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]
Define Semantics by translation to given language
- Analogy: “C means what the compiler makes of it”
- VCG = Verification Condition Generator
- ⇒ Programm is correct if conditions can be proven
Inductively-Defined Predicates
Recap: Tail Recursion as Iteration

\[
\text{datatype instr = Init int | Add int | Mul int | Jump string}
\]

\[
\text{type-synonym state = int}
\]
\[
\text{type-synonym block = "instr list"}
\]

\[
\text{fun exec :: "state \Rightarrow instr list \Rightarrow state"}
\]
\[
\text{where } \text{exec s [] = s}
\]
\[
\text{exec s (Init i # xs) = exec i xs}
\]
\[
\text{exec s (Add i # xs) = exec (s + i) xs}
\]
\[
\text{exec s (Mul i # xs) = exec (s * i) xs}
\]

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

```haskell
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 (\(p, s, xs\). length xs)"")
  apply auto[4]

Problem: cannot bound the length of remaining computation
  \(\forall p \mid x s a. p l = \text{Some } a \implies \text{length } a < \text{Suc (length } xs)\)
Solution: Execution as Relation

\[
\text{inductive execpp :: } \text{prog } \Rightarrow \text{ state } \Rightarrow \text{ instr list } \Rightarrow \text{ state } \Rightarrow \text{ bool}
\]

where

\[
\begin{align*}
\text{s'} &= s \quad \Rightarrow \text{ execpp } p \ s \ [\ ] \ s' \\
\text{execpp } p \ i \ x s \ s' &= \text{ execpp } p \ s \ (\text{Init} \ i \ \# \ x s) \ s' \\
\text{execpp } p \ (s + i) \ x s \ s' &= \text{ execpp } p \ s \ (\text{Add} \ i \ \# \ x s) \ s' \\
\text{execpp } p \ (s \times i) \ x s \ s' &= \text{ execpp } p \ s \ (\text{Mul} \ i \ \# \ x s) \ s' \\
\text{[ case } p \text{ of} & \\
\hspace{1em} \text{Some } b & \Rightarrow \text{ execpp } p \ s \ b \ s' \\
\hspace{2em} \text{None} & \Rightarrow s' = s \\
\text{]} &= \text{ execpp } p \ s \ (\text{Jump} \ l \ \# \ x s) \ s'
\end{align*}
\]

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

\[
\text{lemma } \text{execpp } ["b" \mapsto [M u l 3]] 14 [J u m p "b"] 42 \\
\text{apply } \text{(rule execpp.intros)} \\
\text{apply } \text{(simp). . .}
\]
In case you are wondering . . .

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

\[
\text{inductive\_cases [elim!]: } \"\text{execpp p s }\text{[]} s\"' \]

\[
\text{inductive\_cases [elim!]: } \"\text{execpp p s (Init i }\#\text{ xs)} s\"' \]

\[
\text{inductive\_cases [elim!]: } \"\text{execpp p s (Add i }\#\text{ xs)} s\"' \]

\[
\text{inductive\_cases [elim!]: } \"\text{execpp p s (Mul i }\#\text{ xs)} s\"' \]

\[
\text{inductive\_cases [elim!]: } \"\text{execpp p s (Jump l }\#\text{ xs)} s\"' \]

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

\[
\text{lemma execpp-deterministic:} \\
\quad " [[ \text{execpp p s xs s'} 1; \text{execpp p s xs s'} 2 ] \implies s' 1 = s' 2"
\]

\text{apply (induct arbitrary: s' 2 rule: execpp.induct)}
\text{apply (force intro: execpp.intros split: split-if-asm)}
\text{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
  - \( P.\text{intros} \): the given introduction rules
  - \( P.\text{induct} \): induction over rule applications
  - \( P.\text{cases} \): case distinction by applied rules
- Command \text{inductive\_cases}
  - Specialized .\text{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
    ⇒ Can be used to express several languages

- Today: abstract syntax ($\approx$ parse trees)
- Later: concrete syntax
Statements (Commands)

```plaintext
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, ...)

```plaintext
    | Call "p"
    | Throw
    | Catch com com
```

Side-conditions / run-time errors

```plaintext
    | 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 | lookup s i > 0 \}

• Example: assignment i=i+1; becomes

  (λs. update i (lookup s 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
  \[
  \begin{align*}
  \left[ & \Gamma \vdash \langle c_1, \text{Normal} s \rangle \Rightarrow s' ; \\
  & \Gamma \vdash \langle c_2, s' \rangle \Rightarrow t \\
  \right] \Rightarrow \Gamma \vdash \langle \text{Seq} c_1 c_2, \text{Normal} s \rangle \Rightarrow t
  \end{align*}
  \]
The General Basic Command

- Definition is simple
  \[ \Gamma \vdash \langle \text{Basic } f, \text{Normal } s \rangle \Rightarrow \text{Normal } (fs) \]
  - Pass current state to function \( f \)
  - Treat result of function as new state
  \[ \Rightarrow f \text{ is a general “state update function”} \]
- 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

\[
\begin{align*}
[s \in b; \\
\Gamma \vdash \langle c_1, \text{Normal} s \rangle \Rightarrow t \\
] \implies \Gamma \vdash \langle \text{Cond} b c_1 c_2, \text{Normal} s \rangle \Rightarrow t
\end{align*}
\]

\[
\begin{align*}
[s \notin b; \\
\Gamma \vdash \langle c_2, \text{Normal} s \rangle \Rightarrow t \\
] \implies \Gamma \vdash \langle \text{Cond} b c_1 c_2, \text{Normal} s \rangle \Rightarrow t
\end{align*}
\]

- 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

\[
\begin{align*}
[s \in b; \\
\Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow s'; \\
\Gamma \vdash \langle \text{While } b \ c, s' \rangle \Rightarrow t \\
\] \implies \Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow t
\end{align*}
\]

\[
\begin{align*}
[s \notin b \\
\] \implies \Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow \text{Normal } s
\end{align*}
\]

- 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
  \[
  \left[ \Gamma \vdash \langle c_1, \text{Normal} s \rangle \Rightarrow \text{Abrupt} s' ;
  \Gamma \vdash \langle c_2, \text{Normal} s' \rangle \Rightarrow t
  \right] \Rightarrow \Gamma \vdash \langle \text{Catch } c_1 c_2, \text{Normal} s \rangle \Rightarrow t
  \]
- Execute \( c_1 \) and check whether it terminates abruptly
- Begin \( c_2 \) in normal execution mode \( \Rightarrow \) 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

\[
\Gamma \vdash \text{Some bdy};
\Gamma \vdash \langle \text{bdy}, \text{Normal s} \rangle \Rightarrow t
\]

\[
\Gamma \vdash \langle \text{Call p}, \text{Normal s} \rangle \Rightarrow t
\]

• Define & derive extended version for “real” languages

\[
\Gamma \vdash \text{Some bdy};
\Gamma \vdash \langle \text{bdy}, \text{Normal (init s)} \rangle \Rightarrow \text{Normal t};
\Gamma \vdash \langle \text{c s t}, \text{Normal (return s t)} \rangle \Rightarrow u
\]

\[
\Gamma \vdash \langle \text{call init p return c}, \text{Normal s} \rangle \Rightarrow u
\]

• 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
    ⇒ “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
  \[\text{inductive} \text{ execn} :: [\text{body, com, xstate}, \text{nat, xstate}] \Rightarrow \text{bool}\]

⇒ Can name & argue about (maximal) call depth \(n\)
Tracing the Call Depth

• Local commands work regardless of call depth
  \[ \Gamma \vdash \langle \text{Basic } f, \text{Normal } s \rangle = n \Rightarrow \text{Normal}(fs) \]

  \[ \begin{array}{c}
  \text{s} \in b; \\
  \Gamma \vdash \langle c_1, \text{Normal } s \rangle = n \Rightarrow t \\
  \end{array} \Rightarrow \Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle = n \Rightarrow t \]

• Only function calls require a non-zero depth
  \[ \begin{array}{c}
  \Gamma \vdash \text{p} = \text{Some bdy}; \\
  \Gamma \vdash \langle \text{bdy}, \text{Normal } s \rangle = n \Rightarrow t \\
  \end{array} \Rightarrow \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle = \text{Suc } n \Rightarrow t \]

• Depth is reduced in the execution of the body
• Calls at depth 0 simply do not yield a result state
  \[ \Rightarrow \text{Simulate non-termination} \]
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
  ```
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


