

Theorem-prover based Testing with HOL-TestGen

Burkhart Wolff¹

¹Université Paris-Sud, LRI, Orsay, France
wolff@lri.fr

M2R: Test des Systemes Informatiques
Orsay, 3 Feb 2010

Outline

- 1 Motivation: Program-based Testing
- 2 Foundation: A Language Embedding
- 3 Example: Squareroot
- 4 Conclusion

Outline

- 1 Motivation: Program-based Testing
- 2 Foundation: A Language Embedding
- 3 Example: Squareroot
- 4 Conclusion

Motivation

- So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \rightarrow post\ x\ (prog\ x)$$

- We have seen, this does not exclude to model reactive sequence test in HOL-TestGen.
- However, this seems still exclude the HOL-TestGen approach from program-based testing approaches (such as JavaPathfinder-SE or Pexx).

How to Realize White-box-Tests in HOL-TestGen?

- Fact: HOL is a powerful *logical framework* used to embed all sorts of specification and programming languages.
- Thus, we can embed the language of our choice in HOL-TestGen. We can represent abstract (and concrete) syntax as data type ...
- and derive the necessary rules for symbolic execution based tests from its semantics ...

The Master-Plan for White-box-Tests in HOL-TestGen?

- We embed an imperative core-language — called IMP — into HOL-TestGen, by defining its syntax and semantics
- We add a specification mechanism for IMP: Hoare-Triples
- we derive rules for symbolic evaluation and loop-unfolding.

Outline

- 1 Motivation: Program-based Testing
- 2 Foundation: A Language Embedding**
- 3 Example: Squareroot
- 4 Conclusion

IMP Syntax

The (abstract) IMP syntax is defined in Com.thy.

Com = Main +

typed decl loc

types

val = nat (*arb.*)

state = loc \Rightarrow val

aexp = state \Rightarrow val

bexp = state \Rightarrow bool

datatype com =

SKIP

| "==" loc aexp (**infixl** 60)

| Semi com com ("_ ; _"[60, 60]10)

| Cond bexp com com

 (" IF _ THEN _ ELSE _"60)

| While bexp com ("WHILE _ DO_"60)

The type loc stands for *locations*. Note that expressions are represented as HOL-functions depending on state. The *datatype com* stands for commands (command sequences).

Example: The Integer Square-Root Program

```

tm  ::= λs. 1;
sum ::= λs. 1;
i   ::= λs. 0;
WHILE λs. (s sum) <= (s a) DO
  (i   ::= λs. (s i) + 1;
   tm  ::= λs. (s tm) + 2;
   sum ::= λs. (s tm) + (s sum))

```

How does this program work?

Note: There is the implicit assumption, that `tm`, `sum` and `i` are distinct locations, i.e. they are not aliases from each other !

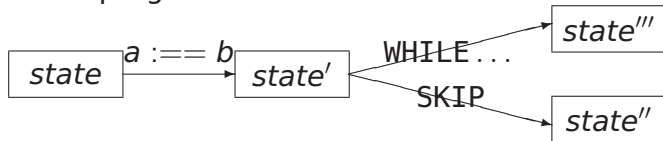
IMP Semantics I: (Natural Semantics)

Natural semantics going back to Plotkin

IMP Semantics I: (Natural Semantics)

Natural semantics going back to Plotkin

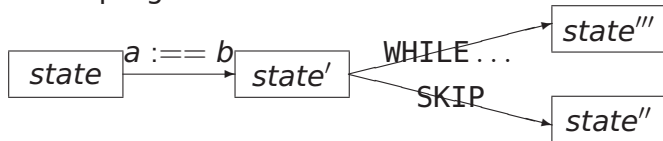
idea: programs relates states.



IMP Semantics I: (Natural Semantics)

Natural semantics going back to Plotkin

idea: programs relates states.



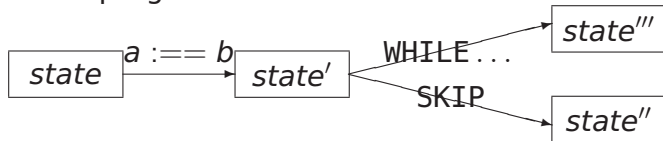
consts $\text{evalc} :: (\text{com} \times \text{state} \times \text{state}) \text{ set}$

translations " $\langle c, s \rangle \xrightarrow{c} s'$ " \equiv " $(c, s, s') \in \text{evalc}$ "

IMP Semantics I: (Natural Semantics)

Natural semantics going back to Plotkin

idea: programs relates states.



consts $\text{evalc} :: (\text{com} \times \text{state} \times \text{state}) \text{ set}$

translations " $\langle c, s \rangle \xrightarrow{c} s'$ " \equiv " $(c, s, s') \in \text{evalc}$ "

The transition relation of natural semantics is inductively defined.

The transition relation of natural semantics is inductively defined.

This means intuitively: The evaluation steps defined by the following rules are the *only* possible steps.

The transition relation of natural semantics is inductively defined.

This means intuitively: The evaluation steps defined by the following rules are the *only* possible steps.

Let's go . . .

The natural semantics as inductive definition:

inductive evalc

intrs

Skip: $\langle \text{SKIP}, s \rangle \xrightarrow{c} s$ Assign: $\langle x ::= a, s \rangle \xrightarrow{c} s[x \mapsto a]$

The natural semantics as inductive definition:

inductive evalc

intrs

Skip: $\langle \text{SKIP}, s \rangle \xrightarrow{c} s$

Assign: $\langle x ::= a, s \rangle \xrightarrow{c} s[x \mapsto a \ s]$

Note that $s[x \mapsto a \ s]$ is an abbreviation for *update* $s \ x \ (a \ s)$, where

$\text{update } s \ x \ v \equiv \lambda y. \text{ if } y=x \text{ then } v \text{ else } s \ y$

The natural semantics as inductive definition:

inductive evalc

intrs

Skip: $\langle \text{SKIP}, s \rangle \xrightarrow{c} s$

Assign: $\langle x ::= a, s \rangle \xrightarrow{c} s[x \mapsto a \ s]$

Note that $s[x \mapsto a \ s]$ is an abbreviation for *update* $s \ x \ (a \ s)$, where

$\text{update } s \ x \ v \equiv \lambda y. \text{ if } y=x \text{ then } v \text{ else } s \ y$

Note that a is of type aexp or bexp .

Excursion: A minimal memory model:

$$\begin{aligned} & (s[x \mapsto E]) x = E \\ x \neq y & \implies (s[x \mapsto E]) y = s y \end{aligned}$$

This small memory theory contains the *typical* rules for updating and memory-access. Note that this rewrite system is in fact executable!

The semantics for the sequential composition of statements can be described as follows:

$$\text{Semi: } \llbracket \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c'} s'' \rrbracket \implies \langle c; c', s \rangle \xrightarrow{c} s''$$

The semantics for the sequential composition of statements can be described as follows:

$$\text{Semi: } \llbracket \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c'} s'' \rrbracket \implies \langle c; c', s \rangle \xrightarrow{c} s''$$

Rationale of natural semantics:

- if you can “jump” via c from s to s' , ...

The semantics for the sequential composition of statements can be described as follows:

$$\text{Semi: } \llbracket \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c'} s'' \rrbracket \implies \langle c; c', s \rangle \xrightarrow{c} s''$$

Rationale of natural semantics:

- if you can “jump” via c from s to s' , ...
- and if you can “jump” via c' from s' to s'' ...

The semantics for the sequential composition of statements can be described as follows:

$$\text{Semi: } \llbracket \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c'} s'' \rrbracket \implies \langle c; c', s \rangle \xrightarrow{c} s''$$

Rationale of natural semantics:

- if you can “jump” via c from s to s' , ...
- and if you can “jump” via c' from s' to s'' ...
- then this means that you can “jump” via the composition $c; c'$ from c to c'' .

The other constructs of the language are treated analogously:

$$\begin{aligned} \text{IfTrue:} \quad & \llbracket b \ s; \langle c, s \rangle \xrightarrow{c} s' \rrbracket \\ & \implies \langle \text{IF } b \ \text{THEN } c \ \text{ELSE } c', s \rangle \xrightarrow{c} s' \end{aligned}$$

$$\begin{aligned} \text{IfFalse:} \quad & \llbracket \neg b \ s; \langle c', s \rangle \xrightarrow{c} s' \rrbracket \\ & \implies \langle \text{IF } b \ \text{THEN } c \ \text{ELSE } c', s \rangle \xrightarrow{c} s' \end{aligned}$$

$$\begin{aligned} \text{WhileFalse:} \quad & \llbracket \neg b \ s \rrbracket \\ & \implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \xrightarrow{c} s \end{aligned}$$

$$\begin{aligned} \text{WhileTrue:} \quad & \llbracket b \ s; \langle c, s \rangle \xrightarrow{c} s'; \langle \text{WHILE } b \ \text{DO } c, s' \rangle \xrightarrow{c} s'' \rrbracket \\ & \implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \xrightarrow{c} s'' \end{aligned}$$

Note that for non-terminating programs no final state can be derived !

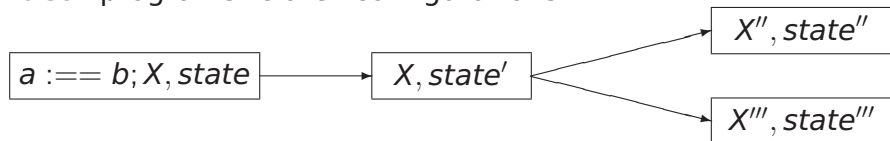
IMP Semantics II: (Transition Semantics)

The **transition semantics** is inspired by abstract machines.

IMP Semantics II: (Transition Semantics)

The **transition semantics** is inspired by abstract machines.

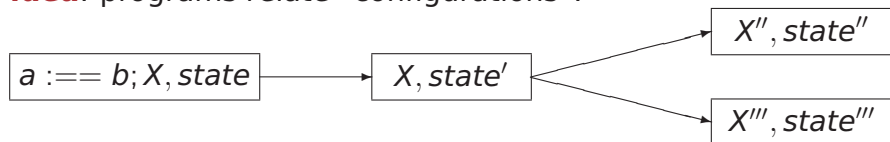
idea: programs relate “configurations”.



IMP Semantics II: (Transition Semantics)

The **transition semantics** is inspired by abstract machines.

idea: programs relate “configurations”.



consts $\text{evalc1} :: ((\text{com} \times \text{state}) \times (\text{com} \times \text{state})) \text{ set}$

translations $"cs \rightarrow cs'" \equiv "(cs, cs') \in \text{evalc1}"$

inductive evalc1

intro

Assign: $(x ::= a, s) \text{ --1--> } (\text{SKIP}, s[x \mapsto a \ s])$ Semi1: $(\text{SKIP}; c, s) \text{ --1--> } (c, s)$ Semi2: $(c, s) \text{ --1--> } (c'', s')$
 $\implies (c; c', s) \text{ --1--> } (c''; c', s')$

inductive evalc1

intro

Assign: $(x ::= a, s) \rightarrow (SKIP, s[x \mapsto a \ s])$ Semi1: $(SKIP; c, s) \rightarrow (c, s)$ Semi2: $(c, s) \rightarrow (c'', s')$
 $\implies (c; c', s) \rightarrow (c''; c', s')$

Rationale of Transition Semantics:

- the first component in a configuration represents a *stack of statements yet to be executed* . . .

inductive evalc1

intro

Assign: $(x ::= a, s) \rightarrow (SKIP, s[x \mapsto a \ s])$ Semi1: $(SKIP; c, s) \rightarrow (c, s)$ Semi2: $(c, s) \rightarrow (c'', s')$
 $\implies (c; c', s) \rightarrow (c''; c', s')$

Rationale of Transition Semantics:

- the first component in a configuration represents a *stack of statements yet to be executed* . . .
- this stack can also be seen as a *program counter* . . .
- transition semantics is close to an abstract machine.

IfTrue:

$$b \ s \Longrightarrow (\text{IF } b \ \text{THEN } c' \ \text{ELSE } c'', s) \ -1-\> (c', s)$$

IfFalse:

$$\neg b \ s \Longrightarrow (\text{IF } b \ \text{THEN } c' \ \text{ELSE } c'', s) \ -1-\> (c'', s)$$

WhileFalse:

$$\neg b \ s \Longrightarrow (\text{WHILE } b \ \text{DO } c, s) \ -1-\> (\text{SKIP}, s)$$

WhileTrue:

$$b \ s \Longrightarrow (\text{WHILE } b \ \text{DO } c, s) \ -1-\> (c; \text{WHILE } b \ \text{DO } c, s)$$

IfTrue:

$$b \ s \Longrightarrow (\text{IF } b \ \text{THEN } c' \ \text{ELSE } c'', s) \ -1-\> (c', s)$$

IfFalse:

$$\neg b \ s \Longrightarrow (\text{IF } b \ \text{THEN } c' \ \text{ELSE } c'', s) \ -1-\> (c'', s)$$

WhileFalse:

$$\neg b \ s \Longrightarrow (\text{WHILE } b \ \text{DO } c, s) \ -1-\> (\text{SKIP}, s)$$

WhileTrue:

$$b \ s \Longrightarrow (\text{WHILE } b \ \text{DO } c, s) \ -1-\> (c; \text{WHILE } b \ \text{DO } c, s)$$

A non-terminating loop always leads to successor configurations ...

IMP Semantics III: (Denotational Semantics)

Idea:

IMP Semantics III: (Denotational Semantics)

Idea:

Associate “the meaning of the program” to a statement directly by a semantic domain. Explain loops as fixpoint (or *limit*) construction on this semantic domain.

IMP Semantics III: (Denotational Semantics)

Idea:

Associate “the meaning of the program” to a statement directly by a semantic domain. Explain loops as fixpoint (or *limit*) construction on this semantic domain.

As semantic domain we choose the state relation:

types `com_den = (state × state) set`

IMP Semantics III: (Denotational Semantics)

Idea:

Associate “the meaning of the program” to a statement directly by a semantic domain. Explain loops as fixpoint (or *limit*) construction on this semantic domain.

As semantic domain we choose the state relation:

types $\text{com_den} = (\text{state} \times \text{state}) \text{ set}$
and declare the semantic function:

consts $C :: \text{com} \Rightarrow \text{com_den}$

The semantic function C is defined recursively over the syntax.

primrec

$C(\text{SKIP}) = \text{Id}$ (* \equiv identity relation *)

$C(x ::= a) = \{(s,t). t = s[x \mapsto a]\}$

$C(c ; c') = C(c') \circ C(c)$ (* \equiv seq. composition *)

$C(\text{IF } b \text{ THEN } c' \text{ ELSE } c'') =$
 $\{(s,t). (s,t) \in C(c') \wedge b(s)\} \cup$
 $\{(s,t). (s,t) \in C(c'') \wedge \neg b(s)\}$ "

$C(\text{WHILE } b \text{ DO } c) = \text{lfp } (\Gamma b (C(c)))$ "

primrec

$$C(\text{SKIP}) = \text{Id} \quad (* \equiv \textit{identity relation} *)$$

$$C(x ::= a) = \{(s,t). t = s[x \mapsto a]\}$$

$$C(c ; c') = C(c') \circ C(c) \quad (* \equiv \textit{seq. composition} *)$$

$$C(\text{IF } b \text{ THEN } c' \text{ ELSE } c'') = \\ \{(s,t). (s,t) \in C(c') \wedge b(s)\} \cup \\ \{(s,t). (s,t) \in C(c'') \wedge \neg b(s)\}$$

$$C(\text{WHILE } b \text{ DO } c) = \text{lfp } (\Gamma b (C(c)))$$

where:

$$\Gamma b c \equiv (\lambda \varphi. \{(s,t). (s,t) \in (\varphi \circ c) \wedge b(s)\} \cup \\ \{(s,t). s=t \wedge \neg b(s)\})$$

and where the least-fixpoint-operator $\text{lfp } F$ corresponds in this special case to:

$$\bigcup_{n \in \mathbb{N}} F^n$$

IMP Semantics: Theorems I

Theorem: Natural and Transition Semantics Equivalent

$$(c, s) \text{ --*--> } (\text{SKIP}, t) = (\langle c, s \rangle \xrightarrow{c} t)$$

where $cs \text{ --*--> } cs' \equiv (cs, cs') \in \text{evalc1}^*$, i.e. the new arrow denotes the transitive closure over old one.

IMP Semantics: Theorems I

Theorem: Natural and Transition Semantics Equivalent

$$(c, s) \text{ --*--> } (\text{SKIP}, t) = (\langle c, s \rangle \xrightarrow{c} t)$$

where $cs \text{ --*--> } cs' \equiv (cs, cs') \in \text{evalc1}^*$, i.e. the new arrow denotes the transitive closure over old one.

Theorem: Denotational and Natural Semantics Equivalent

$$((s, t) \in C c) = (\langle c, s \rangle \xrightarrow{c} t)$$

IMP Semantics: Theorems I

Theorem: Natural and Transition Semantics Equivalent

$$(c, s) \text{ --*--> } (\text{SKIP}, t) = (\langle c, s \rangle \xrightarrow{c} t)$$

where $cs \text{ --*--> } cs' \equiv (cs, cs') \in \text{evalc1}^*$, i.e. the new arrow denotes the transitive closure over old one.

Theorem: Denotational and Natural Semantics Equivalent

$$((s, t) \in C c) = (\langle c, s \rangle \xrightarrow{c} t)$$

i.e. all three semantics are closely related !

IMP Semantics: Theorems II

Theorem: Natural Semantics can be evaluated equationally !!!

$$\langle \text{SKIP}, s \rangle \xrightarrow{c} s' = (s' = s)$$

$$\langle x := a, s \rangle \xrightarrow{c} s' = (s' = s[x \mapsto a])$$

$$\langle c; c', s \rangle \xrightarrow{c} s' = (\exists s''. \langle c, s \rangle \xrightarrow{c} s'' \wedge \langle c', s'' \rangle \xrightarrow{c} s')$$

$$\langle \text{IF } b \text{ THEN } c \text{ ELSE } c', s \rangle \xrightarrow{c} s' = (b \wedge \langle c, s \rangle \xrightarrow{c} s') \vee$$

$$(\neg b \wedge \langle c', s \rangle \xrightarrow{c} s')$$

Note: This is the key for evaluating a program symbolically !!!

Example: “a:=2;b:=2*a”

$$\begin{aligned}
& \langle a:=\lambda s. 2; b:=\lambda s. 2 * (s a), s \rangle \xrightarrow{c} s' \\
\equiv & (\exists s''. \langle a:=\lambda s. 2, s \rangle \xrightarrow{c} s'' \wedge \langle b:=\lambda s. 2 * (s a), s'' \rangle \xrightarrow{c} s') \\
\equiv & (\exists s''. s'' = s[a \mapsto (\lambda s. 2) s] \wedge s' = s''[b \mapsto (\lambda s. 2 * (s a)) s'']) \\
\equiv & (\exists s''. s'' = s[a \mapsto 2] \wedge s' = s''[b \mapsto 2 * (s'' a)]) \\
\equiv & s' = s[a \mapsto 2][b \mapsto 2 * (s[a \mapsto 2] a)] \\
\equiv & s' = s[a \mapsto 2][b \mapsto 2 * 2] \\
\equiv & s' = s[a \mapsto 2][b \mapsto 4]
\end{aligned}$$

Note:

- 1 The λ -notation is perhaps a bit irritating, but helps to get the nitty-gritty details of substitution right.
- 2 The forth step is correct due to the “one-point-rule” $(\exists x. x = e \wedge P(x)) = P(e)$.
- 3 This does not work for the loop and for recursion...

IMP Semantics: Theorems III

Denotational semantics makes it easy to prove facts like:

$$C(\text{WHILE } b \text{ DO } c) = C(\text{IF } b \text{ THEN } c; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP})$$

$$C(\text{SKIP}; c) = C(c)$$

$$C(c; \text{SKIP}) = C(c)$$

$$C((c; d); e) = C(c; (d; e))$$

$$C((\text{IF } b \text{ THEN } c \text{ ELSE } d); e) = C(\text{IF } b \text{ THEN } c; e \text{ ELSE } d; e)$$

etc.

Program Annotations: Assertions revisited.

For our scenario, we need a mechanism to combine programs with their specifications.

The Standard: Hoare-Tripel with Pre- and Post-Conditions a special form of assertions.

types $\text{assn} = \text{state} \Rightarrow \text{bool}$

consts $\text{valid} :: (\text{assn} \times \text{com} \times \text{assn}) \Rightarrow \text{bool}$ ("|= { } _ { }")

defs

$|= \{P\}c\{Q\} \equiv \forall s. \forall t. (s,t) \in C(c) \longrightarrow P s \longrightarrow Q t$

Note that this reflects partial correctness; for a non-terminating program c , i.e. $(s,t) \notin C(c)$, a Hoare-Triple does not enforce anything as post-condition !

Finally: Symbolic Evaluation.

For programs without loop, we have already anything together for symbolic evaluation:

$$\forall s s'. \langle c, s \rangle \xrightarrow{c} s' \wedge P s \rightarrow Q s' \\ \implies \models \{P\} c \{Q\}$$

or in more formal, natural-deduction notation:

$$\frac{\begin{array}{c} [\langle c, s \rangle \rightarrow_c s', P s]_{s, s'} \\ \vdots \\ Q s' \end{array}}{\models \{P\} c \{Q\}}$$

Applied in backwards-inference, this rule *generates* the constraints for the states that were amenable to equational evaluation rules shown before.

Outline

- 1 Motivation: Program-based Testing
- 2 Foundation: A Language Embedding
- 3 Example: Squareroot**
- 4 Conclusion

Example: “ $\models \{0 \leq x\} a ::= x; b ::= 2 * a \{0 \leq b\}$ ”

$$|= \{ \lambda s. 0 \leq s \ x \} a ::= \lambda s. s \ x; b ::= \lambda s. 2 * (s \ a) \{ \lambda s. 0 \leq s \ b \}$$

$$\Leftarrow s' = s[a \mapsto s \ x][b \mapsto 2 * (s[a \mapsto s \ x] \ a)] \wedge 0 \leq s \ x \longrightarrow 0 \leq s' \ b$$

$$\equiv s' = s[a \mapsto s \ x][b \mapsto 2 * (s \ x)] \wedge \text{“PRE } s\text{”} \longrightarrow \text{“POST } s'\text{”}$$

$$\equiv \text{“PRE } s\text{”} \longrightarrow \text{“POST } (s[a \mapsto s \ x][b \mapsto 2 * (s \ x)])\text{”}$$

Note:

- **Note:** the logical constant

$s' = s[a \mapsto s \ x][b \mapsto 2 * s \ x] \wedge 0 \leq s \ x$ consists of the constraint that functionally relate pre-state s to post-state s' and the **Path-Condition** (in this case just “PRE s ”).

- This also works for conditionals ... Revise !
- The implication is actually the core validation problem: It means that for a certain path, we search for the solution of a path condition that validates the post-condition. We can decide to 1) keep it as test hypothesis, 2) test k witnesses and add a uniformity hypothesis, or 3) verify it.

Validation of Post-Conditions for a Given Path:

Ad 1 : Add $THYP(PRE\ s \rightarrow POST(s[a \mapsto s\ x][b \mapsto 2 * (s\ x)]))$
(is: $THYP(0 \leq s\ x \rightarrow 0 \leq 2 * s\ x)$) as test hypothesis.

Ad 2 : Find witness to $\exists s. 0 \leq s\ x$, run a test on this witness
(does it establish the post-condition?) and add the
uniformity-hypothesis:

$$THYP(\exists s. 0 \leq s\ x \rightarrow 0 \leq 2 * s\ x \rightarrow \forall s. 0 \leq s\ x \rightarrow 0 \leq 2 * s\ x).$$

Ad 3 : Verify the implication, which is in this case easy.

Option 1 can be used to model weaker coverage criteria than all statements and k loops, option 2 can be significantly easier to show than option 3, but as the latter shows, for simple formulas, testing is not *necessarily* the best solution.

Control-heuristics necessary.

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

What to do with loops ???

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

What to do with loops ???

Answer: Unfolding to a certain depth.

Handling Loops (and Recursion).

We have found a symbolic execution method that works for programs with assignments, SKIP's, sequentials, and conditionals.

What to do with loops ???

Answer: Unfolding to a certain depth.

In the sequel, we define an unfolding function, prove it semantically correct with respect to C, and apply the procedure above again.

Handling Loops (and Recursion).

consts unwind :: "nat \times com \Rightarrow com"

recdef unwind "less_than <*lex*> measure(λ s. size s)"

"unwind(n, SKIP) = SKIP"

"unwind(n, a ::= E) = (a ::= E)"

"unwind(n, IF b THEN c ELSE d) = IF b THEN unwind(n,c) ELSE unwind(n,d)"

"unwind(n, WHILE b DO c) =

if 0 < n

then IF b THEN unwind(n,c)@@unwind(n- 1,WHILE b DO c) ELSE SKIP

else WHILE b DO unwind(0, c))"

"unwind(n, SKIP; c) = unwind(n, c)"

"unwind(n, c ; SKIP) = unwind(n, c)"

"unwind(n, (IF b THEN c ELSE d) ; e) =

(IF b THEN (unwind(n,c;e)) ELSE(unwind(n,d;e)))"

"unwind(n, (c ; d); e) = (unwind(n, c;d))@@(unwind(n,e))"

"unwind(n, c ; d) = (unwind(n, c))@@(unwind(n, d))"

Handling Loops (and Recursion).

where the primitive recursive auxiliary function $c@@d$ appends a command d to the last command in c that is reachable from the root via sequential composition modes.

consts "@@" :: "[com,com] \Rightarrow com" (infixr 70)

primrec

"SKIP @@ c = c"

"(x ::= E) @@ c = ((x ::= E); c)"

"(c;d) @@ e = (c; d @@ e)"

"(IF b THEN c ELSE d) @@ e = (IF b THEN c @@ e ELSE d @@ e)"

"(WHILE b DO c) @@ e = ((WHILE b DO c); e)"

Handling Loops (and Recursion).

Proofs for Correctness are straight-forward (done in Isabelle/HOL) based on the shown rules for denotationally equivalent programs ...

Theorem: Unwind and Concat correct

$C(c @@ d) = C(c;d)$ and $C(\text{unwind}(n,c)) = C(c)$

Handling Loops (and Recursion).

This allows us (together with the equivalence of natural and denotational semantics) to generalize our scheme:

Handling Loops (and Recursion).

This allows us (together with the equivalence of natural and denotational semantics) to generalize our scheme:

$$\forall s s'. \langle \text{unwind}(n,c), s \rangle \xrightarrow{c} s' \wedge P s \rightarrow Q s' \\ \implies \models \{P\} c \{Q\}$$

for an arbitrary (user-defined!) n !

Or in natural deduction notation:

$$\frac{\begin{array}{c} [\langle \text{unwind}(n, c), s \rangle \xrightarrow{c} s', P s]_{s, s'} \\ \vdots \\ Q s' \end{array}}{\models \{P\} c \{Q\}}$$

Handling Loops (and Recursion).

Example:

“ $\models \{True\} \text{integer_squareroot} \{i^2 \leq a \wedge a \leq (i + 1)^2\}$ ”

Setting the depth to $n = 3$ and running the process yields:

Handling Loops (and Recursion).

Example:

“ $\models \{True\} \textit{integer_squareroot} \{i^2 \leq a \wedge a \leq (i + 1)^2\}$ ”

Setting the depth to $n = 3$ and running the process yields:

1. $\llbracket 9 \leq s \ a; \langle \textit{WHILE} \ \lambda s. \ s \ \textit{sum} \leq s \ a$
 $\quad \textit{DO} \ i ::= \lambda s. \ \textit{Suc} \ (s \ i);$
 $\quad \quad (\textit{tm} ::= \lambda s. \ \textit{Suc} \ (\textit{Suc} \ (s \ \textit{tm})));$
 $\quad \quad \textit{sum} ::= \lambda s. \ s \ \textit{tm} + s \ \textit{sum} \rangle,$
 $\quad s(i ::= 3, \ \textit{tm} ::= 7, \ \textit{sum} ::= 16) \rangle \xrightarrow{c} s'$
 $\rrbracket \implies \textit{post} \ s'$
2. $\llbracket 4 \leq s \ a; \ 8 < s \ a; \ s' = s \ (i ::= 2, \ \textit{tm} ::= 5, \ \textit{sum} ::= 9) \rrbracket \implies \textit{post} \ s'$
3. $\llbracket 1 \leq s \ a; \ s \ a < 4; \ s' = s \ (i ::= 1, \ \textit{tm} ::= 3, \ \textit{sum} ::= 4) \rrbracket \implies \textit{post} \ s'$
4. $\llbracket s \ a = 0; \ s' = s(\textit{tm} ::= 1, \ \textit{sum} ::= 1, \ i ::= 0) \rrbracket \implies \textit{post} \ s'$

which is a neat enumeration of all path-conditions for paths up to $n = 3$ times through the loop, except subgoal 1, which is:

Explicit test-Hypothesis in White-Box-Tests:

1. $\text{THYP}(9 \leq s \wedge a \wedge \langle \text{WHILE } \lambda s. s \text{ sum} \leq s \wedge a$
 $\text{DO } i ::= \lambda s. \text{Suc } (s \ i) ;$
 $(\text{tm} ::= \lambda s. \text{Suc } (\text{Suc } (s \ \text{tm})) ;$
 $\text{sum} ::= \lambda s. s \ \text{tm} + s \ \text{sum}),$
 $s(i := 3, \text{tm} := 7, \text{sum} := 16) \rangle \xrightarrow{c} s'$
 $\rightarrow \text{post } s')$

... a kind of “structural” regularity hypothesis !

Summary: Program-based Tests in HOL-TestGen:

- 1 It is possible to do white-box tests in HOL-TestGen
- 2 Requisite: Denotational and Natural Semantics for a programming language
- 3 Proven correct unfolding scheme
- 4 Explicit Test-Hypotheses Concept also applicable for Program-based Testing
- 5 Can either verify or test paths ...

Summary (II) : Program-based Tests in HOL-TestGen:

Open Questions:

- 1 Does it scale for *large programs* ???
- 2 Does it scale for *complex memory models* ???
- 3 What heuristics should we choose ???
- 4 How to combine the approach with randomized tests?
- 5 How to design Modular Test Methods ???

Outline

- 1 Motivation: Program-based Testing
- 2 Foundation: A Language Embedding
- 3 Example: Squareroot
- 4 Conclusion**

Conclusion I

- Approach based on theorem proving
 - test specifications are written in HOL
 - functional programming, higher-order, pattern matching
- Test hypothesis explicit and controllable by the user (could even be verified!)
- Proof-state explosion controllable by the user
- Although logically puristic, systematic unit-test of a “real” compiler library is feasible!
- Verified tool inside a (well-known) theorem prover

Conclusion II

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same!

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)
- HOL-Testgen is a **verified test-tool** (entirely based on derived rules ...)
- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Conclusion II

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same!
TS pattern **Unit Test**:

$$\text{pre } x \longrightarrow \text{post } x(\text{prog } x)$$

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)
- HOL-Testgen is a **verified test-tool** (entirely based on derived rules ...)
- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Conclusion II

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same!
TS pattern **Sequence Test**:

$$\text{accept } trace \implies P(\text{Mfold } trace \ \sigma_0 \text{prog})$$

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)
- HOL-Testgen is a **verified test-tool** (entirely based on derived rules ...)
- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Conclusion II

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same!
TS pattern **Reactive Sequence Test**:

$$\text{accept } trace \implies P(\text{Mfold } trace \sigma_0$$

(observer observer rebind subst prog))

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)
- HOL-Testgen is a **verified test-tool** (entirely based on derived rules ...)
- The **White-box Test** offers potentials to prune unfeasible paths early ... (but no large programs tried so far ...)

Bibliography I