves `q->x`=`x0` and `q->y` = `y0` if `p`≠`q`.

- Fields with different names never overlap

```
x0   = q->x;
p->y = 0;
```

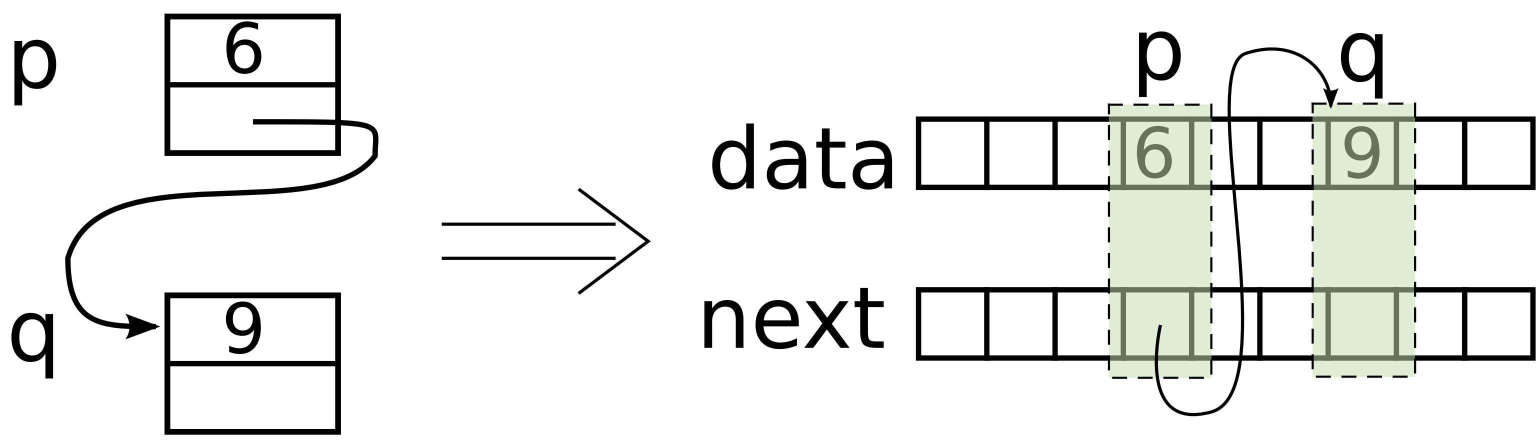
  leaves `q->x`=`x0` regardless of whether `p`=`q`.

- Idea
  - Different names of fields imply non-interference
  - For the same field, we can use inequality of pointers

Burstall's Idea

- Burstall's memory model [3]
  - Introduce an array for each known record field
  - Index the array by the object pointer
- Alternative names
  - Components-as-arrays
  - Split-heap model
- Enables just the above reasoning
  - Different names of fields imply non-interference
  - For the same field, we can use inequality of pointers

Core Idea



Defining Burstall's Model

- Basic definitions

$$\begin{aligned} \mathrm{NULL} &\equiv \mathrm{Addr}\,0 \\ \text{field-get}\,f\,p &\equiv f\,p \\ \text{field-upd}\,f\,p\,x &\equiv f\,(p := x) \end{aligned}$$

- Abbreviation

$$\text{"}p \rightarrow f\text{"} \equiv \text{"SimplC.field-get}\,f\,p\text{"}$$

- Interaction of `get`/`upd`

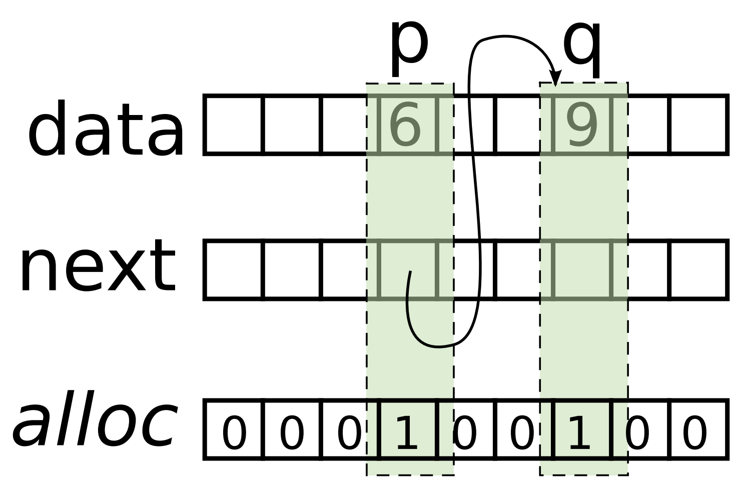
$$\begin{aligned} &p \neq q \Longrightarrow \text{field-get}\,(\text{field-upd}\,f\,p\,v)\,q = \text{field-get}\,f\,q \\ &\text{field-get}\,(\text{field-upd}\,f\,p\,v)\,p = v \end{aligned}$$

$\Rightarrow$ Reflects the behaviour of strongly typed languages

```
r0   = q->y;
p->x = 0;
r1   = q->y;
```

Allocatedness

- Runtime-errors in accessing objects
  - In Java etc.: accessing the `null` object
  - In C++: accessing a non-allocated object
- Model: introduce `bool`-field `alloc`
- Introduce guard `alloc` p



Setting Up the model

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

- SimplC introduces global arrays
- Keeps type information for type-checking

$\Rightarrow$ Can translate field accesses as array accesses as desired

Embedding by Example

- Read access `r = p->x;`

  Guard HeapAlloc $\{\!\!|\, \acute{}p \rightarrow \acute{}\text{tst-alloc} \,|\!\!\}$ (´r :== field-get ´tst-x ´p)

- Write access

  Guard HeapAlloc $\{\!\!|\, \acute{}p \rightarrow \acute{}\text{tst-alloc} \,|\!\!\}$
  (´tst-x :== field-upd ´tst-x ´p ´k)

Embedding – Almost Readable

```
| loop (EArrow (p, (TStruct s), f)) k =
  loop p
   (sGuard @{const HeapAlloc}
       (field_acc_cond ctx vctx s (Bound 0))
       (bind1 (field_get ctx vctx s f (Bound 0)) k))
| loop (EAssign (t, EArrow(p, TStruct s, f), e)) k =
  loop p
    (loop e
      (sSeq
        (sGuard @{const HeapAlloc}
                  (field_acc_cond ctx vctx s (Bound 1))
          (sBasic (field_put ctx vctx s f (Bound 1) (Bound 0))))
        (incr_bv (2,0,k))))
```

Evaluation: Basic Proofs

- $\{\ \mathrm{r0} = \mathrm{q}{\rightarrow}\mathrm{point\text{-}y}\ \}$ `p->x = 0;` $\{\mathrm{r0} = \mathrm{q\text{-}{>}point\text{-}y}\}$

  Proof obligation: fields with different names are disjoint

  $\mathrm{field\text{-}get\ point\text{-}y\ q} = \mathrm{point\text{-}y\ q}$

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

  Proof obligation: exclude aliasing

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

Linked Lists on the Heap

Defining Lists

- Standard definition: enumerate the nodes (following [7])

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

  - Start at `p` & end at `q` after finitely many steps
  - Burstall model: pass current `alloc` & `nxt` as parameters
  - All nodes are allocated and not `NULL`
- Beware: cycles are not excluded

⇒ An address may occur several times

Basic Properties of Lists

- Definition could be given equivalently as (z.B. [7])

  lemma list-alt:
  $$\text{"list alloc nxt p [] q} = (\text{p} = \text{q})\text{"}$$
  $$\text{"list alloc nxt p (x \# xs) q} = (\text{p} = \text{x} \wedge \text{alloc p} \wedge \text{p} \neq \text{NULL} \wedge \text{list alloc nxt (p} \rightarrow \text{nxt) xs q)"}$$

- Splitting the first element is possible for $p \neq q$

  lemma list-step:
  $$\text{"p} \neq \text{q} \Longrightarrow \text{list alloc nxt p xs q} = (\exists\, \text{xs'. xs} = \text{p \# xs'} \wedge \text{p} \neq \text{NULL} \wedge \text{alloc p} \wedge \text{list alloc nxt (field-get nxt p) xs' q)"}$$

- Using special case of $q = \texttt{NULL}$

  $$\text{"list alloc nxt p xs q} \Longrightarrow \text{NULL} \notin \text{set xs"}$$
  $$\text{"list alloc nxt NULL xs q} = (\text{xs} = [\,] \wedge \text{q} = \text{NULL})\text{"}$$
  $$\text{"list alloc nxt p (xs @ ys) q} = (\exists\, \text{r. list alloc nxt p xs r} \wedge \text{list alloc nxt r ys q)"}$$

Separation Lemma for Lists

- Again: formulate abstraction over fragments of lists

- Result depends only on nodes in the fragment

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

$\Rightarrow$ Proofs about disjointness of node/pointer sets

$\Rightarrow$ Will do this next time

Access a List's Data

definition
"list-data data xs = map (field-get data) xs"

lemma list-data-sep[simp]:
"$p \notin$ set xs $\Longrightarrow$ list-data (field-upd data p d) xs = list-data data xs"

lemma list-data-simps[simp]:
"list-data data [] = []"
"list-data data (x # xs) = field-get data x # list-data data xs"
"list-data data (xs @ ys) = list-data data xs @ list-data data ys"

- Refer back to enumeration of list nodes
- But look at the corresponding data fields instead
- Separation lemma same as for `list`

Example: Counting the List Nodes

define-struct {*. . . *as before*. . . *}

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;
}
```

Summary

- Today: introduction to the heap
- We need all previously introduced concepts
  - Shallow embedding
  - Abstraction predicates
  - Separation lemmas
- Then we get
  - Burstall's effective heap model
  - Abstractions for lists on the heap
- Next week: manipulating heap lists
  - List reversal
  - List insert

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] R.M. Burstall. Some techniques for proving correctness of programs which alter data stuctures. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, number 7. Edinburgh University Press, 1972.

[4] Holger Gast. Lightweight separation. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics 21st International Conference (TPHOLs 2008)*, volume 5170 of *LNCS*. Springer, 2008.

[5] Holger Gast. Reasoning about memory layouts. *Formal Methods in System Design*, 37(2-3):141–170, 2010.

[6] Andrew McCreight. Practical tactics for separation logic. 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.

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

[8] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, number 2142 in LNCS, pages 1–19. Springer, 2001.

[9] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 02)*, 2002.

[10] 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.