Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

4.6.2013

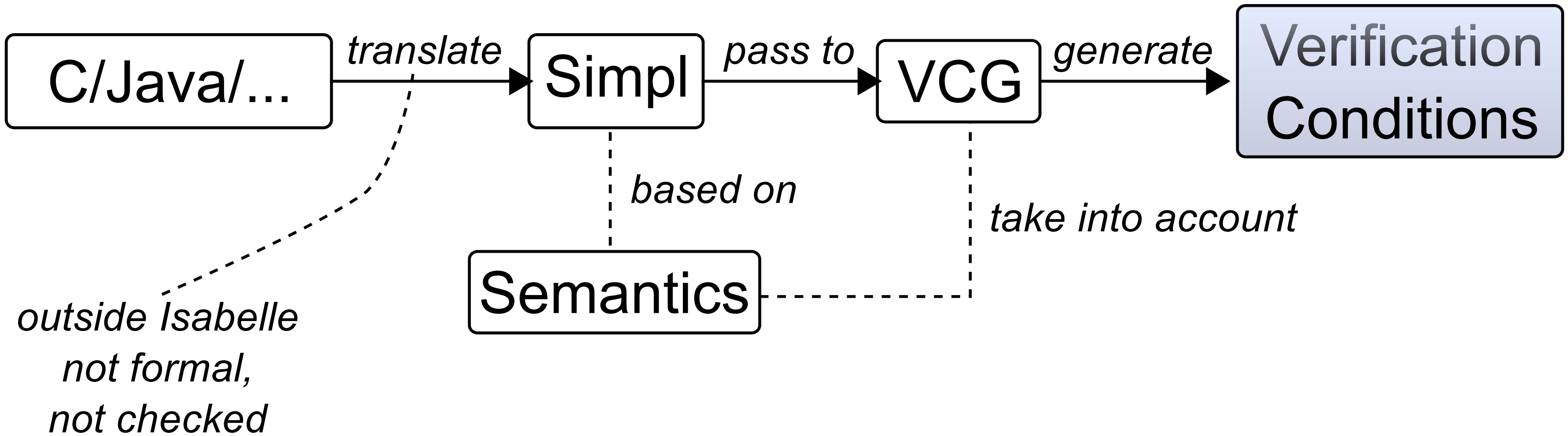
Today

- Shallow embedding of SimplC to Simpl

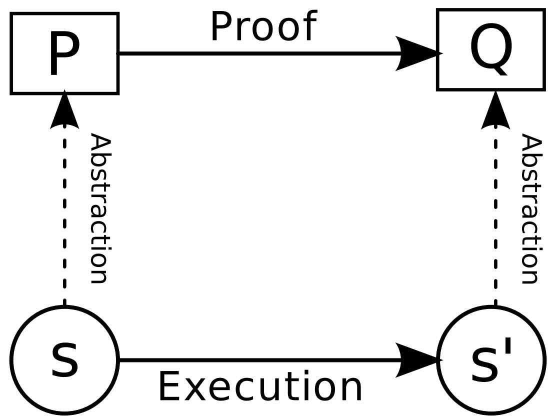
- Finally: start proving programs correct!

Recap

The General Plan



Partial Correctness in Simpl



- Definition of correctness

$$\Gamma \models P\, c\, Q,A \equiv$$
$$\forall s\, t.\;\; s \in \mathrm{Normal}\,`\,P \longrightarrow$$
$$\Gamma \vdash \langle c,s \rangle \Rightarrow t \longrightarrow$$
$$t \in \mathrm{Normal}\,`\,Q \cup \mathrm{Abrupt}\,`\,A$$

- Verification logic: $\Gamma,\Theta \vdash P\, c\, Q,A$
- Soundness of VCG

$$\Gamma,\Theta \vdash P\, c\, Q,A \quad \Longrightarrow \quad \Gamma,\Theta \models P\, c\, Q,A$$

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 $\approx$

  (Basic (λs. *update-locals* `j_'` (*lookup-locals* `i_'` s) s)

  - Write to the local variable `j`
  - The value obtained by reading local variable `i`
  - (Naming convention post-fix `_'`)

Primitive Operators

- The assignment

```
j = i + 1;
```

- Becomes

  $\acute{}\mathsf{j} :== \acute{}\mathsf{i} + 1$

- Internally (`Groups.plus-class.plus` is name of "+")

  $(\mathsf{Basic}\,(\lambda \mathsf{s}.\ \mathit{update\text{-}locals}\ \mathtt{j\_'}$
  $(\mathsf{Groups.plus\text{-}class.plus}\,(\mathit{lookup\text{-}locals}\ \mathtt{i\_'}\ \mathsf{s})\,1)\,\mathsf{s})$

- 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

  $\textbf{datatype}\ \text{'a array} = \text{Array}\ \text{"int} \times (\text{int} \Rightarrow \text{'a)"}$

- Basic: modify / read array

  $\text{amap f a} \equiv \textbf{case}\ \text{a}\ \textbf{of}\ \text{Array a'} \Rightarrow \text{Array (f a')}$
  $\text{aacc f a} \equiv \textbf{case}\ \text{a}\ \textbf{of}\ \text{Array a'} \Rightarrow \text{f a'}$

- Then define read/write access (functionally)

  $\text{aget i a} \equiv \text{aacc}\ (\lambda(\text{n,c}).\ \text{c i})\ \text{a}$
  $\text{aset i x a} \equiv \text{amap}\ (\lambda(\text{n,c}).\ (\text{n, c(i := x)}))\ \text{a}$

- And a safety check

  $\text{acheck i a} \equiv \text{aacc}\ (\lambda(\text{n,c}).\ 0 \leq \text{i} \wedge \text{i} < \text{n})\ \text{a}$

A Tiny Theory of Arrays (II)

- Prove basic simplification rules (and declare `simp`)

  $$\mathsf{aget}\,\mathsf{i}\,(\mathsf{aset}\,\mathsf{i}\,\mathsf{x}\,\mathsf{a}) = \mathsf{x}$$

  $$\mathsf{i} \neq \mathsf{j} \Longrightarrow \mathsf{aget}\,\mathsf{i}\,(\mathsf{aset}\,\mathsf{j}\,\mathsf{x}\,\mathsf{a}) = \mathsf{aget}\,\mathsf{i}\,\mathsf{a}$$

  $$\mathsf{acheck}\,\mathsf{i}\,\mathsf{a} = (0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{alength}\,\mathsf{a})$$

  $$\mathsf{alength}\,(\mathsf{aset}\,\mathsf{i}\,\mathsf{x}\,\mathsf{a}) = \mathsf{alength}\,\mathsf{a}$$

- Interaction of `aget` and `aset` (two cases)
- Unfolding `acheck` such that the simplifier proves it
- Interaction of `alength` and `aset`

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

  $\Rightarrow$ 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

  $$\acute{} \mathrm{i} \gg \mathtt{a\_}.\\ \quad (\acute{}\mathrm{i} :== \mathtt{a\_} + 1;;\\ \quad \mathrm{Guard}\ \text{"array-bounds"}\ \{\!|\mathrm{acheck}\,(\mathtt{a\_} + 1)\ \acute{}\mathrm{a}|\!\}\\ \quad (\acute{}\mathrm{i} :== \mathrm{aget}\,(\mathtt{a\_} + 1)\ \acute{}\mathrm{a}))$$

- 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

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

- Becomes as expected (internal representation: `Cond`)

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

- Combination with state updates is possible

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

- Becomes

  ´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

$$
\begin{array}{l}
\acute{}\mathrm{i} \gg \mathtt{a\_}. \\
\quad (\acute{}\mathrm{i} :== \mathtt{a\_} - 1;; \\
\quad\ \ \acute{}\mathrm{test\_} :== 0 \le \mathtt{a\_} - 1) \\
\mathrm{WHILE}\ \acute{}\mathrm{test\_}\ \mathrm{DO} \\
\quad \acute{}\mathrm{j} :== \acute{}\mathrm{j} + 1 \\
\quad \acute{}\mathrm{i} \gg \mathtt{a\_}. \\
\quad\quad (\acute{}\mathrm{i} :== \mathtt{a\_} - 1;; \\
\quad\quad\ \ \acute{}\mathrm{test\_} :== 0 \le \mathtt{a\_} - 1) \\
\mathrm{OD}
\end{array}
$$

- 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

$\Rightarrow$ 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 ∧ j = J"
post: "i = J ∧ j = I"
{*
 t = i;
 i = j;
 j = t;
*}
 apply vcg
 apply simp
 done
```

Logical Variables

- Note free variables `I` and `J`

  pre: "$i = I \wedge j = J$"
  post: "$i = J \wedge j = I$"

$\Rightarrow$ logical variables (or auxiliary variables)

- Connect initial and final state

$\Rightarrow$ Specification as input/output-relation

- Goal: post-condition refers to result vars & logical vars
- Generated goal/triple for example

$$\forall J\, I.\ \Gamma \vdash \{\!|\ ´i = I \wedge ´j = J |\!\}\ ´t :== ´i;; (´i :== ´j;; ´j :== ´t)\ \{\!|\ ´i = J \wedge ´j = I |\!\}$$

Mastering Loops

- Goal: multiply `i` and `j` by summation

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

- The while rule was

$$\frac{P \implies I \qquad \{\ I \wedge t\ \}\ b\ \{\ I\ \} \qquad I \wedge \neg t \implies Q}{\{\ P\ \}\ \textsf{while}\ (t)\ b\ \{\ Q\ \}}$$

- 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 $a = b \wedge I$
  - 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;
while (i > 0) {
  r = r + j;
  i = i - 1;
}
```

- Changing variables `r`, `i`
- Initial value of `i` is `I`

⇒ What are the relationships?

⇒ 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!

$$r = (I - i) * j;$$

Copying an Array

```
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"

$$\forall 0 \leq j < i.\ \mathtt{dst}[j] = \mathtt{dst}[j]$$

$\Rightarrow$ 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

```
s = 0;
i = 0;
while (i != n) {
  s = s + a[i];
  i = i + 1;
}
```

- Post-condition: $s = (\sum k = 0..<N.\ aget\ k\ a)$
- Inventing the invariant
  - Changing variables: `i`, `s`
  - Partial result: sum up to `i` (excluded) in `s`

$$s = (\sum k = 0..<i.\ aget\ k\ a)$$

  - Side-conditions for proper array access

$$0 \leq i \wedge i \leq n \wedge n = alength\ a$$

Verifying Summation

```
s = 0; i = 0;
/*@ 0 ≤ i ∧ i ≤ n ∧ n = alength a ∧ n = N ∧
          s = (∑ k=0..<i. aget k a) */
while (i != n) {
 s = s + a[i];
 i = i + 1;
}
apply vcg
apply auto
apply (simp add: ivl-split-last)
done
```

- Pre-condition: bind logical variables & safety

$$n = \text{alength}\ a \wedge n = N \wedge 0 \leq n$$

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

$$i \leq j \Longrightarrow \{(i::int)..<j+1\} = \{i..<j\} \cup \{j\}$$

Wrapping Up

- We have gone the whole way
  - Define syntax & semantics of Simpl
  - Parse language SimplC
  - Translate SimplC to Simpl as shallow embedding

  $\Rightarrow$ Defines the semantics of SimplC
  - Generate verification conditions

  $\Rightarrow$ 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