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

C/Java/... \(\xrightarrow{\text{translate}}\) Simpl \(\xrightarrow{\text{pass to}}\) VCG \(\xrightarrow{\text{generate}}\) Verification Conditions

outside Isabelle
not formal, not checked

Semantics

based on
take into account
Partial Correctness

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

Definition: construct a list of all array elements

\[
elems_{\text{ij}} \equiv \text{map} \ (\text{aget a} \circ \text{int}) \ [\text{nat i}..<\text{nat j}]
\]
Correctness with Abstractions

Proof in application domain

Abstraction + Separation Lemmas

Proofs about state

Execution

\( W \xrightarrow{\text{Abstraction}} W' \)

\( P \xrightarrow{\text{Abstraction}} Q \)

\( S \xrightarrow{\text{Execution}} S' \)
Separation Lemmas & Split

\[ k \notin \{i..<j\} \implies \text{elems}(\text{aset } k \ y \ a)_{ij} = \text{elems} \ a_{ij} \]

\[ \left[0 \leq i; i < j\right] \implies \text{elems} \ a_{ij} = \textbf{aget} \ a_{i} \ # \ \text{elems} \ a_{(i+1)j} \]

\[ \left[0 \leq i; i < j\right] \implies \text{elems} \ a_{ij} = \text{elems} \ a_{i(j-1)} @ [\textbf{aget} \ a{(j-1)}] \]
Working with Abstractions

- separation lemma
- split to expose modification
- remove effects
- proofs with abstraction
- abstraction
- assertion "after side-effects"
- program's side-effects
A Heap of Objects
Goal: Linked Lists on the Heap

- Linked list is stored on the heap using
  ```c
  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 name → value
⇒ Model: state is tuple/record of current values
- Example: we can derive x=x0 after

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

12 — H.Gast gasth@in.tum.de
Interactive Software Verification, 18.6.2013
Heaps in C

• Consider C definition
  
  ```c
  struct point { int x; int y; }
  ```

• Most programs are well-behaved (they use structs according to their definitions)
  
  ```c
  p->x = q->x + dx;
p->y = q->y + dy;
  ```

• However, some are more vicious
  
  ```c
  memcpy(p,q,sizeof(struct point));
  ```
  
  or
  
  ```c
  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 $\text{addr} \rightarrow \text{byte}$
  - 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]

⇒ 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

⇒ Well-behaved access

\[
\begin{align*}
p->x &= q->x + dx; \\
p->y &= q->y + dy;
\end{align*}
\]
Reasoning about Well-Behaved Programs

- Non-equal pointers refer to entirely different objects
  
  \[
  x_0 = q->x; \\
  y_0 = q->y; \\
  p->x = 0; \\
  \]
  leaves \( q->x = x_0 \) and \( q->y = y_0 \) if \( p \neq q \).

- Fields with different names never overlap
  
  \[
  x_0 = q->x; \\
  p->y = 0; \\
  \]
  leaves \( q->x = x_0 \) 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

\[ \begin{array}{c}
p \quad 6 \\
q \quad 9
data \quad 6 \quad 9 \\
next \quad |
\end{array} \]
Defining Burstall’s Model

- Basic definitions
  
  \[
  \begin{align*}
  \text{NULL} & \equiv \text{Addr 0} \\
  \text{field-get } f \ p & \equiv f \ p \\
  \text{field-upd } f \ p \ x & \equiv f (p := x)
  \end{align*}
  \]

- Abbreviation
  
  "p \to f" \equiv "\text{SimplC.field-get } f \ p"

- Interaction of get/upd
  
  \[
  \begin{align*}
  p \neq q \implies & \quad \text{field-get (field-upd } f \ p \ v) \ q = \text{field-get } f \ q \\
  & \quad \text{field-get (field-upd } f \ p \ v) \ p = v
  \end{align*}
  \]

⇒ Reflects the behaviour of strongly typed languages

\[
\begin{align*}
  r0 & = q->y; \\
  p->x & = 0; \\
  r1 & = q->y;
\end{align*}
\]
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

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

- SimplC introduces global arrays
- Keeps type information for type-checking
  ⇒ Can translate field accesses as array accesses as desired
Embedding by Example

• Read access \( r = p->x; \)
  
  Guard HeapAlloc \( \{ p \rightarrow \text{tst-alloc} \} (r := \text{field-get} \ \text{tst-x} \ \ p) \)

• Write access
  
  Guard HeapAlloc \( \{ p \rightarrow \text{tst-alloc} \} \)
  
  \( (\text{tst-x}:=\text{field-upd} \ \text{tst-x} \ p \ \ k) \)
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

- \( \{ r_0 = q \rightarrow \text{point-}y \} \ p \rightarrow x = 0 ; \{ r_0 = q \rightarrow \text{point-}y \} \)

  Proof obligation: fields with different names are disjoint
  \( \text{field-get point-}y \ q = \text{point-}y \ q \)

- \( \{ r_0 = q \rightarrow \text{point-}y \land \text{point-alloc} \ p \land p \neq q \} \ p \rightarrow y = 0 ; \{ r_0 = q \rightarrow \text{point-}y \} \)

  Proof obligation: exclude aliasing
  \( \text{field-get point-}y \ q = \text{field-get} \ (\text{field-upd point-}y \ p \ 0) \ q \)
Linked Lists on the Heap
Defining Lists

- Standard definition: enumerate the nodes (following [7])
  
  \textbf{inductive} list :: " alloc ⇒ nxt ⇒ addr ⇒ addr list ⇒ addr ⇒ bool"

  \textbf{where}
  
  emptyI: "[ p = q; xs = [] ] \implies \text{list alloc nxt p xs q}"

  consI: "[ alloc p; p \neq \text{NULL}; \text{list alloc nxt (p \rightarrow nxt) xs' q} ] \implies \text{list alloc nxt p (p \neq 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:
  
  "list alloc nxt p [] q = (p = q)"
  "list alloc nxt p (x # xs) q = (p = x ∧ alloc p ∧ p ≠ NULL ∧
  list alloc nxt (p → nxt) xs q)"

- Splitting the first element is possible for $p ≠ q$
  
  **Lemma** list-step:
  
  "p ≠ q ⇒
  list alloc nxt p xs q = (∃ xs’. xs = p ≠ xs’ ∧ p ≠ NULL ∧
  alloc p ∧ list alloc nxt (field-get nxt p) xs’ q)"

- Using special case of $q = \text{NULL}$
  
  "list alloc nxt p xs q \implies \text{NULL} \notin \text{set} xs"
  "list alloc nxt \text{NULL} xs q = (xs = [] ∧ q = \text{NULL})"
  "list alloc nxt p (xs @ ys) q = (∃ r. list alloc nxt p xs r ∧ 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]:
  \[ r \notin \text{set } xs \implies \text{list alloc (field-upd nxt r y) } p \times s \times q = \text{list alloc } \text{nxt } p \times s \times q \]

⇒ Proofs about disjointness of node/pointer sets

⇒ 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 ∉ set xs ⇔ 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

\[\text{define-struct \{*. . . as before. . . *\}}\]

pre: "list node-alloc node-next p XS NULL"
post: "nat n = length XS"

\[
n = 0;
/*@ \exists \text{ys. list node-alloc node-next p ys NULL} \land
\text{length XS} = \text{nat n + length ys} \land 0 \leq n */
\]

while (p != null) {
    n = n + 1;
    p = p->next;
}

30 — H.Gast gasth@in.tum.de
Interactive Software Verification, 18.6.2013
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;
}

31 — H.Gast gasth@in.tum.de "Interactive Software Verification, 18.6.2013"
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


