Interactive Software Verification

Spring Term 2013

Holger Gast
gasth@in.tum.de

4.6.2013
Today

- Shallow embedding of SimplC to Simpl
- Finally: start proving programs correct!
Recap
The General Plan

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

outside Isabelle
not formal, not checked

based on
take into account

Semantics
Partial Correctness in Simpl

- Definition of correctness
  \[ \Gamma \models P, C, Q, A \equiv \forall s. s \in \text{Normal } P \rightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow t \rightarrow t \in \text{Normal } Q \cup \text{Abrupt } A \]

- Verification logic: \( \Gamma, \Theta \vdash P, C, Q, A \)

- Soundness of VCG
  \[ \Gamma, \Theta \vdash P, C, Q, A \Rightarrow \Gamma, \Theta \models P, C, Q, A \]
Embedding SimplC into Simpl
Creating a Simpl Program

- Phases of the embedding
  1. Parser yields: abstract syntax tree (AST)
  2. Type-checker: annotates with types
  3. Translator: creates Simpl commands (needs type info)

- General considerations
  - Keeping track of “the current state”
  - Accessing / modifying the state
  - Translating primitive operations
  - Handling side-effects in expressions
  - Generating “safety checks”

- Presentation: by example, using Simpl external syntax
Reading and Writing Variables

- The assignment
  
  \[ j = i; \]

- Becomes
  
  \[ j := i \]

- Internally \( \approx \)
  
  \[ (\text{Basic}(\lambda s. \text{update-locals} \ j\_\prime (\text{lookup-locals} i\_\prime s) s)) \]

- Write to the local variable \( j \)
- The value obtained by reading local variable \( i \)
- (Naming convention post-fix \(_\prime\))
Primitive Operators

• The assignment

\[ j = i + 1; \]

• Becomes

\[ \texttt{j:== } \texttt{i+1} \]

• Internally (Groups.plus-class.plus is name of “+”)

\[
(\text{Basic}(\lambda s. \text{update-locals } j_\prime
(\text{Groups.plus-class.plus}(\text{lookup-locals } i_\prime s) 1)s)
\]

• For each primitive operator
  • Evaluate the two operands into HOL values
  • Then apply the corresponding HOL computation
A Tiny Theory of Arrays

• Wrap an array into a new datatype

  \texttt{datatype}'a\ \texttt{array} = \texttt{Array}\ "\texttt{int} \times (\texttt{int} \Rightarrow \ 'a)"

• Basic: modify / read array

  \texttt{amap}\ f\ a \equiv \texttt{case}\ a\ of\ \texttt{Array}\ a' \Rightarrow \texttt{Array}\ (f\ a')

  \texttt{aacc}\ f\ a \equiv \texttt{case}\ a\ of\ \texttt{Array}\ a' \Rightarrow f\ a'

• Then define read/write access (functionally)

  \texttt{aget}\ i\ a \equiv \texttt{aacc}\ (\lambda(n,c).c\ i)\ a

  \texttt{aset}\ i\ x\ a \equiv \texttt{amap}\ (\lambda(n,c).\langle n, c(i:={x})\rangle)\ a

• And a safety check

  \texttt{acheck}\ i\ a \equiv \texttt{aacc}\ (\lambda(n,c).0 \leq i \land i < n)\ a
A Tiny Theory of Arrays (II)

• Prove basic simplification rules (and declare simp)
  \( \text{ageti}(\text{aseti} \times a) = x \)
  \( i \neq j \implies \text{ageti}(\text{asetj} \times a) = \text{ageti}a \)
  \( \text{acheckia} = (0 \leq i \land i < \text{alengtha}) \)
  \( \text{alength}(\text{aseti} \times a) = \text{alengtha} \)

• Interaction of ageti and aseti (two cases)

• Unfolding acheck such that the simplifier proves it

• Interaction of alength and aseti
Safety Conditions in Simpl

- The array access
  \[ j = a[i]; \]

- Becomes
  Guard "array-bounds" \{ acheck \ i \ a \} ( \ j \ := \ aget \ i \ a )

- Evaluate acheck in the current state
  ⇒ User must prove that check returns true
- Then perform the modification (without further checks)
Keeping track of the State

- The assignment

\[ i = a[i = i + 1]; \]

- Becomes

\[ 'i \gg a_._ {\{ \text{achek (a_ + 1) \ 'a} \}} \]

- Read \( \gg \) as “bind” (derived from DynCom, left out in overview)
  - Read \( i \) in the current state, keep the result as \( a_._ \)
  - Update \( i \) to \( a_ + 1 \)
  - Then check safety of access (Seq, written ;;)
  - Finally modify \( i \)
Conditionals

• The conditional

```plaintext
if (i > 0) r = 1;
else r = 2;
```

• Becomes as expected (internal representation: Cond)

```plaintext
IF 0 < ´i THEN ´r:=:= 1 ELSE ´r:=:= 2 FI
```

• Combination with state updates is possible

```plaintext
if ((i = i - 1) > 0) r = 1;
else r = 2;
```

• Becomes

```plaintext
´i ≡ a_

(´i:=:= a_-1;; IF 0 < a_-1 THEN ´r:=:= 1 ELSE ´r:=:= 2 FI)
```
While Loops

- Problem: side-effects in the while expression
  
  ```
  while ((i = i - 1) >= 0) j = j + 1;
  ```

- Simpl’s While does not support this directly

- Trick: use `While True` and break by `Throw`

  ```
  TRY WHILE True
  DO  `i >>= a_.
      (`i :== a_-1;;
       IF 0 <= a_-1 THEN `j :== `j + 1
       ELSE `thrown_ :== ThrowBreak;; THROW FI)
  OD
  CATCH IF `thrown_ = ThrowBreak THEN SKIP ELSE THROW FI END
  ```

- Keep exception in local variable `thrown_`

- Catch only this kind of exception
Alternative Embedding of Loops

while ((i = i - 1) >= 0) j = j + 1;

- Unwind the execution of side-effects

\[
\begin{align*}
\text{\texttt{\textbackslash{}i} & \gg \texttt{a}. \\
(\texttt{\textbackslash{}i}:==\texttt{a}-1; \\
\texttt{\textbackslash{}test}:==0 \leq \texttt{a}-1)\
\texttt{WHILE} \texttt{\textbackslash{}test} \texttt{DO} \\
\texttt{\textbackslash{}j}:==\texttt{\textbackslash{}j}+1 \\
\texttt{\textbackslash{}i} & \gg \texttt{a}. \\
(\texttt{\textbackslash{}i}:==\texttt{a}-1; \\
\texttt{\textbackslash{}test}:==0 \leq \texttt{a}-1)\
\texttt{OD}
\end{align*}
\]

- Yields equivalent verification conditions

- (Aesthetic) Disadvantage: duplication of code / no 1–1 correspondence between code and embedding
Done!

- Given: source language SimplC

- Standard compiler front-end
  - Parse using lex/yacc
  - Type-check

- Define semantics by translation to Simpl

⇒ SimplC “inherits” semantics of Simpl

- Now forward: generate VCs for the Simpl program!
Using the VCG
A Basic Example

verify-statement swap
vars: {*
  int i;
  int j;
  int t;
*}
pre: "i = I \land j = J"
post: "i = J \land j = I"
{* 
  t = i;
  i = j;
  j = t;
*}
apply vcg
apply simp
done
Logical Variables

• Note free variables I and J
  
  pre: ”i = I ∧ j = J”
  post: ”i = J ∧ j = I”

⇒ logical variables (or auxiliary variables)
  
  • Connect initial and final state
  ⇒ Specification as input/output-relation
  
  • Goal: post-condition refers to result vars & logical vars
  
  • Generated goal/triple for example

\[ \forall J \forall I. \Gamma \vdash \{ \mathbf{\delta i = I} \land \mathbf{\delta j = J} \} \]
\[ \mathbf{\delta t := \delta i ; ( \delta i := \delta j ; \delta j := \delta t) } \]
\[ \{ \mathbf{\delta i = J} \land \mathbf{\delta j = I} \} \]
Mastering Loops

• Goal: multiply $i$ and $j$ by summation

r = 0;
while (i > 0) {
    r = r + j;
    i = i - 1;
}

• The while rule was

\[
\frac{P \implies I}{\{ I \land t \} \ b \ { I \}}
\]

\[
I \land \neg t \implies Q
\]

\[
\frac{\{ P \} \ while (t) \ b \ { Q \}}
\]

• We need to “guess”/“invent” a suitable invariant
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
Verifying Multiplication

\[ r = 0; \]
\[ \text{while } (i > 0) \{ \]
\[ \quad r = r + j; \]
\[ \quad i = i - 1; \]
\[ \} \]

- Changing variables \( r, i \)
- Initial value of \( i \) is \( I \)

\[ \Rightarrow \text{What are the relationships?} \]
\[ \Rightarrow \text{What is the current partial result?} \]
- Look hard at the code, think a bit
- See: \( j \) has been added a few times to \( r \)
- More precisely: \( I - i \) times!

\[ r = (I - i) \times j; \]
Copying an Array

```c
i = 0;
while (i != n) {
    dst[i] = src[i];
    i = i + 1;
}
```

- **Post-condition**: all elements have been copied
- **Changing variables**: $i$, $src$, $dst$
- **Invariant**: partial result is “copy up to $i$”
  \[
  \forall 0 \leq j < i. \, dst[j] = dst[j]
  \]

$\Rightarrow$ With $i = n$ after end of loop matches post-condition
- **Safety-condition**: $i$ is legal index in both arrays
  $\Rightarrow$ Include into pre-condition
Summing Up the Array Elements

\[
\begin{align*}
s &= 0; \\
i &= 0; \\
\textbf{while} \ (i \neq n) \ {\{} \\
&\quad s = s + a[i]; \\
&\quad i = i + 1; \\
{\}}
\end{align*}
\]

- Post-condition: \( s = (∑ k=0..<N. a\text{get } k \ a) \)
- Inventing the invariant
  - Changing variables: \( i, s \)
  - Partial result: sum up to \( i \) (excluded) in \( s \)
    \[
    s = (∑ k=0..<i. a\text{get } k \ a)
    \]
- Side-conditions for proper array access
  \[
  0 \leq i \land i \leq n \land n = a\text{length } a
  \]
Verifying Summation

\[ s = 0; \ i = 0; \]
\[ \text{/*@ 0} \leq i \land i \leq n \land n = \text{alength a} \land n = N \land \]
\[ s = \left( \sum_{k=0}^{<i} a \cdot \text{aget k a} \right) */ \]

\text{while} (i \neq n) \{ \]
\[ s = s + a[i]; \]
\[ i = i + 1; \]
\} \]
\text{apply vcg} \]
\text{apply auto} \]
\text{apply} (\text{simp add: ivl-split-last}) \]
\text{done} \]

- Pre-condition: bind logical variables & safety

\[ n = \text{alength a} \land n = N \land 0 \leq n \]

- Need one insight (by looking at the goals; lemma ivl-split-last)

\[ i \leq j \implies \{ (i::\text{int})..<j+1 \} = \{ i..<j \} \cup \{ j \} \]
Wrapping Up

- We have gone the whole way
  - Define syntax & semantics of Simpl
  - Parse language SimplC
  - Translate SimplC to Simpl as shallow embedding
    ⇒ Defines the semantics of SimplC
  - Generate verification conditions
    ⇒ Program correctness by soundness of VCG
- Embedding
  - Mostly straightforward
  - Keep track of “current” state
  - Side-effects in the while condition
- Basics of verification: logical variables & loop invariants