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

Proof in application domain

Abstraction + Separation Lemmas

Proofs about state

Execution

W → W'

P → Q

S → S'

Abstraction

Description

Interactive Software Verification, 25.6.2013
The Problem of Aliasing

\( p \rightarrow x \nabla y \quad \text{\&} \quad q \rightarrow x \nabla y \)

Proof obligation: exclude aliasing

\[
\{ r0 = q \rightarrow \text{point-y} \land \text{point-alloc } p \land p \neq q \} p \rightarrow y = 0 ; \{ r0 = q \rightarrow \text{point-y} \}
\]

\[
\text{field-get point-y } q = \text{field-get (field-upd point-y } p \ 0 \} q
\]
Separation Lemmas & Split

\[ k \not\in \{i..<j\} \implies \text{elems} (\text{aset} \ k \ y \ a) \ ij = \text{elems} \ a \ ij \]

\[ \checkmark \text{Lift (non-)aliasing proofs to abstractions} \]
Burstall’s Heap Model

\[ \text{data} \]

\[ \text{next} \]
Using Burstall’s heap model

\textbf{inductive} list :: ” alloc \Rightarrow \text{nxt} \Rightarrow \text{addr} \Rightarrow \text{addr list} \Rightarrow \text{addr} \Rightarrow \text{bool}”

\textbf{where}

emptyI: ” [ p = q; xs = [] ] \Rightarrow \text{list alloc \text{nxt} p \times s q”}

| consI: ” [ alloc p; p \neq \text{NULL}; \text{list alloc \text{nxt} (p \rightarrow \text{nxt}) \times s’ q} |

\text{\[ \Rightarrow \text{list alloc \text{nxt} p (p \neq \times s’) q”} \] \Rightarrow \text{list alloc \text{nxt} p (p \neq \times s’) q”} \}
Example: Counting the List Nodes

```ocaml
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

  \[ \text{lemma list-sep[simp]:} \]
  \[ r \notin \text{set } xs \implies \text{list alloc (field-upd } \text{nxt } r \text{ ) } p \times s q = \text{list alloc } \text{nxt } p \times s q \]

  \[ \Rightarrow \text{ Reduced aliasing problem to proving set properties} \]

  \[ \Rightarrow \text{ Isabelle's provers solve such proof obligations} \]
Example: List Reversal

```c
q = NULL;
while (p \neq NULL) {
    t = p->nxt;
    p->nxt = q;
    q = p;
    p = t;
}
```
Invariant of List Reversal

\[ \exists ys \ zs. \ \text{list alloc n}xt \ p \ zs \ \text{NULL} \land \ \text{list alloc n}xt \ q \ ys \ \text{NULL} \land \\
XS = \text{rev} \ ys @ zs \land \ \text{set} \ zs \cap \ \text{set} \ ys = \{\} \land \text{distinct} \ zs \]

- 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 \( q=\text{NULL} \), i.e. \( ys=[] \)
Preserving the Invariant

\[ \exists \ ys \ zs. \ \text{list alloc} \ \text{nxt} \ p \ zs \ \text{NULL} \ \land \ \text{list alloc} \ \text{nxt} \ q \ ys \ \text{NULL} \ \land \ XS = \text{rev} \ ys \ @ \ zs \ \land \ \text{set} \ zs \ \cap \ \text{set} \ ys = \{\} \ \land \ \text{distinct} \ zs \]

- 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 \( q \)

Before reading on, look at the "cheat sheet" on p. 7
Looking at the Single Steps

- Extract $p$ from the list abstraction

\[
[p \neq \text{NULL}; \exists xs'. zs = p \neq xs' \land \text{alloc } p \land \text{list alloc } \text{nxt } (\text{field-get } \text{nxt } p) \times s' \text{ NULL}; \\
\text{list alloc } \text{nxt } q \times y s \text{ NULL}; \text{set } zs \cap \text{set } ys = \{\}; \text{distinct } zs
\]
\[\implies \exists ysa \text{ zsa.} \\
\text{list alloc } (\text{field-upd } \text{nxt } p \times q) \text{ (field-get } \text{nxt } p) \times zsa \text{ NULL} \land \\
(\exists xs'. ysa = p \neq xs' \land \text{alloc } p \land \text{list alloc } (\text{field-upd } \text{nxt } p \times q) \times q \times s' \text{ NULL}) \land \\
\text{set } zsa \cap \text{set } ysa = \{\} \land \text{distinct } zsa \land \text{rev } ys @ zsa = \text{rev } ysa @ zsa
\]

- $\exists$-quantifiers are annoying (\text{clarsimp} & \text{exI})

\[
[p \neq \text{NULL}; \text{list alloc } \text{nxt } q \times y s \text{ NULL}; \text{alloc } p; \\
\text{list alloc } \text{nxt } (\text{field-get } \text{nxt } p) \times x s' \text{ NULL}; p \notin \text{set } y s; \text{set } xs' \cap \text{set } y s = \{\}; \\
p \notin \text{set } xs'; \text{distinct } xs'
\]
\[\implies \text{list alloc } (\text{field-upd } \text{nxt } p \times q) \text{ (field-get } \text{nxt } p) \times x s' \text{ NULL} \land \\
\text{list alloc } (\text{field-upd } \text{nxt } p \times q) \times y s \text{ NULL} \land \\
\text{set } xs' \cap \text{set } (p \neq y s) = \{\} \land \text{rev } y s @ p \neq xs' = \text{rev } (p \neq y s) @ x s'
\]

- Only remaining problem: side-effects
Rewriting with the Separation Lemma

- Simplify: $\text{list alloc (field-upd \, nxt \, p \, q) (field-get \, nxt \, p) \, xs' \, NULL}$

$\implies$ Pre-condition of $\text{list\_sep}$ yields goal $p \notin \text{set} \, xs'$

- 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 $\implies p \notin \text{set} \, xs$"
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 exl)
apply auto
done
```

- Only a little help needed on one ∃-quantifier

⇒ Can be get rid of this remaining piece of interaction?
Ghost Variables
Ghost-Variblen

- Idea: witnesses for \( \exists \)-quantifiers follow computation
  \[ \Rightarrow \text{Write down witnesses as “assignments”} \]
- 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
  \[ \Rightarrow \text{Will not be required during execution} \]
  - 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

\[ q = \text{null}; \]

\[ // @ ys = [] \]
\[ // @ zs = XS \]

/*@ list node-alloc node-next p zs \text{null} \wedge 
   list node-alloc node-next q ys \text{null} \wedge 
   \text{set} \text{zs} \cap \text{set} \text{ys} = \{\} \wedge \text{distinct} \text{zs} \wedge 
   XS = \text{rev} \text{ys} @ zs */

while (p != \text{null}) { 
  \[ t = p \rightarrow \text{next}; \]
  \[ p \rightarrow \text{next} = q; \]
  \[ q = p; \]
  \[ p = t; \]
  \[ // @ ys = (\text{hd} \text{zs}) \# \text{ys} \]
  \[ // @ zs = \text{tl} \text{zs} \]
}

\[ \ldots \]

by vcg (fastforce simp add: list-step)+  // that’s it!
Looking at the Details

• Loop updates ghost state
  
  list node-alloc node-next p zs NULL ∧
  list node-alloc node-next q ys NULL ∧

  ... 

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

• Compare to idea of code

![Diagram showing list nodes and pointers](image)
Example: Destructive List Append

```c
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: $\exists y_s. \text{list node-alloc node-next } p \ y_s \ \text{NULL} \land \cdots$

$\Rightarrow$ Must fill $y_s$ while proving

- Ghost state: "result" of reading a list from a heap is given

- Can simplify proving proving by classical reasoners

- Introduce rewrite rules

  \[
  \begin{align*}
  \text{list alloc } \text{nxt } p \ [ ] q &= (p = q) \\
  \text{list alloc } \text{nxt } p \ (x \neq xs) q &= (p = x \land \text{alloc} p \land p \neq \text{NULL} \land \text{list alloc } \text{nxt} \ (p \rightarrow \text{nxt}) \ xs \ q) \\
  \text{list alloc } \text{nxt } p \ (xs @ y_s) q &= (\exists \ r. \ \text{list alloc } \text{nxt } p \ xs \ r \land \text{list alloc } \text{nxt } r \ y_s \ q)
  \end{align*}
  \]
The Core Assertions

- **Precondition**
  
  \[ \text{list node-alloc node-next } p \text{ XS NULL} \land \text{list node-alloc node-next } q \text{ YS NULL} \]

- **Postcondition**
  
  \[ \text{list node-alloc node-next } r \text{ (XS @ YS) NULL} \]

- **Invariant**
  
  \[ \text{list node-alloc node-next } r \text{ xs1 } p \land \text{list node-alloc node-next } p \text{ xs2 NULL} \land \text{list node-alloc node-next } q \text{ YS NULL} \land \text{XS } = \text{ xs1 @ xs2 } \land p \neq \text{NULL} \]
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
  \[ p->\text{next} = q; \]
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


