Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

7.5.2013

The Big Picture

Correctness of Programs
Solve verification conditions Arguments about application domain
Hoare Logic
Verification rules for language constructs Generator for verification conditions
Semantics
Define meaning of programs Describe behaviour of programs

Today

- Basis: Abstract Syntax & Semantics

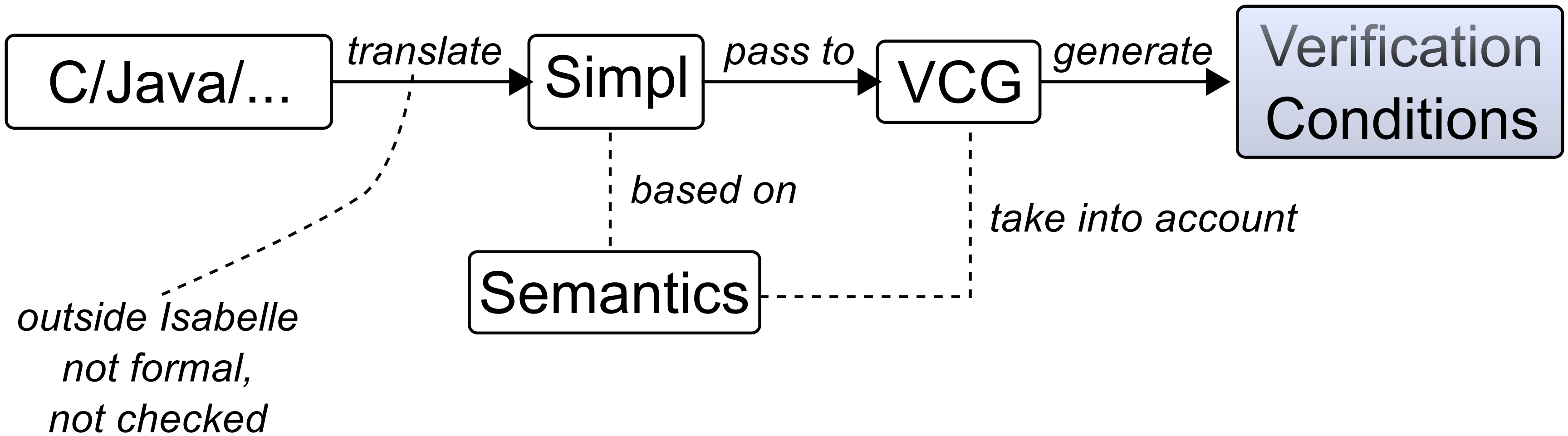
- Using the example of Simpl [5, 4]
  A Sequential Imperative Programming Language
  - Prototypical language
  - Leaves "holes" to be filled by concrete languages

  ⇒ Re-usable development

- Goal: apply Simpl framework to SimplC dialect

- Formal material taken from [4]

Simpl — Idea



- Define Semantics by translation to given language
- Analogy: "C means what the compiler makes of it"
- VCG = Verification Condition Generator

$\Rightarrow$ Programm is correct if conditions can be proven

Inductively-Defined Predicates

Recap: Tail Recursion as Iteration

datatype instr =
 Init int | Add int | Mul int | Jump string

type-synonym state = int
type-synonym block = ''instr list''

fun exec :: ''state ⇒ instr list ⇒ state''
where ''exec s [] = s''
| ''exec s (Init i # xs) = exec i xs''
| ''exec s (Add i # xs) = exec (s + i) xs''
| ''exec s (Mul i # xs) = exec (s * i) xs''

- Abstract syntax of program: datatype
- Iterative execution by case distinction & tail recursion

Adding Jumps

```
type-synonym prog = "string ⇀ block"
function execp :: "prog ⇒ state ⇒ instr list ⇒ state"
where "execp p s [] = s"
| "execp p s (Init i # xs) = execp p i xs"
| "execp p s (Add i # xs) = execp p (s + i) xs"
| "execp p s (Mul i # xs) = execp p (s * i) xs"
| "execp p s (Jump l # xs) =
  (case p l of
   None ⇒ s
   | Some xs ⇒ execp p s xs)"

termination execp
 apply (relation "measure (λ(p,s,xs). length xs)")
 apply auto[4]
```

Problem: cannot bound the length of remaining computation

$$\bigwedge p\, l\, xs\, a.\ p\, l = \mathrm{Some}\, a \Longrightarrow \mathrm{length}\, a < \mathrm{Suc}\,(\mathrm{length}\, xs)$$

Solution: Execution as Relation

```
inductive execpp :: "prog ⇒ state ⇒ instr list ⇒ state ⇒ bool"
where "s' = s        ⟹ execpp p s [] s'"
 | "execpp p i xs s'    ⟹ execpp p s (Init i # xs) s'"
 | "execpp p (s + i) xs s' ⟹ execpp p s (Add i # xs) s'"
 | "execpp p (s * i) xs s' ⟹ execpp p s (Mul i # xs) s'"
 | " ⟦ case p l of
        Some b ⇒ execpp p s b s'
      | None    ⇒ s' = s
   ⟧ ⟹ execpp p s (Jump l # xs) s'"
```

- Use relation instead of function
- Execution = prove relation from `execpp.intros`
- Example:

```
lemma "execpp ["b" ↦ [Mul 3]] 14 [Jump "b" ] 42"
 apply (rule execpp.intros)
 apply (simp). . .
```

In case you are wondering . . .

- Of course, execution is still deterministic
- Prepare for case distinctions on executed instruction

```
inductive_cases [elim!]: "execpp p s [] s'"
inductive_cases [elim!]: "execpp p s (Init i # xs) s'"
inductive_cases [elim!]: "execpp p s (Add i # xs) s'"
inductive_cases [elim!]: "execpp p s (Mul i # xs) s'"
inductive_cases [elim!]: "execpp p s (Jump l # xs) s'"
```

- Now we can prove (by induction and subsequent case distinction)

```
lemma execpp-determistic:
 "⟦ execpp p s xs s'1; execpp p s xs s'2 ⟧ ⟹ s'1 = s'2"
 apply (induct arbitrary: s'2 rule: execpp.induct)
 apply (force intro: execpp.intros split: split-if-asm)+
 done
```

Summary: `inductive`

- User gives introduction rules for new predicate
  - Which values are to be related?
  - All other relationships are excluded (least fixed point construction)
- `inductive` defines constant as least fixed point and
- . . . proves theorems
  - P`.intros`: the given introduction rules
  - P`.induct`: induction over rule applications
  - P`.cases`: case distinction by applied rules
- Command `inductive_cases`
  - Specialized `.cases` by datatype constructors
  - Useful for proving statements about program execution

Syntax of Simpl

Simpl – Syntax

- Simpl is a "prototypical" imperative language
  - Statements
  - Control flow
  - Procedure calls
- Simpl is general
  - Abstracts over concrete language constructs

  $\Rightarrow$ Can be used to express several languages
- Today: abstract syntax ($\approx$ parse trees)
- Later: concrete syntax

Simpl – Syntax (selected & types simplified)

- Statements (Commands)

  datatype com =
  Skip
  | Basic "state ⇒ state"
  | Seq com com
  | Cond bexp com com
  | While bexp com

- Non-local control flow (`throw`, `catch`, `return`, `continue`, `break`, function call, . . .)

  | Call "'p"
  | Throw
  | Catch com com

- Side-conditions / run-time errors

  | Guard bexp com

Shallow Embedding

- Several constructs are "completely general"

  Basic "state ⇒ state"
  Cond bexp com com
  Guard bexp com

- First: boolean tests

  type-synonym bexp = "state set"

- Note: a 'a set is as good as a 'a ⇒ bool (see `Collect` in `Set.thy`)
- Shallow embedding: express semantics directly in HOL
- Example: if-test `i > 0` becomes approx.

$$\{s \mid \texttt{lookup}\ s\ \texttt{i} > 0\}$$

- Example: assignment `i=i+1;` becomes

$$(\lambda s.\ \texttt{update}\ \texttt{i}\ (\texttt{lookup}\ s\ \texttt{i}\ +\ 1)\ s)$$

Comparison Shallow / Deep Embedding

- Shallow embedding
  - Isabelle can already reason about HOL formulae
  - Need not do everything inside HOL
  - Can take short-cuts
  - Extensions simple
- Deep embedding (i.e. with explicit AST)
  - Less work outside Isabelle (only parser)
  - Can reason about language structure
    - Meta-properties such as type-correctness
    - Equivalence of different syntactic expressions
  - Must implement specialized reasoning mechanisms
- For details see [7]

Semantics: Overall Structure

- Distinguish between different outcomes of execution

  datatype xstate = Normal state | Abrupt state | Fault | Stuck

  - Normal termination with final state
  - Abrupt termination by `Throw` (with final state)
  - Violated guard (= runtime check)
  - Stuck (undefined functions, . . .)
- Table of defined functions

  type-synonym body = "procName ⇒ com option"
- Then have usual inductive definition

  inductive
  exec :: "[body,com,xstate,xstate] ⇒ bool"
  ("-⊢ ⟨-,-⟩ ⇒ -" [60,20,98,98] 89)

The Straightforward Commands

- Skip does nothing to the state

  $\Gamma \vdash \langle \text{Skip}, \text{Normal}\ s \rangle \Rightarrow \text{Normal}\ s$

- A sequence is executed sequentially

  $\llbracket\ \Gamma \vdash \langle c_1, \text{Normal}\ s \rangle \Rightarrow s';$
  $\Gamma \vdash \langle c_2, s' \rangle \Rightarrow t$
  $\rrbracket \Longrightarrow \Gamma \vdash \langle \text{Seq}\ c_1\ c_2, \text{Normal}\ s \rangle \Rightarrow t$

The General `Basic` Command

- Definition is simple

  $\Gamma \vdash \langle \mathsf{Basic}\, f, \mathsf{Normal}\, s\rangle \Rightarrow \mathsf{Normal}\, (f\, s)$

  - Pass current state to function `f`
  - Treat result of function as new state

  ⇒ `f` is a general "state update function"
- The resulting behaviour is (rather) astonishing:

  `f` can perform any (terminating) computation on state, e.g.
  - Lookup & update local variables & the heap
  - Allocate / free memory
- Can be used to encode many things
  - `i = i + 1;`
  - `i = (j++) + (k++);`

The Conditional

- A conditional depends on the outcome of the test

$$
\begin{array}{l}
[\![\mathsf{s} \in \mathsf{b}; \\
\ \Gamma \vdash \langle \mathsf{c}_1, \mathsf{Normal}\ \mathsf{s} \rangle \Rightarrow \mathsf{t} \\
]\!] \implies \Gamma \vdash \langle \mathsf{Cond}\ \mathsf{b}\ \mathsf{c}_1\ \mathsf{c}_2, \mathsf{Normal}\ \mathsf{s} \rangle \Rightarrow \mathsf{t}
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{s} \notin \mathsf{b}; \\
\ \Gamma \vdash \langle \mathsf{c}_2, \mathsf{Normal}\ \mathsf{s} \rangle \Rightarrow \mathsf{t} \\
]\!] \implies \Gamma \vdash \langle \mathsf{Cond}\ \mathsf{b}\ \mathsf{c}_1\ \mathsf{c}_2, \mathsf{Normal}\ \mathsf{s} \rangle \Rightarrow \mathsf{t}
\end{array}
$$

- Idea
  - b is a shallow embedding of an if-test
  - Outcome depends on current state
  - “pass” the state to b: $s \in b$ corresponds to $b\ s$

The While Loop

- Very similar to conditional
- While-execution depends on outcome of test

$$
\begin{array}{l}
[\![\mathsf{s} \in \mathsf{b};\\
\quad \Gamma \vdash \langle \mathsf{c}, \mathsf{Normal\ s}\rangle \Rightarrow \mathsf{s'};\\
\quad \Gamma \vdash \langle \mathsf{While\ b\ c}, \mathsf{s'}\rangle \Rightarrow \mathsf{t}\\
]\!] \Longrightarrow \Gamma \vdash \langle \mathsf{While\ b\ c}, \mathsf{Normal\ s}\rangle \Rightarrow \mathsf{t}
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{s} \notin \mathsf{b}\\
]\!] \Longrightarrow \Gamma \vdash \langle \mathsf{While\ b\ c}, \mathsf{Normal\ s}\rangle \Rightarrow \mathsf{Normal\ s}
\end{array}
$$

- Note: iteration by re-execution of loop itself in first case

Abrupt Termination

- Central construct: `Throw`

  $\Gamma \vdash \langle \text{Throw}, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s$

  - Converts ongoing execution into "abrupt termination"
  - Used to encode `break`, `continue`, `return`

- In abrupt termination mode, all statements are skipped

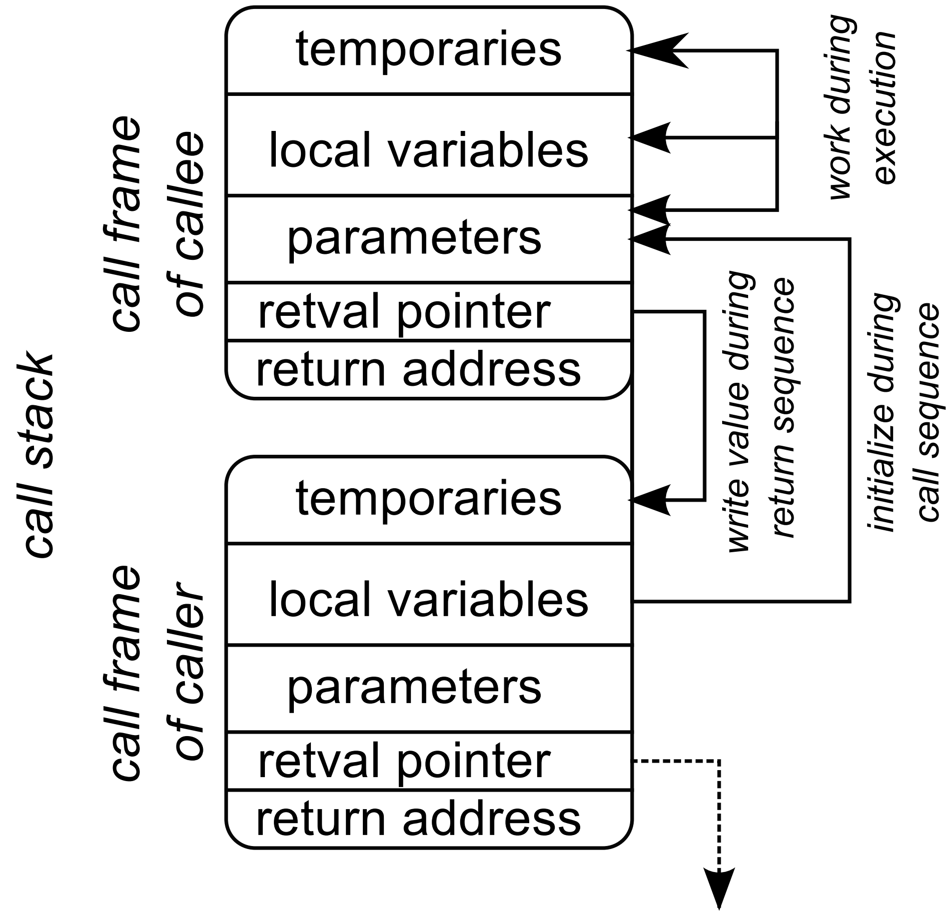
  $\Gamma \vdash \langle c, \text{Abrupt } s \rangle \Rightarrow \text{Abrupt } s$

- Ending abrupt termination

  $$\llbracket \Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s'; \\ \Gamma \vdash \langle c_2, \text{Normal } s' \rangle \Rightarrow t \\ \rrbracket \Longrightarrow \Gamma \vdash \langle \text{Catch } c_1\ c_2, \text{Normal } s \rangle \Rightarrow t$$

  - Execute c_1 and check whether it terminates abruptly
  - Begin c_2 in normal execution mode ⇨ resume "normal"
  - Symmetric rule for c_1 ending non-abruptly

Function Calls in Real Languages



- Local work done in registers [2, 1]
- Parameters often passed in registers
- Return value usually passed in register

Calling Functions

- Basic semantics: look up & execute the function's code

$$
\begin{array}{l}
[\![\Gamma\ p = \mathsf{Some}\ bdy;\\
\quad \Gamma \vdash \langle bdy, \mathsf{Normal}\ s\rangle \Rightarrow t\\
]\!] \Longrightarrow \Gamma \vdash \langle \mathsf{Call}\ p, \mathsf{Normal}\ s\rangle \Rightarrow t
\end{array}
$$

- Define & derive extended version for "real" languages

$$
\begin{array}{l}
[\![\Gamma\ p = \mathsf{Some}\ bdy;\\
\quad \Gamma \vdash \langle bdy, \mathsf{Normal}\ (init\ s)\rangle \Rightarrow \mathsf{Normal}\ t;\\
\quad \Gamma \vdash \langle c\ s\ t, \mathsf{Normal}\ (return\ s\ t)\rangle \Rightarrow u\\
]\!] \Longrightarrow \Gamma \vdash \langle call\ init\ p\ return\ c, \mathsf{Normal}\ s\rangle \Rightarrow u
\end{array}
$$

  - Pass arguments into local parameters within `init`
  - Execute the method's body
  - Store return value in caller's local state in `return`

A Variant with Call Depth

- Semantics keeps call-depth implicit

- For derivation of Hoare rules
  - Must argue about recursive procedure calls

  $\Rightarrow$ "p is correct assuming that p is correct"
  - Idea: Argument by induction on call-depth [3, 6]

- Define variant of semantics with explicit call depth

  inductive "execn" :: "[body,com,xstate,nat,xstate] ⇒ bool"

$\Rightarrow$ Can name & argue about (maximal) call depth n

Tracing the Call Depth

- Local commands work regardless of call depth

$$\Gamma \vdash \langle \mathsf{Basic}\, f, \mathsf{Normal}\, s\rangle =n\Rightarrow \mathsf{Normal}\,(f\, s)$$

$$\llbracket s \in b;\ \Gamma \vdash \langle c_1, \mathsf{Normal}\, s\rangle =n\Rightarrow t\ \rrbracket \Longrightarrow \Gamma \vdash \langle \mathsf{Cond}\, b\, c_1\, c_2, \mathsf{Normal}\, s\rangle =n\Rightarrow t$$

- Only function calls require a non-zero depth

$$\llbracket \Gamma\, p = \mathsf{Some}\, bdy;\ \Gamma \vdash \langle bdy, \mathsf{Normal}\, s\rangle =n\Rightarrow t\ \rrbracket \Longrightarrow \Gamma \vdash \langle \mathsf{Call}\, p, \mathsf{Normal}\, s\rangle =\mathsf{Suc}\, n\Rightarrow t$$

  - Depth is reduced in the execution of the body
  - Calls at depth 0 simply do not yield a result state

$\Rightarrow$ Simulate non-termination

A Concrete Syntax for Simpl

- Variables must be declared by `hoarestate`
- References prefixed by tick ´ (*not* the apostrophe)
- Assignment to variable `´x :==` e
- Sequencing a`;;`b
- Conditional

  `IF` e `THEN` *then-branch*

  `ELSE` *else-branch*

  `FI`

- Loops

  `WHILE` e

  `INV {|` *inv* `|}`

  `DO` *body* `OD`

A Glimpse At Verification

Interactive Demo

Today

- Inductively-defined relations in Isabelle
- Abstract syntax as datatypes / parse trees
- Introduction to Simpl
  - Abstract syntax
  - Semantics
- Concept: Shallow embedding (vs. deep) embedding
- The `xstate` trick for different outcomes

References

[1] Aho, Hopcroft, and Sethi. *Compilers – Principles, Tools, Techniques*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambrigde University Press, Cambridge, United Kingdom, 1998.

[3] Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 201–215, London, UK, 1996. Springer-Verlag.

[4] Norbert Schirmer. A Sequential Imperative Programming Language. http://afp.sourceforge.net/entries/Simpl.shtml.

[5] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2005.

[6] David von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999. http://isabelle.in.tum.de/Bali/papers/FSTTCS99.html.

[7] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*. Springer, 2004.