Interactive Software Verification

Spring Term 2013

Holger Gast
gasth@in.tum.de

11.6.2013
Today

- Techniques
  - Abstractions predicates/functions
  - Separation lemmas

- Examples
  - Filling an array
  - Partition re-verified
  - Bubble sort
Remark: Semantics of Short-circuit Ops

- Short-circuit semantics of && and ||:
  Execute right-hand side only if lhs does not determine result
- Example: protect array access

  ```
  if (0<=i && i<n && a[i]>0) ...
  ```

- Embedding
  ```
  IF 0 ≤ i
      THEN IF i < n
          THEN Guard ”array-bounds” { acheck i a }
              (IF 0 < aget a i THEN i := 0 FI)
          FI
      FI
  ```

- Note: bounds-check protected by outer IF

  ⇒ Checked conditions available in proof
Remark: Semantics of Short-circuit Ops

- Proof obligation in example
  \[ \forall a \in \text{n. } [0 \leq i; i < \text{alength a}] \implies \text{acheck i a} \]

- Optimization: inline if no guards occur
  
  IF \(0 \leq i \land i < n\)
  THEN Guard "array-bounds" \{acheck \(i\ a\}\}
  (IF \(0 < \text{aget } a\ i\) THEN \(i := 0\) FI)
  FI
Recap: Strategies for Loop Invariants

- Generalize the desired post-condition
  - Loop test is \( a \neq b \), then post-condition from \( a = b \land I \)
  - If loop test is \( a < b \), then add \( a \leq b \) to invariant
- Describe (precisely) the achieved partial result
  - Consider the variables that are modified in the body
  - And capture the relationship between them
  - ... and the relationship to the input values (logical vars)
- Add safety assertions
  - About the range of index variables
  - About pointers not being null
- Preserve information from before loop
Example: Partition

- Desired post-condition
  
  \( (\forall k \in \{0..<i\}. \text{aget } a k \leq p) \land (\forall k \in \{i..<\text{alength } a\}. \text{aget } a k > p) \)

- Idea

<table>
<thead>
<tr>
<th>(\leq p)</th>
<th>?</th>
<th>(\geq p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
<td>j</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
</table>

- Invariant—main part
  
  \( (\forall k \in \{0..<i\}. \text{aget } a k \leq p) \land (\forall k \in \{j..<\text{alength } a\}. \text{aget } a k > p) \land \)

- Invariant—safety assertions
  
  \( 0 \leq i \land i \leq j \land j \leq \text{alength } a \land \)

- Invariant—preserved information
  
  \( n = \text{alength } a \)
Abstraction Predicates
Arrays So Far

- Statements about arrays so far low-level

- Example: loop invariant of partition

  \[
  (\forall k \in \{0..<i\}. a \text{get a } k \leq p) \land \\
  (\forall k \in \{j..<\text{alength a}\}. a \text{get a } k > p) \land \ldots
  \]

- View so far: arrays as—well—arrays

- \ldots but that’s not what we usually like to think about
Abstraction & Programming

• Crucial to programming: encapsulation & abstraction

• Library for hash-maps
  • Internally use arrays & buckets for collisions
  • Externally: pretend to represent int → int

⇒ Desirable: abstraction predicate
  hash-map a m
  • a is the array
  • m is the abstract represented map value

• Today: work out the issues with basic examples
Abstraction on Arrays

- Usual idea: array as representation for sequence
  - Indexing not really adequate
  - Talk about “sequence stored in array” instead
  - Example: sorted (elems a 0 N)

⇒ Define function \( \text{elems } a \text{ i } j \)
  - Yield ‘a sequence from ‘a array
  - Access slice \([i,j]\)

- Definition (for those who are curious/functionally inclined)

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

- Re-use notion & theory of \( \text{upto} \)
- Use array as a mapping from indizes to values
- Cast from \text{int} to \text{nat} where necessary
Verifying Array Initialization

- **First: version with explicit indices**

```c
i = 0;
/*@ 0 \leq i \wedge i \leq n \wedge n \leq alength a \wedge n = N \wedge
(\forall j \in \{0..<i\}. a[\text{get } a j] = k)
*/
while (i != n) {
a[i] = k;
i = i + 1;
}
```

- **Post-condition**

\[ \forall j \in \{0..<N\}. a[\text{get } a j] = k \]

- **Proof: straightforward** (because Isabelle knows about sets)
Creating a Sequence

- Read/use array initialization as sequence creation

- Abstracted post-condition
  \[
  \text{elems } a \ 0 \ N = \text{replicate } K \ N
  \]

- With previous loop-invariant: prove in the end
  \[
  \left[ 0 \leq n; n \leq \text{alength } a; \forall j \in \{0..<n\}. \text{aget } a \ j = k \right] \implies \text{elems } a \ 0 \ n = \text{replicate (nat } n \) k
  \]

- Requires some thought & lemma \text{map_replicate_const}: 
  \[
  \text{map (} \lambda x. \ k \xs = \text{replicate (length } \xs \) k
  \]
Re-phrasing the Invariant

- Use standard strategy: invariant is partial result towards desired post-condition

⇒ Yields loop invariant

\[ 0 \leq i \land i \leq n \land n \leq \text{alength a} \land n = N \land k = K \land \text{elems a 0 i = replicate (nat i) k} \]

⇒ Proof of post-condition trivial

- Preserving loop invariant yields goal

\[ \left[ 0 \leq i; i \leq n; n \leq \text{alength a}; \text{elems a 0 i = replicate (nat i) k}; i \neq n \right] \implies \text{elems (aset i k a) 0 (i + 1) = replicate (nat (i + 1)) k} \]
Issues in Proving the Invariant

\[ 0 \leq i; i \leq n; n \leq \text{alength a}; \\
\text{elems a 0 i} = \text{replicate (nat i) k}; \\
i \neq n \\
\] \implies \text{elems (aset i k a) 0 (i + 1)} = \text{replicate (nat (i + 1)) k}

- Interaction: abstraction\&modification
  
  \ldots \text{elems (aset i k a) 0 (i + 1)} \ldots

- Using the premises / old invariant / induction hypothesis

  \text{elems a 0 i} = \text{replicate (nat i) k}; \\
  \ldots \implies \text{elems (aset i k a) 0 (i + 1)} = \text{replicate (nat (i + 1)) k}

- Reasoning within the abstraction

  \text{replicate (nat (i + 1)) k pd} = \ldots \text{replicate i k} \ldots
Idea: Splitting the Abstraction

- Idea of the loop step

\[
\begin{array}{c}
\text{elems a 0 (i+1)} \\
\uparrow \\
\text{k ... k ... k} & \text{k} \\
0 & i
\end{array}
\]

⇒ Need to split the abstraction

\[
\begin{array}{c}
\text{elems a 0 (i+1)} \\
\rightarrow \\
\text{elems a 0 i} & a[i]
\end{array}
\]

- Achieved two objectives at once
  - Reasoning about place i seperately
  - Use loop invariant / induction hypothesis
Splitting `elems`

- General lemma: split `elems` at intermediate point `k`
  \[
  \left[ 0 \leq i; i \leq k; k \leq j \right] \implies \text{elems} a i j = \text{elems} a i k @ \text{elems} a k j
  \]

- Special cases: first & last element
  \[
  \left[ 0 \leq i; i < j \right] \implies \text{elems} a i j = \text{aget} a i \neq \text{elems} a (i+1) j
  \]
  \[
  \left[ 0 \leq i; i < j \right] \implies \text{elems} a i j = \text{elems} a i (j - 1) @ [ \text{aget} a (j - 1) ]
  \]

- Idea: programs usually modify sequence at beginning/end

- Heuristics: split at modification point (similar to case distinctions in `partition`)
  \[
  \left[ 0 \leq i; i \leq k; k < j \right] \implies \text{elems} (\text{aset} k x a) i j = \text{elems} a i k @ [ x ] @ \text{elems} a (k + 1) j
  \]
  Note: we know that at `k`, the value is `x`
Simplifying elems

- Anticipate special cases occurring in verification

- Empty sequence of elements before/end loop
  \[ \text{elems}_{a\ i\ i} = [] \]

- Singleton sequences (split off from beginning/end)
  \[
  \begin{align*}
  0 \leq i & \implies \text{elems}_{a\ i\ (i+1)} = [\text{aget}\ a\ i] \\
  0 < i & \implies \text{elems}_{a\ (i-1)\ i} = [\text{aget}\ a\ (i-1)]
  \end{align*}
  \]
Proceeding with the Proof

- **Original**
  
  \[ \text{elems}(\text{aset i k a}) \ 0 \ (i + 1) = \text{replicate}(\text{nat} \ (i + 1)) \ k \]

- **With \text{elems-split-last}**
  
  \[ \text{elems}(\text{aset i k a}) \ 0 \ i \ @ \ [k] = \text{replicate}(\text{nat} \ (i + 1)) \ k \]

- **Splitting the \text{replicate} to align (nat-add-distrib+replicate_append_same)**
  
  \[ \text{elems}(\text{aset i k a}) \ 0 \ i \ @ \ [k] = \text{replicate}(\text{nat} \ i) \ k \ @ \ [k] \]

- **Remains only**
  
  \[ \text{elems}(\text{aset i k a}) \ 0 \ i = \text{replicate}(\text{nat} \ i) \ k \]
Separation Lemmas

- Problematic: \( \text{elems}(\text{aset } i \ k \ a) \ 0 \ i = \text{replicate}(\text{nat } i \ k) \)

  \[
  \begin{array}{c|c}
  \text{elems } a \ 0 \ i & \text{k} \\
  \hline
  0 & i
  \end{array}
  \]

- Noted before: interaction read/update in proof obligations
- Had lemma to disregard irrelevant update
  \[
  i \neq j \implies \text{aget}(\text{aset } j \times a \ i) = \text{aget } a \ i
  \]
- Need similar lemmas for any introduced abstraction
- General idea: separation lemmas [1, 2]
- For the case of \( \text{elems} \)
  \[
  k \notin \{i..<j\} \implies \text{elems}(\text{aset } k \ y \ a \ i \ j) = \text{elems } a \ i \ j
  \]
Working with Abstractions

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

- Recall old formulation

<table>
<thead>
<tr>
<th>leq p</th>
<th>?</th>
<th>geq p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
<td>j</td>
</tr>
</tbody>
</table>

\[
(\forall k \in \{0 .. < i\}. \text{aget a } k \leq p) \land \\
(\forall k \in \{j .. < \text{alength a}\}. \text{aget a } k > p)
\]

- Proof was rather cumbersome, with case-distinctions

```plaintext
apply auto
apply (simp-all only: not-le not-less)
apply (case-tac "" k = j - 1"")
apply simp
apply simp
apply simp
apply (case-tac "" k < i"")
apply simp
apply simp
apply simp
```

... three more case distinctions...
Partition with Abstraction

- **Re-phrase invariant** (plus safety of indexes, as usual)
  \[
  (\forall x \in \text{set}(\text{elems} a 0 i). x \leq p) \land \\
  (\forall x \in \text{set}(\text{elems} a j (\text{alength} a)). x > p)
  \]

- **Proof-obligations have the form**
  \[
  \left[ \forall x \in \text{set}(\text{elems} a 0 i). x \leq p; \ldots \right. \\
  \quad \text{aget} a i \leq p; \\
  \quad x \in \text{set}(\text{elems} a 0 (i + 1)) \\
  \left. \right] \implies x \leq p
  \]

  ⇒ **Splitting off the last element does the trick (use fast)**
  \[
  \left[ \forall x \in \text{set}(\text{elems} a 0 i). x \leq p; \ldots \right. \\
  \quad \text{aget} a i \leq p; \\
  \quad x = \text{aget} a i \lor x \in \text{set}(\text{elems} a 0 i) \\
  \left. \right] \implies x \leq p
  \]

  ⇒ **Introduces case-distinction along abstraction & program**
Verifying Bubble-Sort
Bubble-Sort

```c
i = n;
while (i != 0) {
    i = i - 1;
    j = i;
    while (j < n - 1 && a[j + 1] < a[j]) {
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
        j = j + 1;
    }
}
```
Specification

- **Pre-condition as usual:** \( n = \text{alength} \ a \land n = N \land 0 \leq n \)

- **Invariant of outer loop**
  \[
  0 \leq i \land i \leq n \land n = N \land n = \text{alength} \ a \land \\
  \text{sorted} \ (\text{elems} \ a \ i \ N)
  \]

- **Invariant of inner loop** (imagine that without elems & explicit \( \forall \! \))
  \[
  0 \leq i \land i \leq j \land j < n \land n = N \land n = \text{alength} \ a \land \\
  \text{sorted} \ (\text{elems} \ a \ i \ j \ @ \ \text{elems} \ a \ (j + 1) \ n) \land \\
  (\forall x \in \text{set} \ (\text{elems} \ a \ i \ j). \ x \leq \text{aget} \ a \ j)
  \]

- **Note analogous “splitting” of sorted list by \text{sorted_append}**
  \[
  \text{sorted} \ (\text{xs}@\text{ys}) = (\text{sorted} \ \text{xs} \ & \ \text{sorted} \ \text{ys} \ & \ (\forall x \in \text{set} \ \text{xs}. \ \forall y \in \text{set} \ \text{ys}. \ x \leq y))
  \]
Proofs for Bubble-Sort

Interactively during the lecture, see Demo08.thy
Today

- Strategies for invariants by more examples
- Abstraction predicates/functions
- Separation lemmas
References