Theorem-prover based Testing with HOL-TestGen

Burkhart Wolff\textsuperscript{1}

\textsuperscript{1}Université Paris-Sud, LRI, Orsay, France
wolff@lri.fr

M2R: Test des Systemes Informatiques
Orsay, 15th Dec 2009
Outline

1 Advanced Test Scenarios
Outline

1. Advanced Test Scenarios
Tuning the Workflow by Interactive Proof

Observations:

- Test-theorem generations is fairly easy ...
- Test-data generation is fairly hard ...
  (it does not really matter if you use random solving or just plain enumeration !!!)
- Both are scalable processes ... 
  (via parameters like depth, iterations, ...)
- There are bad and less bad forms of test-theorems !!!
- **Recall**: Test-theorem and test-data generation are normal form computations:
  \[ \Rightarrow \] More Rules, better results ...
What makes a Test-case “Bad”

- redundancy.
- many unsatisfiable constraints.
- many constraints with unclear logical status.
- constraints that are difficult to solve. (like arithmetics).
Case Studies: Red-black Trees

Motivation

Test a non-trivial and widely-used data structure.

- part of the SML standard library
- widely used internally in the sml/NJ compiler, e.g., for providing efficient implementation for Sets, Bags, . . . ;
- very hard to generate (balanced) instances randomly
Modeling Red-black Trees I

Red-Black Trees:

**Red Invariant:** each red node has a black parent.

**Black Invariant:** each path from the root to an empty node (leaf) has the same number of black nodes.

datatype
color = R | B
tree = E | T color (α tree) (β::ord item) (α tree)
Modeling Red-black Trees II

- Red-Black Trees: Test Theory

**consts**

- redinv :: tree ⇒ bool
- blackinv :: tree ⇒ bool

**recdef** blackinv measure (λ t. (size t))

- blackinv E = True
- blackinv (T color a y b) =
  - ((blackinv a) ∧ (blackinv b))
  - ∧ ((max B (height a)) = (max B (height b))))

recdev redinv measure ...
Red-black Trees: Test Specification

- Red-Black Trees: Test Specification

  **test_spec:**
  
  "isord t ∧ redinv t ∧ blackinv t \\n  ∧ isin (y::int) t \\n  → \\n  (blackinv(prog(y,t)))"

  where prog is the program under test (e.g., delete).

- Using the standard-workflows results, among others:

  RSF → blackinv (prog (100, T B E 7 E))
  blackinv (prog (−91, T B (T R E −91 E) 5 E))
Red-black Trees: A first Summary

Observation:

Guessing (i.e., random-solving) valid red-black trees is difficult.

- On the one hand:
  - random-solving is nearly impossible for solutions which are “difficult” to find
  - only a small fraction of trees with depth $k$ are balanced

- On the other hand:
  - we can quite easily construct valid red-black trees interactively.
Observation:

Guessing (i.e., random-solving) valid red-black trees is difficult.

- On the one hand:
  - random-solving is nearly impossible for solutions which are “difficult” to find
  - only a small fraction of trees with depth $k$ are balanced

- On the other hand:
  - we can quite easily construct valid red-black trees interactively.

**Question:**

Can we improve the test-data generation by using our knowledge about red-black trees?
Red-black Trees: Hierarchical Testing I

Idea:
Characterize valid instances of red-black tree in more detail and use this knowledge to guide the test data generation.

- First attempt:
  enumerate the height of some trees without black nodes

  **lemma** maxB_0_1:
  "max_B_height (E:: int tree) = 0"

  **lemma** maxB_0_5:
  "max_B_height (T R (T R E 2 E) (5::int) (T R E 7 E)) = 0"

- But this is tedious . . .
Red-black Trees: Hierarchical Testing I

**Idea:**
Characterize valid instances of red-black tree in more detail and use this knowledge to guide the test data generation.

- First attempt:
  enumerate the height of some trees without black nodes

  **lemma** maxB_0_1:
  "max_B_height (E:: int tree) = 0"

  **lemma** maxB_0_5:
  "max_B_height (T R (T R E 2 E) (5::int) (T R E 7 E)) = 0"

- But this is tedious . . . and error-prone
How to Improve Test-Theorems

- New simplification rule establishing unsatisfiability.
- New rules establishing equational constraints for variables.

\[(\text{max}_B\text{ height } (T \cdot x \cdot t1 \cdot \text{val} \cdot t2) = 0) \implies (x = R)\]

\[(\text{max}_B\text{ height } x = 0) = (x = E \lor \exists a \cdot y \cdot b. \ (x = T \cdot R \cdot a \cdot y \cdot b \land \max(\text{max}_B\text{ height} \cdot a) \land (\text{max}_B\text{ height} \cdot b) = 0)\]

- Many rules are domain specific — few hope that automation pays really off.
Improvement Slots

- logical massage of test-theorem.
- in-situ improvements:
  add new rules into the context before gen_test_cases.
- post-hoc logical massage of test-theorem.
- in-situ improvements:
  add new rules into the context before gen_test_data.
Red-black Trees: sml/NJ Implementation

(a) pre-state

Figure: Test Data for Deleting a Node in a Red-Black Tree
Red-black Trees: sml/NJ Implementation

(b) pre-state: delete “8”

Figure: Test Data for Deleting a Node in a Red-Black Tree
Red-black Trees: sml/NJ Implementation

(b) pre-state: delete “8”      (c) correct result

Figure: Test Data for Deleting a Node in a Red-Black Tree
Red-black Trees: sml/NJ Implementation

(b) pre-state: delete “8”  (c) correct result  (d) result of sml/NJ

Figure: Test Data for Deleting a Node in a Red-Black Tree
Red-black Trees: Summary

- Statistics: 348 test cases were generated (within 2 minutes)
- One error found: crucial violation against red/black-invariants
- Red-black-trees degenerate to linked list (insert/search, etc. only in linear time)
- Not found within 12 years
- Reproduced meanwhile by random test tool
Motivation: Sequence Test

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

  \[ pre \, x \rightarrow post \, x \, (prog \, x) \]

- This seems to limit the HOL-TestGen approach to UNIT-tests.
Apparent Limitations of HOL-TestGen

- No Non-determinism.
Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a relational description of `prog`.

- **No Automata** - No Tests for Sequential Behaviour.
Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a relational description of prog.

- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .
Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a relational description of \textit{prog}.

- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .

- No possibility to describe \textbf{reactive tests}. 
Apparent Limitations of HOL-TestGen

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.

- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .

- HOL has Monads. And therefore means for IO-specifications.
Test-Specification Pattern:

\[
\text{accept trace } \rightarrow \text{P(Mfold trace } \sigma_0 \text{ prog)}
\]

where

\[
\text{Mfold [] } \sigma = \text{Some } \sigma \\
\text{MFold (input::R) = case prog(input, } \sigma \text{) of} \\
\text{None } \Rightarrow \text{None} \\
| \text{Some } \sigma' \Rightarrow \text{Mfold R } \sigma' \text{ prog}
\]

- Can this be used for reactive tests?
Example: A Reactive System I

A toy client-server system:

A channel is requested within a bound $X$, a channel $Y$ is chosen by the server, the client communicates along this channel...
Example: A Reactive System I

- A toy client-server system:

\[
\text{req?X} \rightarrow \text{port!Y[Y < X]} \rightarrow \\
(\text{rec } N. \text{send!D.Y} \rightarrow \text{ack} \rightarrow N \\
\quad \square \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP})
\]

a channel is requested within a bound \(X\), a channel \(Y\) is chosen by the server, the client communicates along this channel . . .
Example: A Reactive System I

- A toy client-server system:

\[
\text{req}\, ?X \rightarrow \text{port}\, !Y[Y < X] \rightarrow \\
(\text{rec}\, N.\, \text{send}\, !D.Y \rightarrow \text{ack} \rightarrow N \\
\quad \Box \, \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP})
\]

a channel is requested within a bound $X$, a channel $Y$ is chosen by the server, the client communicates along this channel . . .

Observation:

$X$ and $Y$ are only known at runtime!
Example: A Reactive System II

Observation:

$X$ and $Y$ are only known at runtime!

- Mfold is a program that manages a state at test run time.
- Use an environment that keeps track of the instances of $X$ and $Y$?

**Infrastructure:** An observer maps **abstract events** (req $X$, port $Y$, ...) in traces to **concrete events** (req 4, port 2, ...) in runs!
Example: A Reactive System

- **Infrastructure:** the observer

  observer rebind substitute postcond ioprog ≡
  \((λ \text{input}. (λ (σ, σ′). let \text{input′}= \text{substitute} \ σ\text{input} \text{in case ioprog \text{input′} \ σ′ of}
    \text{None }⇒ \text{None (\textit{* ioprog failure – eg. timeout ... *)}}
    | \text{Some (output, σ''')} ⇒ let \σ'' = \text{rebind} \ σ\text{output} \text{in (if postcond (σ'',σ''') \text{input′} output then Some(σ'', σ''') else None (\textit{* postcond failure *)}}))))
Example: A Reactive Test IV

- Reactive Test-Specification Pattern:

  \[
  \text{accept } trace \rightarrow \ P(\text{Mfold } trace \sigma_0 (\text{observer rebind subst postcond } ioprog))
  \]

  for reactive systems!
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...
- and derive the necessary rules for symbolic execution based tests ...
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.
The (abstract) IMP syntax is defined in Com.thy.

Com = Main +

**typedef** loc

**type**

val = nat (*arb.*)
state = loc ⇒ val
aexp = state ⇒ val
bexp = state ⇒ bool

**datatype** com =

| SKIP |
| ":==" loc aexp (**infixl** 60) |
| Semi com com ("_ ; _"[60, 60]10) |
| Cond bexp com com |
| 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

\[
\begin{align*}
tm & := \lambda s. 1; \\
sum & := \lambda s. 1; \\
i & := \lambda s. 0; \\
\text{WHILE } \lambda s. (s \text{ sum}) <= (s \text{ a}) \text{ DO} \\
& \quad (i := \lambda s. (s i) + 1; \\
& \quad tm := \lambda s. (s tm) + 2; \\
& \quad \text{sum := } \lambda s. (s tm) + (s \text{ sum}))
\end{align*}
\]

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.

\[
\text{state} \xrightarrow{a := b} \text{state}' \quad \xrightarrow{\text{WHILE \ldots}} \quad \text{state}'' \\
\quad \xrightarrow{\text{SKIP}} \quad \text{state}'''
\]
**IMP Semantics I: (Natural Semantics)**

**Natural semantics going back to Plotkin**

**idea:** programs relates states.

\[
\text{state} \xrightarrow{a \mathrel{:=} b} \text{state}' \xrightarrow{\text{WHILE} ...} \text{state}''' \xrightarrow{\text{SKIP}} \text{state}''
\]

**consts** evalc :: (com × state × state) 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.

\[
\text{state} \xrightarrow{a := b} \text{state'} \quad \text{WHILE} \ldots \quad \text{SKIP} \quad \text{state''} \quad \text{state'''}
\]

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

**translations** "\( \langle c, s \rangle \rightarrow_{c} s' \) " \( \equiv \) "\((c, s, s') \in \text{evalc}\)"
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:

\textbf{inductive} \texttt{evalc}

\texttt{intrs}

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

\texttt{Assign: } \langle x ::= a, s \rangle \xrightarrow{c} s[x \mapsto a \ s]
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 \( \text{update} \ s \ x \ (a \ s) \), 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: $\langle x := a, s \rangle \xrightarrow{c} s[x \mapsto a]$

Note that $s[x \mapsto a]$ is an abbreviation for $\text{update } s \; x \; (a \; s)$, 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 $\text{aexp}$ or $\text{bexp}$.
Excursion: A minimal memory model:

\[(s[x \mapsto E]) \ x = E \]
\[ x \neq y \implies (s[x \mapsto E]) \ y = s \ y \]

This small memory theory contains the \textit{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:

\[ \text{Semi: } [\langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c} s'' ] \implies \langle c; c', s \rangle \xrightarrow{c} s'' \]
The semantics for the sequential composition of statements can be described as follows:

\[
\text{Semi: } \left[ \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c} s'' \right] \implies \langle c; c', s \rangle \xrightarrow{c} s''
\]

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:

\[
\text{Semi: } \left[ \langle c, s \rangle \xrightarrow{c} s'; \langle c', s' \rangle \xrightarrow{c} s'' \right] \implies \langle c; c', s \rangle \xrightarrow{c} s''
\]

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:

\[
\text{Semi: } \left[ \langle c,s \rangle \xrightarrow{c} s' ; \langle c',s' \rangle \xrightarrow{c} s'' \right] \Rightarrow \langle c;c', s \rangle \xrightarrow{c} s''
\]

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:

IfTrue: \[ [ b \; s; \; \langle c, s \rangle \xrightarrow{c} s' ] \]
\[ \xrightarrow{\quad} \langle \text{IF } b \text{ THEN } c \text{ ELSE } c', s \rangle \xrightarrow{c} s' \]

IfFalse: \[ [ \neg b \; s; \; \langle c', s \rangle \xrightarrow{c} s' ] \]
\[ \xrightarrow{\quad} \langle \text{IF } b \text{ THEN } c \text{ ELSE } c', s \rangle \xrightarrow{c} s' \]

WhileFalse: \[ [ \neg b \; s] \]
\[ \xrightarrow{\quad} \langle \text{WHILE } b \text{ DO } c, s \rangle \xrightarrow{c} s \]

WhileTrue: \[ [ b \; s; \; \langle c, s \rangle \xrightarrow{c} s'; \langle \text{WHILE } b \text{ DO } c, s' \rangle \xrightarrow{c} s'' ] \]
\[ \xrightarrow{\quad} \langle \text{WHILE } b \text{ DO } c, s \rangle \xrightarrow{c} s'' \]

Note that for non-terminating programs no final state can be derived!
The **transition semantics** is inspired by abstract machines.
The **transition semantics** is inspired by abstract machines. **Idea**: programs relate “configurations”.

\[
a := b; X, state 
\rightarrow 
X, state' 
\rightarrow 
X'', state''
\rightarrow 
X''', state'''
\]
The **transition semantics** is inspired by abstract machines.

**idea**: programs relate “configurations”.

\[
\begin{align*}
  a & : = b; X, state \quad \rightarrow \quad X, state' \\
  & \quad \rightarrow \quad X'', state'' \\
  & \quad \rightarrow \quad X''', state'''
\end{align*}
\]

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

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

**intro**

Assign: \((x:=a,s) \rightarrow (\text{SKIP}, s[x \mapsto a \ s])\)

Semi1: \((\text{SKIP};c,s) \rightarrow (c,s)\)

Semi2: \((c,s) \rightarrow (c'',s')\)

\[\implies (c;c',s) \rightarrow (c'';c',s')\]
**inductive** evalc1

intro

Assign: $(x:=a,s) \rightarrow (\text{SKIP}, s[x \mapsto a \ s])$

Semi1: $(\text{SKIP};c,s) \rightarrow (c,s)$

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: \((x:=\mathit{a},s) \rightarrow (\text{SKIP}, s[x\mapsto a \ s])\)

Semi1: \((\text{SKIP};c,s) \rightarrow (c,s)\)

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: 
\[ b \text{ s} \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'', s) - 1 \rightarrow (c', s) \]

IfFalse: 
\[ \neg b \text{ s} \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'', s) - 1 \rightarrow (c'', s) \]

WhileFalse: 
\[ \neg b \text{ s} \implies (\text{WHILE } b \text{ DO } c, s) - 1 \rightarrow (\text{SKIP}, s) \]

WhileTrue: 
\[ b \text{ s} \implies (\text{WHILE } b \text{ DO } c, s) - 1 \rightarrow (c; \text{WHILE } b \text{ DO } c, s) \]
IfTrue:

\[ b \; s \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'', s) \xrightarrow{-1} (c', s) \]

IfFalse:

\[ \neg b \; s \implies (\text{IF } b \text{ THEN } c' \text{ ELSE } c'', s) \xrightarrow{-1} (c'', s) \]

WhileFalse:

\[ \neg b \; s \implies (\text{WHILE } b \text{ DO } c, s) \xrightarrow{-1} (\text{SKIP}, s) \]

WhileTrue:

\[ b \; s \implies (\text{WHILE } b \text{ DO } c, s) \xrightarrow{-1} (c; \text{WHILE } b \text{ DO } c, s) \]

A non-terminating loop always leads to successor configurations ...
IMP Semantics III: (Denotational Semantics)

Idea:
**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:

```plaintext
types com_den = (state × state) set
```

and declare the semantic function:

```plaintext
cconsts C :: com ⇒ com_den
```

The semantic function C is defined recursively over the syntax.
**primrec**

\[
\begin{align*}
C(\text{SKIP}) &= \text{Id} \quad (* \equiv \text{identity relation} *) \\
C(x := a) &= \{(s, t). t = s[x \mapsto a, s]\} \\
C(c ; c') &= C(c') \circ C(c) \quad (* \equiv \text{seq. composition} *) \\
C(\text{IF } b \text{ THEN } c' \text{ ELSE } c'') &= \\
&\quad \{(s, t). (s, t) \in C(c') \land b(s)\} \cup \\
&\quad \{(s, t). (s, t) \in C(c'') \land \neg b(s)\} \quad " \\
C(\text{WHILE } b \text{ DO } c) &= \text{lfp } (\Gamma b (C(c))) \quad "
\end{align*}
\]
**primrec**

\[
\begin{align*}
\text{C(SKIP)} & = \text{Id} \\
\text{C(x ::= a)} & = \{(s,t). t = s[x\mapsto a \ s]\} \\
\text{C(c ; c')} & = \text{C(c')} \circ \text{C(c)} \\
\text{C( IF b THEN c' ELSE c'')} & = \{(s,t). (s,t) \in \text{C(c')} \land b(s)\} \cup \\
& \quad \{(s,t). (s,t) \in \text{C(c'')} \land \neg b(s)\}
\end{align*}
\]

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

where:

\[
\Gamma \ b \ c \equiv (\lambda \varphi. \ (s,t). (s,t) \in (\varphi \circ c) \land b(s)) \cup \\
\quad (s,t). s=t \land \neg b(s))
\]

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

\[(c, s) \rightarrow^* (\text{SKIP}, t) = (\langle c, s \rangle \rightarrow^c t)\]

where \(cs \rightarrow^* cs' \equiv (cs, cs') \in \text{evalc1}^*,\) i.e. the new arrow denotes the transitive closure over old one.
IMP Semantics: Theorems I

Theorem: Natural and Transition Semantics Equivalent

\[(c, s) \xrightarrow{\ast} (\text{SKIP}, t) = (\langle c, s \rangle \xrightarrow{c} t)\]

where \(cs \xrightarrow{\ast} cs' \equiv (cs, cs') \in \text{evalc}1^*, \) 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**

\[(c, s) \rightarrow^* (\text{SKIP}, t) = (\langle c, s \rangle \rightarrow_c t)\]

where \(cs \rightarrow^* cs' \equiv (cs, cs') \in \text{evalc}_1^*\), i.e. the new arrow denotes the transitive closure over old one.

**Theorem: Denotational and Natural Semantics Equivalent**

\[((s, t) \in C c) = (\langle c, s \rangle \rightarrow_c t)\]

i.e. all three semantics are closely related!
IMP Semantics: Theorems II

Theorem: Natural Semantics can be evaluated equationally !!!

\[
\begin{align*}
\langle \text{SKIP}, s \rangle \xrightarrow{c} s' &= (s' = s) \\
\langle x := a, s \rangle \xrightarrow{c} s' &= (s' = s[x \mapsto a]) \\
\langle c; c', s \rangle \xrightarrow{c} s' &= (\exists s''. \langle c, s \rangle \xrightarrow{c} s'' \land \langle c', s'' \rangle \xrightarrow{c} s') \\
\langle \text{IF } b \text{ THEN } c \text{ ELSE } c', s \rangle \xrightarrow{c} s' &= (b \, s \land \langle c, s \rangle \xrightarrow{c} s') \lor \\
&(\neg b \, s \land \langle c', s \rangle \xrightarrow{c} s')
\end{align*}
\]

Note: This is the key for evaluating a program symbolically !!!
Example: “a:==2;b:==2*a”

\[ \langle a:==\lambda s. 2; b:==\lambda s. 2 * (s a),s \rangle \xrightarrow{c}s' \]
\[ \equiv (\exists s''. \langle a:==\lambda s. 2,s \rangle \xrightarrow{c}s'' \land \langle b:==\lambda s. 2 * (s a),s'' \rangle \xrightarrow{c}s' \) \]
\[ \equiv (\exists s''. s'' = s[a\mapsto(\lambda s. 2) s] \land s' = s''[b\mapsto(\lambda s. 2 * (s a)) s'']) \]
\[ \equiv (\exists s''. s'' = s[a\mapsto 2] \land s' = s''[b\mapsto 2 * (s'' a)]) \]
\[ \equiv s' = s[a\mapsto 2][b\mapsto 2 * (s[a \mapsto 2] a)] \]
\[ \equiv s' = s[a\mapsto 2][b\mapsto 2 * 2] \]
\[ \equiv s' = s[a\mapsto 2][b\mapsto 4] \]

Note:

1. The $\lambda$-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” $(\exists x. x = e \land P(x)) = P(e)$.
3. This does not work for the loop and for recursion...
Denotational semantics makes it easy to prove facts like:

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

etc.
Program Annotations: Assertions revisited.

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

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

\[
\text{types } \text{assn} = \text{state} \Rightarrow \text{bool} \\
\text{consts } \text{valid} :: (\text{assn} \times \text{com} \times \text{assn}) \Rightarrow \text{bool} ("|= \{\_\} _ \{\_\}"")
\]

\[
\text{defs} \\
| = \{P\}c\{Q\} \equiv \forall s. \forall t. (s,t) \in C(c) \rightarrow P s \rightarrow Q t"
\]

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:

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

$$\implies \models \{P\}c\{Q\}$$

or in more formal, natural-deduction notation:

$$\left[\begin{array}{c}
\langle c, s \rangle \rightarrow_{c} s', P s
\end{array}\right]_{s,s'}$$

$$\cdots$$

$$Q s'$$

$$\models \{P\} c \{Q\}$$

Applied in backwards-inference, this rule generates the constraints for the states that were amenable to equational evaluation rules shown before.
Example: “\( \models \{0 \leq x\} a:=x; b:=2*a\{0 \leq b\} \)”

\[
\models \{ \lambda s. 0 \leq s \, x \} \ a:=\lambda s. \ s \, x; \ b:=\lambda s. \ 2 \ast (s \ a) \ \{ \lambda s. 0 \leq s \ b \} \\
\iff s' = s[a\mapsto s \, x][b\mapsto 2 \ast (s[a\mapsto s \, x] \ a)] \land 0 \leq s \, x \rightarrow 0 \leq s' \ b \\
\equiv s' = s[a\mapsto s \, x][b\mapsto 2 \ast (s \, x)] \land “PRE \ s’” \rightarrow “POST \ s’” \\
\equiv “PRE \ s’” \rightarrow “POST (s[a\mapsto s \, x][b\mapsto 2 \ast (s \, x)])” \\
\]

Note:

- **Note**: the logical constraint

  \( s' = s[a\mapsto s \, x][b\mapsto 2 \ast s \, x] \land 0 \leq s \, x \) 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 $\text{THYP}(\text{PRE } s \rightarrow \text{POST}(s[a \mapsto s \ x][b \mapsto 2 \times (s \ x)]))$
(is: $\text{THYP}(0 \leq s \ x \rightarrow 0 \leq 2 \times s \ x)$) 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:
$\text{THYP}(\exists s. 0 \leq s \ x \rightarrow 0 \leq 2 \times s \ x \rightarrow \forall s. 0 \leq s \ x \rightarrow 0 \leq 2 \times s \ x)$.

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 × com ⇒ 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] ⇒ 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 THENc @@ e ELSEd @@ e)"
   "(WHILE b DO c) @@ e = ((WHILE b DOc);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**

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

$$\forall s, s'. \langle \text{unwind}(n, c), s \rangle \overset{c}{\rightarrow} s' \land P \, s \rightarrow Q \, s'$$

$$\Longrightarrow |\models \{P\} c \{Q\}$$

for an arbitrary (user-defined!) $n$!

Or in natural deduction notation:

$$\begin{array}{c}
\langle \text{unwind}(n, c), s \rangle \rightarrow_c s' \quad P \, s \rightarrow Q \, s' \\
\vdots \\
Q \, s' \\
\hline
|\models \{P\} \, c \, \{Q\}
\end{array}$$
Handling Loops (and Recursion).

Example:

“\(\models \{True\} \text{integer\_sqr\_root} \{i^2 \leq a \land a \leq (i + 1)^2\}\)”

Setting the depth to \(n = 3\) and running the process yields:
Handling Loops (and Recursion).

**Example:**

“\(|= \{\text{True}\} \ \text{integer\textunderscore squareroot} \ \{i^2 \leq a \land a \leq (i + 1)^2\}\)”

Setting the depth to \(n = 3\) and running the process yields:

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

2. \(4 \leq s a; 8 < s a ; s’ = s (i := 2, \ \text{tm} := 5, \ \text{sum} := 9) \rangle \xrightarrow{}\text{post } s’\)

3. \(1 \leq s a; s a < 4; s’ = s (i := 1, \ \text{tm} := 3, \ \text{sum} := 4) \rangle \xrightarrow{}\text{post } s’\)

4. \(s a = 0 ; s’ = s(\ \text{tm} := 1, \ \text{sum} := 1, \ i := 0) \rangle \xrightarrow{}\text{post } s’\)

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. \( \text{THYP} \left( 9 \leqslant s \ a \land \langle \text{WHILE} \ s. \ s \sum \leqslant s \ a \right. \)
    
    \[ \text{DO } i :== \lambda s. \text{Suc} \ (s \ i) ; \]
    
    \[ (\text{tm} :== \lambda s. \text{Suc} \ (\text{Suc} \ (s \ \text{tm}))) ; \]
    
    \[ \text{sum} :== \lambda s. \ s \ \text{tm} + s \ \text{sum} , \]
    
    \[ s(i := 3, \ \text{tm} := 7, \ \text{sum} := 16) \xrightarrow{c} s' \]
    
    \[ \rightarrow \text{post } s' \)
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 ???