An Introduction to MBT with HOL-TestGen

Burkhart Wolff[1]

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

DigiCosme Spring School 2013:
Program Analysis and Verification
Supelec 22-26 April 2013

Outline

Static Functional Test with

Outline

Our First Vision

Testing and proof-based verification may converge,
in a precise technical sense.

We will show this for:

- specification-based (black-box) unit testing
- generation and management of formal test hypothesis
- verification of test hypothesis (not discussed here)

Our Second Vision

- **Observation:**
  Any testcase-generation technique is based on and limited by underlying constraint-solution techniques.
- **Approach:**
  Testing should be integrated in an environment combining **automated and interactive proof techniques**.
- the test engineer must decide over, abstraction level, split rules, breadth and depth of data structure exploration ...
- we mistrust the dream of a **push-button** solution
- byproduct: a **verified** test-tool

Components of HOL-TestGen

- **HOL (Higher-order Logic):**
  - “Functional Programming Language with Quantifiers”
  - plus definitional libraries on Sets, Lists, . . .
  - can be used meta-language for Hoare Calculus for Java, Z, . . .
- **HOL-TestGen:**
  - based on the interactive theorem prover Isabelle/HOL
  - implements these visions
- **Prover IDE and jedit client:**
  - user interface for Isabelle and HOL-TestGen
  - continuous build and contiuous check models ("theories"), test-specifications, test-plans
  - allows to explore the annotation test-plan with types, theorems, test theorems, test data, ...

Components-Overview

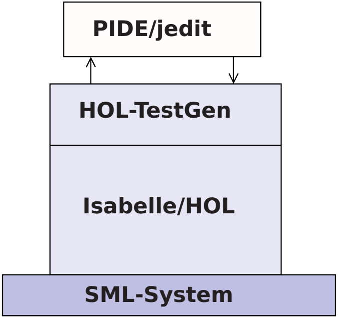


Figure: The Components of HOL-TestGen

A Sample Workflow

The HOL-TestGen Workflow

The HOL-TestGen workflow is basically fivefold:

1. *Step I:* writing a **test theory** (in HOL)
2. *Step II:* writing a **test specification** (in the context of the test theory)
3. *Step III:* generating a **test theorem** (roughly: testcases)
4. *Step IV:* generating **test data**
5. *Step V:* generating a **test script**

And of course:

- building an executable test driver
- and running the test driver

Step I: Writing a Test Theory

- Write **data types** in HOL:

```
theory List_test
imports Testing
begin

datatype 'a list =
    Nil     ("[]")
  | Cons 'a "'a list"     (infixr "#" 65)
```

Step I: Writing a Test Theory

- Write **recursive functions** in HOL:

```
primrec is_sorted:: "('a::ord) list ⇒bool"
where  "is_sorted []        = True"
       "is_sorted (x#xs) = case xs of
                              [] ⇒ True
                            | y#ys ⇒((x < y) ∨(x = y))
                                    ∧ is_sorted xs"
```

Step II: Write a Test Specification

- writing a **test specification** (TS)
  as HOL-TestGen command:

test_spec "is_sorted (prog (l::('a list)))"

Step III: Generating Testcases

- executing the **testcase generator** in form of an Isabelle proof method:

  apply(gen_test_cases "prog")

- concluded by the command:

  store_test_thm "test_sorting"

  . . . that binds the current proof state as **test theorem** to the name `test_sorting`.

Step III: Generating Testcases

- The test theorem contains clauses (the **test-cases**):

  is_sorted (prog [])
  is_sorted (prog [?X1X17])
  is_sorted (prog [?X2X13, ?X1X12])
  is_sorted (prog [?X3X7, ?X2X6, ?X1X5])

- as well as clauses (the **test-hypothesis**):

  $\mathrm{THYP}((\exists x.\ \mathrm{is\_sorted}\ (\mathrm{prog}\ [x])) \longrightarrow (\forall x.\ \mathrm{is\_sorted}(\mathrm{prog}\ [x])))$

  $\ldots$

  $\mathrm{THYP}((\forall l.\ 4 < |l| \longrightarrow \mathrm{is\_sorted}(\mathrm{prog}\ l))$

- We will discuss these hypothesises later in great detail.

Step IV: Test Data Generation

- On the test theorem,
  all sorts of logical massages can be performed.
- Finally, a **test data generator** can be executed:

  gen_test_data "test_sorting"
- The test data generator
  - extracts the testcases from the test theorem
  - searches ground instances satisfying the constraints (none in the example)
- Resulting in test statements like:

  is_sorted (prog [])
  is_sorted (prog [3])
  is_sorted (prog [6, 8])
  is_sorted (prog [0, 10, 1])

Step V: Generating A Test Script

- Finally, a **test script** or **test harness** can be generated:

  gen_test_script "test_lists.sml" list" prog

- The generated test script can be used to test an implementation, e. g., in SML, C, or Java

The Complete Test Theory

```
theory List_test
imports Main begin
  primrec is_sorted:: "('a::ord) list ⇒bool"
  where  "is_sorted  []        = True"
           "is_sorted  (x#xs) = case xs of
                                            []  ⇒ True
                                         |  y#ys ⇒((x < y) ∨(x = y))
                                                  ∧ is_sorted xs"

  test_spec "is_sorted (prog (l::('a list)))"
    apply(gen_test_cases prog)
  store_test_thm "test_sorting"

  gen_test_data "test_sorting"
  gen_test_script "test_lists.sml" list" prog
end
```

Testing an Implementation

Executing the generated test script may result in:

```
Test Results:
Test 0 - *** FAILURE: post-condition false, result: [1, 0, 10]
Test 1 -     SUCCESS, result: [6, 8]
Test 2 -     SUCCESS, result: [3]
Test 3 -     SUCCESS, result: []

Summary:
Number successful tests cases:  3 of 4 (ca. 75%)
Number of warnings:             0 of 4 (ca.  0%)
Number of errors:               0 of 4 (ca.  0%)
Number of failures:             1 of 4 (ca. 25%)
Number of fatal errors:         0 of 4 (ca.  0%)

Overall result: failed
```

A Critical Revision

- **But**
  this is complete rubbish !
- This does **not**
  test what we want: a sorting algorithm.
- ... even a program that just returns the empty list would conform to this test !
- ... we need to revise our test !

Step I: Re-Writing the Test Theory

- We write a reference sorter in HOL:

```
fun  ins :: "('a::linorder) ⇒'a list ⇒ 'a list"
where "ins x [] = [x]"
     |"ins x (y#ys) = (if (x < y) then x#y#ys
                                   else (y#(ins x ys)))"
fun  sort:: "('a::linorder) list ⇒ 'a list"
where "sort [] = [] "
     | "sort (x#xs) = ins x (sort xs)"
```

Step II: Re-Write the Test Specification

- and state as **test specification** (TS) that "prog" should behave like "sort":

test_spec "sort(l) = prog(l)"

Step III: Generating Testcases

- we re-executing the **testcase generator** :

  apply(gen_test_cases "prog")

- concluded by the command:

  store_test_thm "test_sorting2"

  ...that binds the current proof state as **test theorem** to the name `test_sorting2`.

Step III: Generating Testcases

- This time, the test theorem contains the test-cases:

$$
\begin{array}{l}
[\,] \;=\; \mathrm{prog}([\,]) \\
[?X1] = \mathrm{prog}([?X1]) \\
[\![\; ?X1 \le ?X2 \;]\!] \Longrightarrow [?X1, ?X2] = \mathrm{prog}([?X1, ?X2]) \\
[\![\; ?X1 > ?X2 \;]\!] \Longrightarrow [?X2, ?X1] = \mathrm{prog}([?X1, ?X2]) \\
\ldots
\end{array}
$$

- as well as all permutations (without having invented this concept) and the test hypothesis:

$$
\begin{array}{l}
\mathrm{THYP}((\exists\, x.\ [x] = \mathrm{prog}\ [x] \longrightarrow (\forall\, x.\ [x] = \mathrm{prog}\ [x]))) \\
.\ \ .\ \ . \\
\mathrm{THYP}(\forall\, l.\ 4 < |l| \longrightarrow \mathrm{sort}\ l = \mathrm{prog}\ l)
\end{array}
$$

Step IV: Test Data Generation

- On the test theorem,
  all sorts of logical massages can be performed.
- Finally, a **test data generator** can be executed:

  gen_test_data "test_sorting2"

- The test data generator
  - extracts the testcases from the test theorem
  - and produces:
- Resulting in test statements like:

  [] = prog []
  [3] = prog [3]
  [6,8] = prog [6, 8]
  [0,19] = prog [19, 0]
  ...

Outline

The Foundations of HOL-TestGen

- Basis:
  - Isabelle/HOL library: 10000 derived rules, ...
  - about 500 are organized in larger data-structures used by Isabelle's proof procedures, ...
- These Rules were used in advanced proof-procedures for:
  - Higher-Order Rewriting
  - Tableaux-based Reasoning —
    a standard technique in automated deduction
  - Arithmetic decision procedures (Coopers Algorithm)
- gen_testcases is an automated tactical program using combination of them.

Some Rewrite Rules

- Rewriting is a easy to understand deduction paradigm (similar FP) centered around equality
- Arithmetic rules, e. g.,

$$\mathrm{Suc}(x + y) = x + \mathrm{Suc}(y)$$
$$x + y = y + x$$
$$\mathrm{Suc}(x) \neq 0$$

- Logic and Set Theory, e. g.,

$$\forall x.\ (P\,x \wedge Q\,x) = (\forall x.\ P\,x) \wedge (\forall x.\ P\,x)$$
$$\bigcup x \in S.\ (P\,x \cup Q\,x) = (\bigcup x \in S.\ P\,x) \cup (\bigcup x \in S.\ Q\,x)$$
$$[\![A = A'; A \Longrightarrow B = B']\!] \Longrightarrow (A \wedge B) = (A' \wedge B')$$

The Core Tableaux-Calculus

- **Safe Introduction** Rules for logical connectives:

$$\frac{}{t = t} \qquad \frac{}{\text{true}} \qquad \frac{P \quad Q}{P \wedge Q} \qquad \frac{\begin{matrix}[\neg Q]\\ \vdots \\ P\end{matrix}}{P \vee Q} \qquad \frac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \to Q} \qquad \frac{\begin{matrix}[P]\\ \vdots \\ \text{false}\end{matrix}}{\neg P} \qquad \ldots$$

- **Safe Elimination** Rules:

$$\frac{\text{false}}{P} \qquad \frac{P \wedge Q \quad \begin{matrix}[P, Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{P \vee Q \quad \begin{matrix}[P]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \frac{P \to Q \quad \begin{matrix}[\neg P]\\ \vdots \\ R\end{matrix} \quad \begin{matrix}[Q]\\ \vdots \\ R\end{matrix}}{R} \qquad \ldots$$

The Core Tableaux-Calculus

- Safe Introduction Quantifier rules:

$$\frac{P\ ?x}{\exists x.\ P\ x} \qquad \frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}$$

- Safe Quantifier Elimination

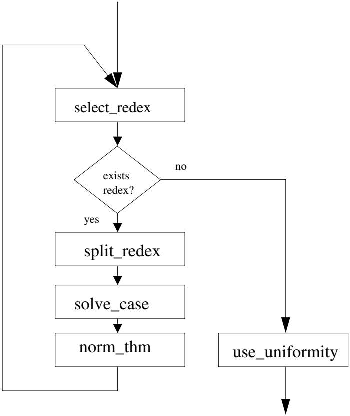
$$\frac{\exists x.\ P\ x \qquad \bigwedge x.\ \begin{matrix}[P\ x] \\ \vdots \\ Q\end{matrix}}{Q}$$

- Critical Rewrite Rule:

$$\text{if } P \text{ then } A \text{ else } B = (P \to A) \wedge (\neg P \to B)$$

The Generic Procedure

`gen_test_cases :`



Chooser: selects a splitting redex (e.g. free variables)

Splitter: applies splitting rules (e.g. regularity hypothesis, see below)

Normalizer: Applies global simplification and tableaux calculi of *E*, i. e. the previously decribed underlying ruleset

Solver: Attempts to eliminate unsatisfiable constraints

Finalizer: Applies minimization and uniformity hypothesis (see below).

Explicit Test Hypothesis: The Concept

- What to do with infinite data-strucutures?
- What is the connection between test-cases and test statements and the test theorems?
- Two problems, one answer: Introducing test hypothesis "on the fly":

$$\mathrm{THYP} : \mathrm{bool} \Rightarrow \mathrm{bool}$$
$$\mathrm{THYP}(x) \equiv x$$

Taming Infinity I: Regularity Hypothesis

- What to do with infinite data-strucutures of type τ?
  Conceptually, we split the set of all data of type τ into

$$\{x :: \tau \mid |x| < k\} \cup \{x :: \tau \mid |x| \geq k\}$$

Taming Infinity I: Motivation

Consider the first set $\{X :: \tau \mid |x| < k\}$
for the case $\tau = \alpha$ list, $k = 2, 3, 4$.
These sets can be presented as:

1) $|x::\tau|<2 = (x = []) \vee (\exists\ a.\ x = [a])$
2) $|x::\tau|<3 = (x = []) \vee (\exists\ a.\ x = [a])$
$\vee (\exists\ a\ b.\ x = [a,b])$
3) $|x::\tau|<4 = (x = []) \vee (\exists\ a.\ x = [a])$
$\vee (\exists\ a\ b.\ x = [a,b]) \vee (\exists\ a\ b\ c.\ x = [a,b,c])$

Taming Infinity I: Data Separation Rules

This motivates the (derived) data-separation rule:

- ($\tau = \alpha$ list, $k = 3$):

$$\frac{\begin{array}{c}[x = []] \\ \vdots \\ P\end{array} \quad \bigwedge a. \begin{array}{c}[x = [a]] \\ \vdots \\ P\end{array} \quad \bigwedge a\, b. \begin{array}{c}[x = [a,b]] \\ \vdots \\ P\end{array} \quad \text{THYP}\ M}{P}$$

- Here, M is an abbreviation for:

$\forall$ x. k < |x| $\longrightarrow$P x

Taming Infinity II: Uniformity Hypothesis

- What is the connection between test cases and test statements and the test theorems?
- Well, the “uniformity hypothesis”:
- *Once the program behaves correct for one test case, it behaves correct for all test cases ...*

Taming Infinity II: Uniformity Hypothesis

- Using the **uniformity hypothesis**, a test case:

  n) $[\![\text{C1 x}; \ \ldots; \ \text{Cm x}]\!] \Longrightarrow \text{TS x}$

  is transformed into:

  n) $[\![\text{C1 ?x}; \ \ldots; \ \text{Cm ?x}]\!] \Longrightarrow \text{TS ?x}$

  n+1) $\text{THYP}((\exists \text{x. C1 x} \ldots \text{Cm x} \longrightarrow \text{TS x})$
  $\longrightarrow (\forall \text{x. C1 x} \ldots \text{Cm x} \longrightarrow \text{TS x}))$

Testcase Generation by NF Computations

Test-theorem is computed out of the test specification by

- a heuristicts applying **Data-Separation Theorems**
- a **rewriting** normal-form computation
- a **tableaux-reasoning** normal-form computation
- **shifting** variables referring to the program under test prog test into the conclusion, e.g.:

  $$[\![\neg(\text{prog } x = c); \neg(\text{prog } x = d)]\!] \Longrightarrow A$$

  is transformed equivalently into

  $$[\![\neg A]\!] \Longrightarrow (\text{prog } x = c) \vee (\text{prog } x = d)$$

- as a final step, all resulting clauses were normalized by applying uniformity hypothesis to each free variable.

A Sample Derivation of a Test Theorem

Testcase Generation: An Example

```
theory TestPrimRec
imports Main
begin
primrec
  x mem []    = False
  x mem (y#S) = if y = x
                then True
                else x mem S

test_spec:
  "x mem S ⟹prog x S"
apply(gen_testcase 0 0)
```

1) prog x [x]
2) $\bigwedge$b. prog x [x,b]
3) $\bigwedge$a. a$\neq$x$\Longrightarrow$prog x [a,x]
4) THYP(3 $\leq$size (S)
$\longrightarrow\forall$ x. x mem S
$\longrightarrow$prog x S)

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

Sample Derivation of Test Theorems

Example

$$\text{x mem S} \longrightarrow \text{prog x S}$$

is transformed via data-separation lemma to:

1. $\text{S=[]} \Longrightarrow \text{x mem S} \longrightarrow \text{prog x S}$
2. $\bigwedge \text{a. S=[a]} \Longrightarrow \text{x mem S} \longrightarrow \text{prog x S}$
3. $\bigwedge \text{a b. S=[a,b]} \Longrightarrow \text{x mem S} \longrightarrow \text{prog x S}$
4. $\text{THYP}(\forall\ \text{S. } 3 \leq |\text{S}| \longrightarrow \text{x mem S} \longrightarrow \text{prog x S})$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

canonization leads to:

1. $x \text{ mem } [] \Longrightarrow \text{prog } x\ []$
2. $\bigwedge a.\ x \text{ mem } [a] \Longrightarrow \text{prog } x\ [a]$
3. $\bigwedge a\ b.\ x \text{ mem } [a,b] \Longrightarrow \text{prog } x\ [a,b]$
4. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

which is reduced via the equation for mem:

1. $\text{false} \Longrightarrow \text{prog } x\ []$

2. $\bigwedge a.\ \text{if } a = x \text{ then True}$
   $\quad\quad \text{else } x \text{ mem } [] \Longrightarrow \text{prog } x\ [a]$
3. $\bigwedge a\ b.\ \text{if } a = x \text{ then True}$
   $\quad\quad \text{else } x \text{ mem } [b] \Longrightarrow \text{prog } x\ [a,b]$
4. $\text{THYP}(3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

erasure for unsatisfyable constraints and rewriting conditionals yields:

2. $\bigwedge a.\ a = x \vee (a \neq x \wedge \text{false}) \Longrightarrow \text{prog } x\ [a]$
3. $\bigwedge a\ b.\ a = x \vee (a \neq x \wedge x \text{ mem } [b]) \Longrightarrow \text{prog } x\ [a,b]$
4. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

...which is further reduced by tableaux rules and canconization to:

2. $\bigwedge a.\ \text{prog } a\ [a]$

3. $\bigwedge a\ b.\ a = x \Longrightarrow \text{prog } x\ [a,b]$

3'. $\bigwedge a\ b.\ [\![\ a \neq x;\ x \text{ mem } [b]\ ]\!] \Longrightarrow \text{prog } x\ [a,b]$

4. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow prog\ x\ S$

...which is reduced by canonization and rewriting of mem to:

2. $\bigwedge a.\ prog\ x\ [x]$

3. $\bigwedge a\ b.\ prog\ x\ [x,b]$

3'. $\bigwedge a\ b.\ a \neq x \Longrightarrow prog\ x\ [a,x]$

4. $THYP(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow prog\ x\ S)$

Sample Derivation of Test Theorems

Example

$x \text{ mem } S \longrightarrow \text{prog } x\ S$

...as a final step, uniformity is expressed:

1. $\text{prog } ?x1\ [?x1]$
2. $\text{prog } ?x2\ [?x2,?b2]$
3. $?a3 \neq ?x1 \Longrightarrow \text{prog } ?x3\ [?a3,?x3]$
4. $\text{THYP}(\exists x.\text{prog } x\ [x] \longrightarrow \text{prog } x\ [x]$

   ...

7. $\text{THYP}(\forall\ S.\ 3 \leq |S| \longrightarrow x \text{ mem } S \longrightarrow \text{prog } x\ S)$

A Sample Derivation of a Test Theorem

Test Case Generation (I)

The test-theorem for a test specification TS has the general form:

$$[\![TC_1; \ldots; TC_n; \mathrm{THYP}\ H_1; \ldots; \mathrm{THYP}\ H_m]\!] \Longrightarrow TS$$

where the **test cases** TC_i have the form:

$$\exists x. C_1 x \wedge \ldots \wedge C_m x \Longrightarrow P\ x\ (prog\ x)$$

and where the **test-hypothesis** are either uniformity or regularity hypothethises.
The C_i in a test case were also called **constraints** of the testcase.

Test Case Generation (II)

- The overall meaning of the test-theorem is:
  - **if** the program passes the tests for all test-cases,
  - and **if** the test hypothesis are valid for *PUT*,
  - **then** *PUT* complies to testspecification *TS*.
- **Thus, the test-theorem establishes a formal link between test and verification !!!**

Using Constraint Solving

Test data generation is now a constraint satisfaction problem.

- We eliminate the existential quantifiers (or equivalently: the meta variables ?x , ?y, . . .) by constructing values (“ground instances”) satisfying the constraints. This is done by:
  - random testing (for a smaller input space!!!)
  - arithmetic decision procedures
  - reusing pre-compiled abstract test cases
  - . . .
  - interactive simplify and check, if constraints went away!
- Output: Sets of instantiated test theorems (to be converted into Test Driver Code)

Correctness of a Test-Theorem

A Test-Theorem is *correct* iff the implication:

$$TC_1 \wedge \ldots \wedge TC_n \wedge \mathrm{THYP}\ H_1 \wedge \ldots \wedge \mathrm{THYP}\ H_m \Longrightarrow TS$$

is logically valid.
Well, actually correctness is assumed if we speak of a correctness-*theorem*.

Completeness of a Test-Theorem

A Test-Theorem is *complete* iff the implication:

$$TS \Longrightarrow (TC_1 \wedge \ldots \wedge TC_n \wedge \mathrm{THYP}\ \ H_1 \wedge \ldots \wedge \mathrm{THYP}\ \ H_m)$$

is logically valid.

Minimality of a Test-Theorem

A Test-Theorem is *minimal* iff the test cases are pairwise disjoint, i. e.

$$\{x.Ci_1x \wedge \ldots \wedge Ci_mx\} \cap \{x.Cj_1x \wedge \ldots \wedge Cj_nx\} = \{\}$$

is logically valid for all $i \neq j$. This means that the partitions of input are disjoint.

Theoretical Properties: The Case for HOL-TestGen

- generated test-theorems are correct by construction
- … and complete (by meta-theoretic arguments)
- … but not necessarily minimal (although, in practice, for data-type-oriented specifications, not far from minimality).

Outline

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:
  $\Longrightarrow$ 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.

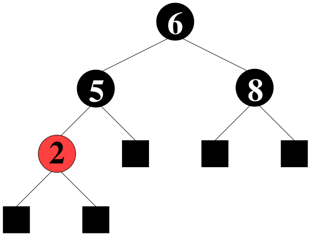
- part of the SML standard library
- widely used internally in the sml/NJ compiler, e. g., for providing efficient implementation for Sets, Bags, . . . ;
- very hard to generate (balanced) instances randomly

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 $\Rightarrow$ bool

blackinv :: tree $\Rightarrow$ bool

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

blackinv E = True

blackinv (T color a y b) =

((blackinv a) $\wedge$(blackinv b)

$\wedge$ ((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 $\wedge$ redinv t $\wedge$blackinv t
  $\wedge$ isin (y::int) t
  $\longrightarrow$
  (blackinv(prog(y,t)))"

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

- Using the standard-workflows results, among others:

  RSF $\longrightarrow$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.

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.
- **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.

$$(\text{max\_B\_height}\ (T\ x\ t1\ val\ t2) = 0) \Longrightarrow (x = R)$$

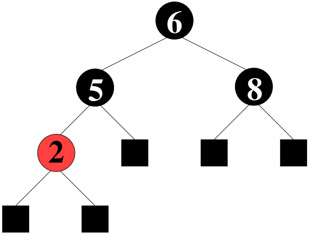
$$\begin{aligned}(\text{max\_B\_height}\ x = 0) &= \\ (x = E \vee \exists\, a\ y\ b.\ x = T\ R\ a\ y\ b \wedge &\ \text{max}(\text{max\_B\_height}\ a) \\ &\quad (\text{max\_B\_height}\ b) = 0)\end{aligned}$$

- 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**.

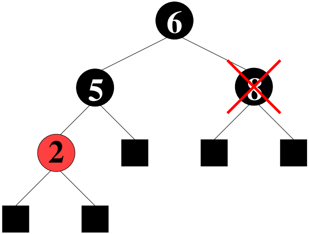
Red-black Trees: sml/NJ Implementation



(a) pre-state

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

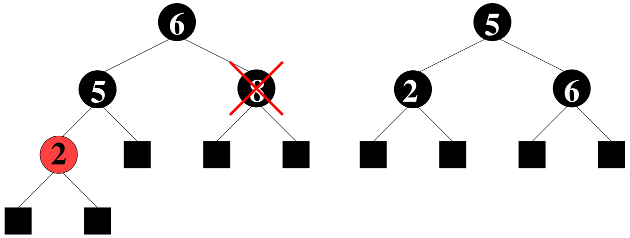
Red-black Trees: sml/NJ Implementation



(b) pre-state: delete "8"

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

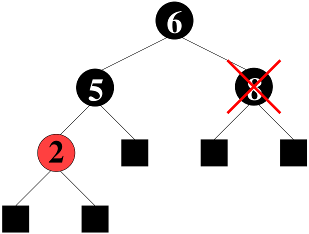
Red-black Trees: sml/NJ Implementation



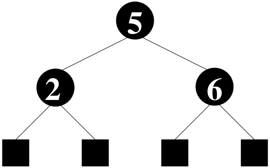
(b) pre-state: delete “8” (c) correct result

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

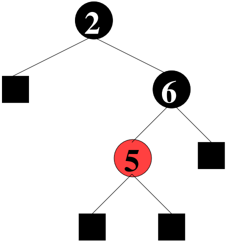
Red-black Trees: sml/NJ Implementation



(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:

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

- 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 *prog*.
- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .
- No possibility to describe **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.

Representing Sequence Test

- Test-Specification Pattern:

  accept trace $\rightarrow$P(Mfold trace σ_0 prog)

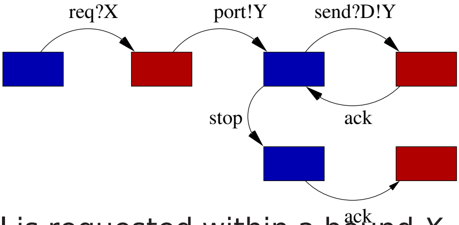
  where

  Mfold [] σ = Some σ
  MFold (input::R) = **case** prog(input, σ) **of**
  None $\Rightarrow$ None
  | Some σ‘$\Rightarrow$ Mfold R σ’ 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:

$$
\begin{aligned}
&\mathrm{req}?X \rightarrow \mathrm{port}!Y[Y < X] \rightarrow \\
&\qquad (\mathrm{rec}\, N.\ \mathrm{send}!D.Y \rightarrow \mathrm{ack} \rightarrow N \\
&\qquad\qquad \Box\, \mathrm{stop} \rightarrow \mathrm{ack} \rightarrow \mathrm{SKIP})
\end{aligned}
$$

  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:

$$
\begin{aligned}
&\mathrm{req}?X \rightarrow \mathrm{port}!Y[Y < X] \rightarrow \\
&\qquad\qquad (\mathrm{rec}\, N.\, \mathrm{send}!D.Y \rightarrow \mathrm{ack} \rightarrow N \\
&\qquad\qquad\qquad\qquad\qquad \Box\, \mathrm{stop} \rightarrow \mathrm{ack} \rightarrow \mathrm{SKIP})
\end{aligned}
$$

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 $\equiv$

(λ input. (λ (σ, σ'). **let** input'= substitute σinput **in**

case ioprog input' σ' **of**

None $\Rightarrow$None *(∗ ioprog failure − eg. timeout ... ∗)*

| Some (output, σ''') $\Rightarrow$**let** σ'' = rebind σoutput **in**

(if postcond (σ'',σ''') input' output

then Some(σ'', σ''')

else None *(∗ postcond failure ∗)*)))"

Example: A Reactive Test IV

- Reactive Test-Specification Pattern:

  $$\text{accept}\, \mathit{trace} \longrightarrow$$
  $$P(\text{Mfold}\, \mathit{trace}\, \sigma_0\, (\text{observer rebind}\, \text{subst postcond}\, \mathit{ioprog}))$$

- for reactive systems!

Motivation

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

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

- 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.

IMP Syntax

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

```
Com = Main +
typedecl loc
types
  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
                 (" 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 !

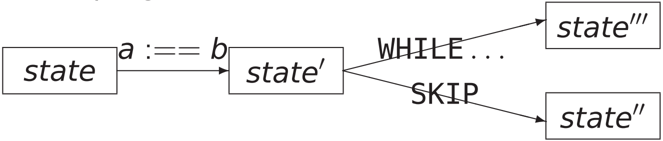
IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

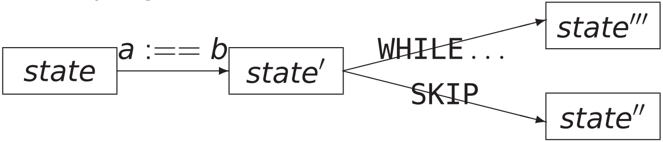
idea: programs relates states.



IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

idea: programs relates states.



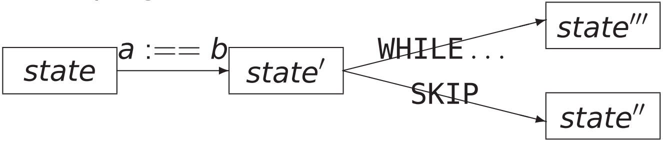
consts evalc :: (com $\times$state $\times$state) set

translations "$\langle$c,s$\rangle$ $\underset{c}{\longrightarrow}$ s’ " $\equiv$ "(c,s,s’) $\in$evalc"

IMP Semantics I: (Natural Semantics

Natural semantics going back to Plotkin

idea: programs relates states.



consts evalc :: (com $\times$state $\times$state) set

translations "$\langle$c,s$\rangle$ $\underset{c}{\longrightarrow}$ s’ " $\equiv$ "(c,s,s’) $\in$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: $\langle \text{SKIP,s} \rangle \xrightarrow[c]{} \text{s}$

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

The natural semantics as inductive definition:

inductive evalc

intrs

Skip: $\langle \mathrm{SKIP}, s \rangle \xrightarrow[c]{} s$

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

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

The natural semantics as inductive definition:

inductive evalc
  intrs
  Skip: $\langle$SKIP,s$\rangle$ $\underset{c}{\longrightarrow}$ s
  Assign: $\langle$x :== a,s$\rangle$ $\underset{c}{\longrightarrow}$ s[x $\mapsto$a s]

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

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

Note that a is of type aexp or bexp.

Excursion: A minimal memory model:

$$(s[x \mapsto E])\ x = E$$

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

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:

Semi: $[\![\langle c,s\rangle \xrightarrow[c]{} s';\ \langle c',s'\rangle \xrightarrow[c]{} s''\ ]\!] \Longrightarrow \langle c;c',\ s\rangle \xrightarrow[c]{} s''$

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

Semi: $[\![\langle c,s\rangle \underset{c}{\rightarrow} s'; \ \langle c',s'\rangle \underset{c}{\rightarrow} s'' \ ]\!] \Longrightarrow \langle c;c',\ s\rangle \underset{c}{\rightarrow} s''$

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:

$$\text{Semi: } [\![\langle c,s\rangle \xrightarrow[c]{} s';\ \langle c',s'\rangle \xrightarrow[c]{} s'']\!] \Longrightarrow \langle c;c',\ s\rangle \xrightarrow[c]{} s''$$

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:

$$\text{Semi: } [\![\langle c,s \rangle \xrightarrow[c]{} s';\ \langle c',s' \rangle \xrightarrow[c]{} s'']\!] \Longrightarrow \langle c;c',\ s \rangle \xrightarrow[c]{} s''$$

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: $[\![\ b\ s;\ \langle c,s\rangle \xrightarrow[c]{} s'\ ]\!]$
$\Longrightarrow \langle\ \mathtt{IF}\ \ b\ \mathtt{THEN}\ c\ \mathtt{ELSE}\ c',\ s\rangle \xrightarrow[c]{} s'$

IfFalse: $[\![\ \neg b\ s;\ \langle c',s\rangle \xrightarrow[c]{} s'\ ]\!]$
$\Longrightarrow \langle\ \mathtt{IF}\ \ b\ \mathtt{THEN}\ c\ \mathtt{ELSE}\ c',\ s\rangle \xrightarrow[c]{} s'$

WhileFalse: $[\![\neg b\ s]\!]$
$\Longrightarrow \langle \mathtt{WHILE}\ \ b\ \mathtt{DO}\ \ c,\ s\rangle \xrightarrow[c]{} s$

WhileTrue: $[\![\ b\ s;\ \langle c,s\rangle \xrightarrow[c]{} s';\langle \mathtt{WHILE}\ \ b\ \mathtt{DO}\ \ c,s'\rangle \xrightarrow[c]{} s''\ ]\!]$
$\Longrightarrow \langle \mathtt{WHILE}\ \ b\ \mathtt{DO}\ \ c,\ s\rangle \xrightarrow[c]{} s''$

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".

$$
\begin{tikzcd}
 & & \boxed{X'', state''} \\
\boxed{a :== b; X, state} \arrow[r] & \boxed{X, state'} \arrow[ur] \arrow[dr] & \\
 & & \boxed{X''', state'''}
\end{tikzcd}
$$

IMP Semantics II: (Transition Semantics)

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

idea: programs relate “configurations”.

$$\begin{tikzcd}
 & & \boxed{X'', state''} \\
\boxed{a :== b; X, state} \arrow[r] & \boxed{X, state'} \arrow[ur] \arrow[dr] & \\
 & & \boxed{X''', state'''}
\end{tikzcd}$$

consts evalc1 :: ((com $\times$state) $\times$(com $\times$state)) set

translations "cs $-$1$-$> cs’" $\equiv$"(cs,cs’) $\in$evalc1"

inductive evalc1

intro

Assign: (x:==a,s) −1−> (SKIP, s[x ↦a s])

Semi1: (SKIP;c,s) −1−> (c,s)

Semi2: (c,s) −1−> (c'',s')

$\Longrightarrow$ (c;c',s) −1−> (c'';c',s')

inductive evalc1

intro

```
Assign:  (x:==a,s) −1−> (SKIP, s[x ↦a s])
Semi1:   (SKIP;c,s) −1−> (c,s)
Semi2:   (c,s)      −1−> (c'',s')
           ⟹  (c;c',s) −1−> (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) −1−> (SKIP, s[x ↦a s])
  Semi1: (SKIP;c,s) −1−> (c,s)
  Semi2: (c,s) −1−> (c'',s')
    ⟹ (c;c',s) −1−> (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:

$b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c',s)

IfFalse:

$\neg b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c'',s)

WhileFalse:

$\neg b\ s \Longrightarrow$ (WHILE b DO c,s) −1−> (SKIP,s)

WhileTrue:

$b\ s \Longrightarrow$ (WHILE b DO c,s) −1−> (c;WHILE b DOc,s)

IfTrue:

$b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c', s)

IfFalse:

$\neg b\ s \Longrightarrow$ (IF b THEN c' ELSE c'', s) −1−> (c'', s)

WhileFalse:

$\neg b\ s \Longrightarrow$ (WHILE b DO c, s) −1−> (SKIP, s)

WhileTrue:

$b\ s \Longrightarrow$ (WHILE b DO c, s) −1−> (c;WHILE b 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 $\times$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 com_den = (state $\times$state) set
and declare the semantic function:

consts C :: com $\Rightarrow$com_den

The semantic function C is defined recursively over the syntax.

primrec

C(SKIP) = Id (* ≡identity relation *)

C(x :== a) = {(s,t). t = s[x↦ a s]}

C(c ; c') = C(c') O C(c) (* ≡seq. composition *)

C(IF b THEN c' ELSE c'') =

{(s,t). (s,t) ∈C(c') ∧b(s)} ∪

{(s,t). (s,t) ∈C(c'') ∧¬b(s)}"

C(WHILE b DO c) = lfp (Γ b (C(c)))"

primrec

C(SKIP) = Id (* ≡identity relation *)

C(x :== a) = {(s,t). t = s[x↦ a s]}

C(c ; c') = C(c') O C(c) (* ≡seq. composition *)

C(IF b THEN c' ELSE c'') =

{(s,t). (s,t) ∈C(c') ∧b(s)} ∪

{(s,t). (s,t) ∈C(c'') ∧¬b(s)}"

C(WHILE b DO c) = lfp (Γ b (C(c)))"

where:

Γ b c ≡(λφ. {(s,t). (s,t) ∈(φ O c) ∧b(s)} ∪

{(s,t). s=t ∧¬b(s)})

and where the least-fixpoint-operator *lfp F* corresponds in this special case to:

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

IMP Semantics:Theorems I

Theorem: Natural and Transition Semantics Equivalent

$(c,\ s)$ −∗−> (SKIP, t) = $(\langle c,s\rangle \xrightarrow[c]{} t)$

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

IMP Semantics:Theorems I

Theorem: Natural and Transition Semantics Equivalent

$$(c,\ s)\ -{*}-\!>\ (\mathrm{SKIP}, t) = (\langle c,s\rangle \xrightarrow[c]{} t)$$

where $cs\ -{*}-\!>\ cs' \equiv (cs,cs') \in evalc1^{*}$, 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

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

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

Theorem: Denotational and Natural Semantics Equivalent

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

i.e. all three semantics are closely related !

IMP Semantics:Theorems II

Theorem: Natural Semantics can be evaluated equationally !!!

$$\langle \mathrm{SKIP}, s\rangle \xrightarrow[c]{} s' = (s' = s)$$

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

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

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

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

Example: “a:==2;b:==2*a”

$$\langle \text{a:==}\lambda \text{s. 2; b:==}\lambda \text{s. } 2 * (\text{s a}),\text{s}\rangle \xrightarrow[c]{} \text{s'}$$

$$\equiv (\exists \text{s''}.\ \langle \text{a:==}\lambda \text{s. } 2,\text{s}\rangle \xrightarrow[c]{} \text{s''} \wedge \langle \text{b:==}\lambda \text{s. } 2 * (\text{s a}),\text{s''}\rangle \xrightarrow[c]{} \text{s'})$$

$$\equiv (\exists \text{s''}.\ \text{s''} = \text{s[a}\mapsto(\lambda \text{s. } 2)\ \text{s]} \wedge \text{s'} = \text{s''[b}\mapsto(\lambda \text{s. } 2 * (\text{s a}))\ \text{s'']})$$

$$\equiv (\exists \text{s''}.\ \text{s''} = \text{s[a}\mapsto 2] \wedge \text{s'} = \text{s''[b}\mapsto 2 * (\text{s'' a})])$$

$$\equiv \text{s'} = \text{s[a}\mapsto 2][\text{b}\mapsto 2 * (\text{s[a}\mapsto 2]\ \text{a})]$$

$$\equiv \text{s'} = \text{s[a}\mapsto 2][\text{b}\mapsto 2 * 2]$$

$$\equiv \text{s'} = \text{s[a}\mapsto 2][\text{b}\mapsto 4]$$

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 (WHILE b DO c) = C (IF b THEN c; WHILE b DO c ELSE SKIP)
C (SKIP ; c) = C(c)
C (c; SKIP) = C(c)
C ((c ; d); e) = C(c;(d;e))
C ((IF b THEN c ELSE d); e) = C(IF b THEN c ; e 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 assn = state $\Rightarrow$bool
consts valid :: (assn $\times$com $\times$assn) $\Rightarrow$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 correctnes; 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:

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

or in more formal, natural-deduction notation:

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

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

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

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

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

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

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

Note:

- **Note**: the logical constaint $s' = s[a \mapsto s\ x][b \mapsto 2 * s\ x] \wedge 0 \leq s\ x$ 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 $THYP(PRE\ s \to POST(s[a \mapsto s\ x][b \mapsto 2 * (s\ x)]))$ (is: $THYP(0 \leq s\ x \to 0 \leq 2 * s\ x)$) 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: $THYP(\exists s.\ 0 \leq s\ x \to 0 \leq 2 * s\ x \to \forall s.\ 0 \leq s\ x \to 0 \leq 2 * s\ x)$.

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 ×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) ELSEunwind(n
"unwind(n, WHILEb DO c) =
    if  0 < n
      then IF  b THEN unwind(n,c)@@unwind(n− 1,WHILE b DOc) ELSESKI
      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) and C(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:

$$\forall\, s\ s'.\ \langle\ \text{unwind}(n,c)\ ,s\rangle \xrightarrow[c]{} s' \wedge \quad P\ s \to Q\ s'$$
$$\Longrightarrow \models \{P\}c\{Q\}$$

for an arbitrary (user-defined!) n !

Or in natural deduction notation:

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

Handling Loops (and Recursion).

Example:

“$\models \{True\}\ integer\_squareroot\ \{i^2 \leq a \wedge a \leq (i+1)^2\}$**”**

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

Handling Loops (and Recursion).

Example:

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

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

1. ⟦ 9 ≤s a; ⟨`WHILE` λs. s sum ≤s a
   `DO` i :== λs. Suc (s i) ;
   (tm :== λs. Suc (Suc (s tm)) ;
   sum :== λs. s tm + s sum),
   s(i := 3, tm := 7, sum := 16)⟩ $\xrightarrow[c]{}$s’
   ⟧ ⟹post s’
2. ⟦ 4 ≤s a; 8 < s a ; s’ = s (i := 2, tm := 5, sum := 9) ⟧ ⟹post s’
3. ⟦ 1 ≤s a; s a < 4; s’ = s (i := 1, tm := 3, sum := 4) ⟧ ⟹post s’
4. ⟦ s a = 0 ; s’ = s(tm := 1, sum := 1, i := 0) ⟧ ⟹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. $\text{THYP}(9 \leq \text{s a} \wedge \langle \texttt{WHILE}\ \lambda s.\ \text{s sum} \leq \text{s a}$
   $\texttt{DO}\ \ i :== \lambda s.\ \text{Suc (s i)} ;$
   $(\text{tm} :== \lambda s.\ \text{Suc (Suc (s tm))} ;$
   $\text{sum} :== \lambda s.\ \text{s tm} + \text{s sum}),$
   $s(i := 3, \text{tm} := 7, \text{sum} := 16)\rangle \xrightarrow[c]{} s'$
   $\rightarrow \text{post } s')$

... 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

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.
- This seems to exclude testing of systems with internal state.

Motivation: Sequence Test Example I

Example: A little Bank - Acount System.

internal var register : table[client, nat]integer

op deposit (c : client, no : account_no, amount:nat) : unit
    pre (c,no) : dom(register)
    post register'=register[(c,no) := register(c,no) + amount]

op balance (c : client, no : account_no) : int
    pre (c,no) : dom(register)
    post register'=register and result = register(c,no)

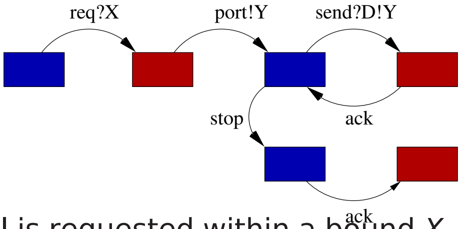
op withdraw(c : client, no : account_no, amount:nat) : unit
    pre (c,no) : dom(register) and register(c,no) >= amount
    post register'=register[(c,no) := register(c,no) − amount]

Motivation: Sequence Test Example II

1. **Problem:** Only the public interface (i. e. the operations deposit, balance and withdraw. The internal (hidden) state is not accessible.
2. **Problem:** we can therefore only control the state by *sequences* of operation calls, not just produce data and leave it to one operation call as in uit tests.
3. **Problem:** The spec does not speak about the initial states.

Motivation: A Reactive System Example 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 . . .

Motivation: A Reactive System Example I

- A toy client-server system:

$$\begin{aligned} &\mathrm{req}?X \rightarrow \mathrm{port}!Y[Y < X] \rightarrow \\ &\qquad (\mathrm{rec}\, N.\, \mathrm{send}!D.Y \rightarrow \mathrm{ack} \rightarrow N \\ &\qquad\qquad \Box\, \mathrm{stop} \rightarrow \mathrm{ack} \rightarrow \mathrm{SKIP}) \end{aligned}$$

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

Motivation: A Reactive System Example I

- A toy client-server system:

$$\begin{aligned} &\mathsf{req}?X \to \mathsf{port}!Y[Y < X] \to \\ &\qquad (\mathsf{rec}\,N.\,\mathsf{send}!D.Y \to \mathsf{ack} \to N \\ &\qquad\qquad \Box\,\mathsf{stop} \to \mathsf{ack} \to \mathsf{SKIP}) \end{aligned}$$

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

Motivation: A Reactive System Example II

Observation:

X and Y are only known at runtime!

- a test-driver is needed that manages a serialization of tests at test run time.
- ... including 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!

Apparent Limitations of HOL-TestGen

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

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

- **No Non-determinism**.

Apparent Limitations of HOL-TestGen

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

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

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

Apparent Limitations of HOL-TestGen

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

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

- 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

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

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

- 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

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

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

- 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 . . .
- No possibility to describe **reactive tests**.

Apparent Limitations of HOL-TestGen

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

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

- 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.

Outline

The core of state-based computations:

state transitions from state σ to σ' emmiting output out!

$$\boxed{\sigma} \xrightarrow{\mathit{out}!} \boxed{\sigma'}$$

Such state-transitions can be modeled in various ways:

- as total functions: $\sigma \Rightarrow (\mathrm{o} \times \sigma)$
- as partial functions: $\sigma \Rightarrow (\mathrm{o} \times \sigma)\ \mathrm{option}$
- as relations: $\sigma \Rightarrow (\mathrm{o} \times \sigma)\ \mathrm{set}$
- as finite series relation: $\sigma \Rightarrow (\mathrm{o} \times \sigma)\ \mathrm{list}$
- as infinite series relation: $\sigma \Rightarrow (\mathrm{o} \times \sigma)\ \mathrm{sequence}$
- ...

We write for this form of type scheme $(o, \sigma)Mon_\phi$ for ϕ in $\{, option, set, list, \ldots\}$. Note that $(o, \sigma)Mon_\phi$ in itself is not a type in the Isabelle type-system (only the instances thereof).

Background:

If a type $(o, \sigma)Mon_\phi$ is completed to an algebraic stucture with two operations :

$$\texttt{bind}_\phi :: [(\alpha, \sigma)Mon_\phi, \alpha \Rightarrow (\beta, \sigma)Mon_\phi] \Rightarrow (\beta, \sigma)Mon_\phi$$

and

$$\texttt{unit}_\phi :: \alpha \Rightarrow (\alpha, \sigma)Mon_\phi$$

satisfying the associativity and both neutrality laws:

1. **associativity**:
   $\texttt{bind}_\phi\ F\ (\lambda y.\texttt{bind}_\phi\ G\ H) = \texttt{bind}_\phi(\texttt{bind}_\phi\ F\ (\lambda y.\texttt{bind}_\phi\ G)\ H$
2. **neutrality_left**: $\texttt{bind}_\phi\ (unit\ F)\ G = G$
3. **neutrality_right**: $\texttt{bind}_\phi\ F(unitG) = F$

What is the Relevance for Computing?

1. Monads talk of the sequential "glue", the _;_ and `resulte` in imperative languages.
2. Monads are an abstraction of "computational structures" arranging computations based on an underying state. This can be used in (for example):
    1. computations based on state
    2. computations based on state involving exceptions
    3. computations based on state involving backtracking
    4. computations based on state involving alltogether
    5. ...
3. They have in intensively used for the study of programming and specification language semantics
4. ... some of them are executable and were intensively used in purely functional languages such as Haskell.

A basic case for “imperative programming”: the *state-exception*-Monad Mon_{SE} based on the type $(o, \sigma)Mon_{SE} = \sigma \Rightarrow (o \times \sigma)option$.

1. It composes partial functions
2. In case a function evaluation fails (which can be viewed as “an exception occured”), the execution is stopped and the state remains unchanged (pretty much like Java or SML),
3. ... otherwise the execution continues with the new state.
4. $\texttt{unit}_{SE}$ corresponds to the usual “result” operation.

We define:

1. **definition** bind_SE::[(o,σ)MON_SE,o$\Rightarrow$(o,σ)MON_SE]$\Rightarrow$(o,σ
   where "bind_SE f g $\equiv\lambda\sigma$. **case** f σ **of**
   None $\Rightarrow$None
   | Some (out, σ') $\Rightarrow$g out σ'"

2. **definition** unit_SE :: "o $\Rightarrow$(o, σ)MON_SE"
   where "unit_SE e $\equiv\lambda\sigma$. Some(e,σ)"

where we use the syntax

$$x \leftarrow f; g\ x$$

for $\mathtt{bind}_{SE}\ f\ (\lambda x.g)$ and $\mathtt{return}\ e$ for $\mathtt{unit}_{SE}\ e$.

Test Sequences as Monadic Compositions

In the state exception monad, we can already represent a particular form of test-driver equivalent to a *test sequence*:

1. A **test sequence** has the form:

   $$x_1 \leftarrow put_1; x_2 \leftarrow (\lambda\_.\ put_2); \ldots; x_n \leftarrow (\lambda\_.\ put_n);$$
   $$\texttt{return}(post\ x_1\ \ldots\ x_n)$$

   i. e. the program steps under test put_i do not depend from output of prior steps.

2. A **reactive test sequence** has the form:

   $$x_1 \leftarrow put_1; x_2 \leftarrow put_2\ x_1; \ldots; x_n \leftarrow\ put_n\ x_1\ \ldots\ x_{n-1};$$
   $$\texttt{return}(post\ x_1\ \ldots\ x_n)$$

   i. e. the program steps under test put_i **may depend** from output of prior steps.

In order to make test-sequences amenable to HOL-TestGen, we need to represent them as data-types (so: lists of put_i). We introduce a $multi-bind$ combinator taking **a list of io-stepping functions** (i. e., in particular, put_l's) and executes them while taking exceptions into account:

consts mbind ::[ι list,ι $\Rightarrow$ (o,σ)MON_SE] $\Rightarrow$(o list,σ)MON_SE
primrec

"mbind [] iostep σ= Some([], σ)"
"mbind (a#H) iostep σ=
(**case** iostep a σ**of**
None $\Rightarrow$ Some([], σ)
|Some (out,σ') $\Rightarrow$(**case** mbind H iostep σ' **of**
None $\Rightarrow$ Some([out],σ')
|Some(outs,σ'')$\Rightarrow$ Some(out#outs,σ'')

Note that mbind has a slightly different behaviour than bind_SE wrt. exceptions!

On this level, we can now state **valid test sequences** as a test specification of the form:

$$\sigma_0 \models (os \leftarrow (\texttt{mbind}\ \iota s\ ioprog); \texttt{return}(post\ os))$$

where the σ_0 is the initial state and the *validity statement* $\_ \models \_$ means: start computation *ioprog* in the initial state and run it sequentially over the input sequence ιs and transfer all outputs *os* to the post condition. Sequences are *valid* iff the postcondition is true. The *validity statement* is defined as follows:

definition valid :: $\sigma \Rightarrow$(bool,σ)MON_SE $\Rightarrow$bool (infix $\models$15)
where $\sigma \models$m $\equiv$(m $\sigma \neq$None $\wedge$fst(the (m σ)))

Remark: From valid test sequence, HOL-TestGen test were generated by exploring the data-structure *input sequence* ιs up to given depths k by the standard mechanisms used in unit-tests.

However, it may be convenient to specify constraints on ιs, let it be by automata, by regular expressions, by temporal formulas or by other means. In the literature, these constraints were also called **test purposes** (TP).

$$TP(\iota s) \Longrightarrow \sigma_0 \models (os \leftarrow (\texttt{mbind}\ \iota s\ ioprog); \texttt{return}(post\ os))$$

A basic case for the "state transition system specification": the *state-relation*-Monad Mon_{SB} based on the type $(o, \sigma)Mon_{SB} = \sigma \Rightarrow (o \times \sigma)set$.

1. It composes relations on states (involving input and output)
2. In case a function evaluation fails (which can be viewed as "an exception occured"), the execution is stopped and the state remains unchanged (roughly like PROLOG),
3. ... otherwise the execution continues with the new state.
4. $\texttt{unit}_{SB}$ corresponds to the usual "result" operation.

We define:

1. **definition** bind_SB::[(α,σ)MON_SB,$\alpha\Rightarrow(\beta,\sigma)$MON_SB] $\Rightarrow(\beta,$
   where "bind_SB f g $\sigma\equiv\bigcup((\lambda(\text{out}, \sigma).\ (\text{g out } \sigma))$ ‘(f σ))"

2. **definition** unit_SB:: o $\Rightarrow$(o,σ)MON_SB
   where "unit_SB e $\equiv\lambda\sigma.\ \{(e,\sigma)\}$"

where we use the syntax

$$x \leftarrow f;;g\ x$$

for `bind`$_{SB}$ $f\ (\lambda x.g)$ and `returns` e for `unit`$_{SB}$ e.

In contrast to $\mathrm{MON\_SE}$, the operations of $\mathrm{MON\_SB}$ are not executable in general (**why?**).

On the other hand, concepts like pre- and post conditions can be easily expressed in terms of $\mathrm{MON\_SB}$.

Example: The post-condition of the operation `balance` is directly expressed in HOL as:

$$\mathtt{post}(c :: \mathit{client}, \mathit{no} :: \mathit{account\_no}) = \\ \lambda\,\sigma.\{(\mathit{result}, \sigma') \mid \sigma = \sigma' \wedge \mathit{result} = \mathit{the}(\mathit{register}(c, \mathit{no}))\}$$

Outline

Revisiting the Little Bank Example I

Example: A Little Bank - Acount System.

```
internal var register : table[client, nat]integer

op  deposit (c : client, no : account_no, amount:nat) : unit
    pre  (c,no) : dom(register)
    post register'=register[(c,no) := register(c,no) + amount]

op  balance (c : client, no : account_no) : int
    pre  (c,no) : dom(register)
    post register'=register and result = register(c,no)

op  withdraw(c : client, no : account_no, amount:nat) : unit
    pre  (c,no) : dom(register) and register(c,no) >= amount
    post register'=register[(c,no) := register(c,no) − amount]
```

In order to formalize input and output implicit in such a specification, such that we can consider it uniformely as “a list of input data” and “a list of output data”, we need to **convert** the given interface into

1. a type for the internal state,
2. a uniform data-type containing all inputs, and
3. a uniform data-type containing all outputs.

This so-called **interface encapsulation** is a syntactic transformation and could in principle be done automatically. (Not supported yet in HOL-TestGen).

Example: Interface Encapsulation

For “Little Bank”, we have:

1. a type for the internal state `register`:
   $(\text{client} \times \text{nat}) \rightharpoonup \text{int}$
2. the inputs data-type:

   datatype in_c = deposit client account_no nat
   | withdraw client account_no nat
   | balance client account_no

3. a uniform data-type containing all outputs:

   datatype out_c = depositO| balanceO nat | withdrawO

This so-called **interface encapsulation** is a syntactic transformation and could in principle be done automatically. (Not supported yet in HOL-TestGen).

Also pre-and post-conditions of “Little Bank” were encapsulated, such that we have now a typed state transition system on σ (= register), in_c and out_c.

consts precond :: "register $\Rightarrow$in_c $\Rightarrow$bool"
primrec
"precond σ(deposit c no m) = ((c,no) $\in$dom σ)"
"precond σ(balance c no) = ((c,no) $\in$dom σ)"
"precond σ(withdraw c no m) = ((c,no) $\in$dom σ
$\wedge$ (int m) $\leq$the(σ(c,no)))"

The post-condition looks as follows:

consts postcond :: "register $\Rightarrow$in_c $\Rightarrow$out_c $\times$register $\Rightarrow$bool"
primrec
"postcond σ(deposit c no m) =
(λ (n,env'). (n = depositO
$\wedge\,\sigma$'=σ ((c,no)$\mapsto$ the(env(c,no)) + int m)))"
"postcond σ(balance c no) =
(λ (n,env'). (σ=σ' $\wedge$($\exists$ x. balanceO x = n
$\wedge$ x = nat(the(σ(c,no))))))"
"postcond σ(withdraw c no m) =
(λ (n,env'). (n = withdrawO
$\wedge\,\sigma$'=σ((c,no)$\mapsto$ the(env(c,no)) $-$ int m)))"

The following combinators — based on the Hilbert-Operator — hold the key for a conversion between monads:

definition impl ::[[σ,ι]$\Rightarrow$bool,$\iota\Rightarrow$ (o,σ)MON_SB] $\Rightarrow\iota\Rightarrow$(o,$\sigma$)MON_
where "impl pre post ι=
(λ σ.if pre $\sigma\,\iota$
then Some(SOME(out,σ'). post $\iota\sigma$(out,σ'))
else arbitrary)"

definition strong_impl ::[[σ,ι]$\Rightarrow$bool, $\iota\Rightarrow$(o,σ)MON_SB]$\Rightarrow$ $\iota\Rightarrow$(o,σ
where "strong_impl pre post ι=
(λ σ. if pre $\sigma\,\iota$
then Some(SOME(out,σ'). post $\iota\sigma$(out,σ'))
else None)"

definition is_strong_impl :: "[$'\sigma \Rightarrow' \iota \Rightarrow$bool,
$'\iota \Rightarrow$ ('o,'σ)MON_SB,
$'\iota \Rightarrow$ ('o, 'σ)MON_SE] $\Rightarrow$bool"
where "is_strong_impl pre post ioprog =
($\forall\ \sigma\iota.$ ($\neg$pre $\sigma\iota \wedge$ ioprog $\iota\sigma$ = None) $\vee$
(pre $\sigma\iota\ \wedge$ ($\exists$ x. ioprog $\iota\sigma$ = Some x)))"

This results in the following:

theorem "is_strong_impl pre post (strong_impl pre post)"

This following characterization of implementable specifications gives the key for turning specs into programs. First, we define the concept of an **implementable** specification, i. e. the fact that there is a function that maps leagal input to output/state pairs, that satisfy the postcondition:

definition implementable::[$\sigma \Rightarrow \iota \Rightarrow$bool, $\iota \Rightarrow$(o,σ)MON_SB] $\Rightarrow$bool
where "implementable pre post =
$(\forall\ \sigma\iota.\ \text{pre}\ \sigma\iota\ \longrightarrow (\exists\ \text{out}\ \sigma'.\ \text{post}\ \iota\sigma(\text{out},\sigma')))$"

This results in the following characterization theorem:

theorem implementable_charn:
"$[\![$implementable pre post; pre $\sigma\iota]\!] \Longrightarrow$
post ι σ (the(strong_impl pre post $\iota\sigma$))"

It is now straight-forward to “convert” our (interface encapsulated) specification into a program. Simply:

```
strong_impl precond postcond
```

does the trick.

This program will report violations of pre- and postconditions as exceptions which were then treated at run-time.

Problem: How can we use the specification to *generate* test-sequences symbolically?

Observation: Our specification is *state-deterministic*, i. e. for each observable output, there is at most one corresponding state.

For this type of specification, we can use HOL-TestGen as follows: we state:

σ_0 $\models$s$\leftarrow$ mbind S (strong_impl precond postcond); return(s = x)

as a constraint, let HOL-TestGen find solutions for x, and use these solutions in the generated test drivers.

For this, we need the generic symbolic evaluation rules:

$(\sigma \models (s \leftarrow \text{return } x \text{ ; return } (P\ s))) = P\ x$

$(\sigma \models (s \leftarrow \text{mbind } (a\#S) \text{ ioprog ; return } (P\ s))) =$
$\quad (\textbf{case} \text{ ioprog a } \sigma\ \textbf{of}$
$\qquad \text{None} \Rightarrow (\sigma \models (\text{return } (P\ [])))$
$\quad\ \mid \text{Some}(b,\sigma') \Rightarrow (\sigma' \models (s \leftarrow \text{ mbind S ioprog ; return } (P\ (b\#s)))))$

The introduced case-statements were eliminated in the case-splitting of the test-case-generation phase.

... and the program specific symbolic evaluation rules (where $H = (\text{strong\_impl precond postcond})$):

$$
\begin{aligned}
&(\sigma \models (s \leftarrow \text{mbind } ((\text{deposit } c \text{ no } m)\#S)\ H;\ \text{return } (P\ s))) = \\
&\quad (\text{if } (c, no) \in \text{dom } \sigma \\
&\quad\ \text{then } (\sigma((c, no) \mapsto \text{the } (\sigma\ (c, no)) + \text{int } m)) \\
&\qquad\qquad\qquad \models (s \leftarrow \text{mbind } S\ H;\ \text{return } (P\ (\text{depositO}\#s))) \\
&\quad\ \text{else } (\sigma \models (\text{return } (P\ []))))
\end{aligned}
$$

$$
\begin{aligned}
&(\sigma \models (s \leftarrow \text{mbind } ((\text{balance } c \text{ no})\#S)\ H;\ \text{return } (P\ s))) = \\
&\quad (\text{if } (c, no) \in \text{dom } \sigma \\
&\quad\ \text{then } (\sigma \models (s \leftarrow \text{mbind } S\ H; \\
&\qquad\qquad\qquad \text{return } (P\ (\text{balanceO}(\text{nat}(\text{the } (\sigma\ (c, no))))\#s)))) \\
&\quad\ \text{else } (\sigma \models (\text{return } (P\ []))))
\end{aligned}
$$

...

Outline

Generating all possible input sequences is far too general: there would a lot of superfluous attempts to access a wrong account with a wrong account number, far too many initial states.
In order to reduce the number of possible input sequences, we define a *test purpose*, i. e. a predicate that constrains the number of possible input traces for one given client with an account which is initially empty.
This raises a particular *testability assumption* (at the beginning, the system is in particular initial state) which results from our lacking init method in our interface.

This test-purpose is formalized as follows:

```
consts test_purpose :: "[client, account_no, in_c list] ⇒bool"
primrec
    "test_purpose c no [] = False"
    "test_purpose c no (a#R) = (case R of
                               []  ⇒ a = balance c no
                             | a'#R' ⇒(((∃ m. a = deposit c no m) ∨
                                        (∃ m. a = withdraw c no m)) ∧
                                        test_purpose c no R))"
```

This terst-purpose formalizes that the input sequences belong to the language expressed as regular expression:

(withdraw c no _ | deposit c no _)∗ balance c no

The test-specification is formalized as follows:

test_spec test_balance:
assumes account_defined: "(c,no) $\in$dom σ_0"
and test_purpose : "test_purpose c no ιs"
and symbolic_run_yields_x :
"σ_0 $\models$(s $\leftarrow$mbind ιs (strong_impl precond postcond);
return (s = x))"
shows " σ_0 $\models$(s $\leftarrow$mbind ιs SUT; return (s = x))"

The resulting test-theorem for k=5 looks follows:

1. $(\lambda a.\ \mathrm{Some}\ 2) \models$
   $(\ s \leftarrow \mathrm{mbind}\ [\mathrm{balance}\ ?X1\ ?X2]\ \mathrm{SUT};\ \mathrm{return}\ s = [\mathrm{balanceO}\ 2]$
2. THYP ...
3. $(\lambda a.\ \mathrm{Some}\ 5) \models$
   $(\ s \leftarrow \mathrm{mbind}$
   $[\mathrm{deposit}\ ?X3\ ?X4\ ?X5,\ \mathrm{balance}\ ?X3\ ?X4]$
   $\mathrm{SUT};\ \mathrm{return}\ s = [\mathrm{depositO},\ \mathrm{balanceO}\ (\mathrm{nat}\ (5 + \mathrm{int}$
   $?X5))])$
4. THYP ...
5. THYP
6. $\mathrm{int}\ ?X6 \leq 7 \Longrightarrow$
   $(\lambda a.\ \mathrm{Some}\ 7) \models (\ s \leftarrow \mathrm{mbind}$
   $[\mathrm{withdraw}\ ?X7\ ?X8\ ?X6,\ \mathrm{balance}\ ?X7\ ?X8]\ \mathrm{SUT};$
   $\mathrm{return}\ s = [\mathrm{withdrawO},\ \mathrm{balanceO}\ (\mathrm{nat}\ (7 - \mathrm{int}\ ?X6))])$

Caution: Which are the underlying *Testability Hypothesis* (to be clear: *not* Test-Hypotheses) of this problem ???

Well, we made two (more or less explicit) testability hypothesis underlying our test-construction, that must be assured by other means than just running the test:

1. **initialization condition** (reflected by the assumption $(c,no) \in dom\ \sigma\_0$). We must assume that a concrete user and accountnumber is defined.
2. **deterministism condition** (reflected by the assumption that SUT has type $in\_c \Rightarrow (out\_c, register) Mon\_SE$). We assume that SUT behaves indeed like a function in a state in the sense of our model; we assume it is deterministic and will not have *hidden* state or engage in *hidden* state-transitions (like clocks, etc.).

Pragmatically: if we detect violations against these hypotheses during testing, we must refine our model ...

Outline

1. Test-Sequence generation can be formalized as a constraint-resolution problem, too.
2. Reason: We have data-types (this lists and laguages) and Monads in HOL
3. Test-drivers can be generated as well
4. Handling of Testability hypotheses implicit (control over the init-state, *PUT* a function in the sense of the specification)

Outline

Outline

Apparent Limitations of HOL-TestGen

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

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

- **No Non-determinism**.

Apparent Limitations of HOL-TestGen

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

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

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

Apparent Limitations of HOL-TestGen

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

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

- 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

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

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

- 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

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

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

- 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 . . .
- No possibility to describe **reactive tests**.

Apparent Limitations of HOL-TestGen

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

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

- 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.

Outline

A Deterministic Sequence Test Example I

Example: A little Bank - Acount System.

internal var register : table[client, nat]integer

op deposit (c : client, no : account_no, amount:nat) : unit
    pre (c,no) : dom(register)
    post register'=register[(c,no) := register(c,no) + amount]
op balance (c : client, no : account_no) : int
    pre (c,no) : dom(register)
    post register'=register and result = register(c,no)
op withdraw(c : client, no : account_no, amount:nat) : unit
    pre (c,no) : dom(register) and register(c,no) >= amount
    post register'=register[(c,no) := register(c,no) − amount]

A Non-Determin. Sequence Test Example II

Example: A Bank - Acount System with

```
internal var register : table[client, nat]integer
op  init(c : client, no : account_no) : unit
op  deposit (c : client, no : account_no, amount:nat) : unit
     pre  (c,no) : dom(register)
     post register'=register[(c,no) := register(c,no) + amount]
op  balance (c : client, no : account_no) : int
     pre  (c,no) : dom(register)
     post register'=register and result = register(c,no)
op  withdraw(c : client, no : account_no, amount:nat) : int
     pre  (c,no) : dom(register) and register(c,no) >= amount
     post 1<=result and result <= amount and
               register'=register[(c,no) := register(c,no) − result]
```

A Non-Determin. Sequence Test Example II

1. **Old Problem:** Only the public interface (i. e. the operations deposit, balance and withdraw. The internal (hidden) state is not accessible.
2. **Old Problem:** we can therefore only control the state by *sequences* of operation calls, not just produce data and leave it to one operation call as in unit tests.
3. **New Problem:** the operation `withdraw` may non-deterministically change the state (which can still be indirectly observed via outputs); we can therefore not pre-compute all input sequences.
4. The problem of initial states is solved by an explicit init-action creating an account for a client with an account number. (For convenience — but still realistic.)

A Non-Determin. Sequence Test Example II

1. Modified Test-Purpose :

   (init c no) (withdraw c no _ | deposit c no _)* (balance c no)

2. Modified Test-Specification:

   test_spec test_balance2:
   assumes test_purpose : "test_purpose c no ιs"
   shows _ $\models$(os $\leftarrow$mbind ιs SUT;
   return ($|\iota s| = |os| \wedge \forall i \in \{1..|os|\}.$ post' i ιs os))

3. **Note**: This works only for those parts post' of the post-conditions that do not depend on the (not observable) internal state σ.

4. **Note**: For output-deterministic specifications post' can be defined, but the construction is neither necessarily constructive nor executable (=> involves theorem proving

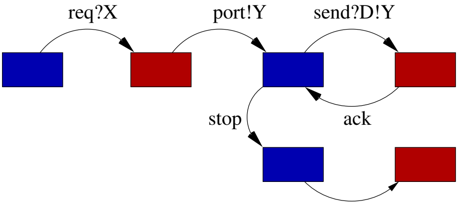
A Non-Determin. Sequence Test Example II

Note: We did not use anywhere the concrete state σ of the $\mathrm{SUT}::\iota \rightarrow (\mathrm{o},\sigma)\mathrm{MON\_SE}$, we can therefore just pass a dummy (for example, the type unit).

Outline

Motivation: A Reactive System Example I

- A toy client-server system, a simplified FTP protocol:



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

Motivation: A Reactive System Example I

- A toy client-server system, a simplified FTP protocol:

$$
\begin{aligned}
&\mathrm{req}?X \rightarrow \mathrm{port}!Y[Y < X] \rightarrow \\
&\qquad (\mathrm{rec}\, N.\ \mathrm{send}!D.Y \rightarrow \mathrm{ack} \rightarrow N \\
&\qquad\qquad \Box\ \mathrm{stop} \rightarrow \mathrm{ack} \rightarrow \mathrm{SKIP})
\end{aligned}
$$

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

Motivation: A Reactive System Example I

- A toy client-server system, a simplified FTP protocol:

$$
\begin{aligned}
&\mathrm{req}?X \to \mathrm{port}!Y[Y < X] \to \\
&\qquad (\mathrm{rec}\, N.\ \mathrm{send}!D.Y \to \mathrm{ack} \to N \\
&\qquad\qquad \Box\, \mathrm{stop} \to \mathrm{ack} \to \mathrm{SKIP})
\end{aligned}
$$

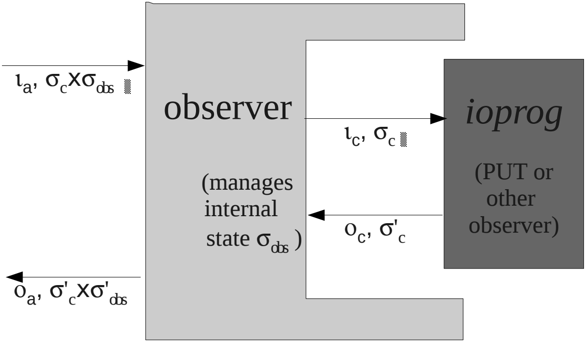
  a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel . . .

Motivation: A Reactive System Example II

Observation:

X and *Y* are only known at runtime!

- a test-driver is needed that manages a serialization of tests at test run time.
- ... including 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!


$\iota_a, \sigma_c \times \sigma_{obs}$
observer
(manages internal state σ_{obs})
ι_c, σ_c
ioprog
(PUT or other observer)
o_c, σ'_c
$o_a, \sigma'_c \times \sigma'_{obs}$

A formal definition looks as follows:

```
definition observer :: "[σ_obs ⇒o_c ⇒σ_obs,
                         σ_obs ⇒ι_a ⇒ι_c,
                         σ_obs ⇒σ⇒ ι_c ⇒ o_c ⇒bool]
                         ⇒  (ι_c ⇒ (o_c, σ)MON_SE)
                         ⇒  (ι_a ⇒ (o_c,σ_obs ×σ)MON_SE) "

where  "observer rebind substitute postcond ioprog ≡
        (λ ι_a. (λ (σ_obs, σ). let ι_c = substitute σ_obs ι_a in
        case ioprog ι_c σ of
             None ⇒None (∗ ioprog failure − eg. timeout ... ∗)
           | Some (o_c, σ’) ⇒(let σ_obs’ = rebind σ_obs o_c
                              in if  postcond σ_obs’ σ’ ι_c out_c
                                     then Some(o_c, (σ_obs’, σ’))
                                     else None (∗ postcond failure ∗)
```

As can be inferred from the type of $\mathrm{observer}$, the function is a a monad-transformer; it transforms the *i/o stepping function ioprog* into another stepping function, which is the combined sub-system consisting of the observer and, for example, a program under test *PUT*.

Thus, our concept of an *i/o stepping function* serves as an interface for varying entities in (reactive) sequence testing.

Note that we made the following testability assumptions:

1. *ioprog* behaves wrt. to the reported state and input as a function, i.e. it behaves deterministically (in the modeled state!), and
2. it is not necessary to destinguish internal failure and post-condition-failure. (Modelling Bug ? This is superfluous and blind featurism ... One could do this by introducing an own "weakening"-monad endo-transformer.)

observer can actually be decomposed into two combinators - one dealing with the management of explicit variables and one that tackles post-conditions ...

```
where  "observer3 rebind substitute ioprog ≡
       (λ ι_a. (λ (σ_obs, σ).
           let ι_c = substitute σ_obs ι_a
           in  case ioprog ι_c σ of
                    None ⇒None (∗ ioprog failure − eg. timeout .
                  | Some (o_c, σ') ⇒(let σ_obs' = rebind σ_obs o_
                                      in  Some(o_c, (σ_obs', σ')) ))
```

and ...

where "observer4 postcond ioprog ≡
(λ ι. (λ σ. **case** ioprog ισ **of**
None ⇒None *(∗ ioprog failure − eg. timeout ... ∗*
| Some (o, σ') ⇒(if postcond σ' ιo
then Some(o, σ')
else None *(∗ postcond failure*

Note that all three definitions of observers are *executable*.

We can build on top of the observer function definitions some theory on observers, which might pave the way for future optimizations. For example, the following decomposition theorem holds:

theorem observer_decompose:
"observer r s (λ x. pc) io = (observer3 r s (observer4 pc io))"

The abstraction assures that pc is a function not referring to the observer state.

Outline

FTP Protocol Example II

We specify explicit variables and a joined type containing abstract events (replacing values by explicit variables) as well as their concrete counterparts.

datatype vars = X | Y
datatype data = Data
types chan = int *(∗ just to make it executable ∗)*

Abstract and concrete events ...

datatype InEvent_conc = req chan | send data chan | stop
datatype InEvent_abs = reqA vars | sendA data vars | stopA
datatype OutEvent_conc = port chan | ack
datatype OutEvent_abs = portA vars | ackA

types InEvent = "InEvent_abs + InEvent_conc"
types OutEvent = "OutEvent_abs + OutEvent_conc"
types event_abs = "InEvent_abs + OutEvent_abs"

The function subsitute maps abstract events containing explicit variables to concrete events by substituting the variables by values communicated in the system run.
It requires an environment ("substitution") where the concrete values occuring in the system run were assigned to variables.

definition lookup :: "['a ⇀'b, 'a] ⇒ 'b"
where "lookup env v ≡the(env v)"

consts substitute :: "[vars ⇀chan, InEvent_abs] ⇒InEvent_conc
primrec
"substitute env (reqA v) = req(lookup env v)"
"substitute env (sendA d v)= send d (lookup env v)"
"substitute env stopA = InEvent_conc.stop"

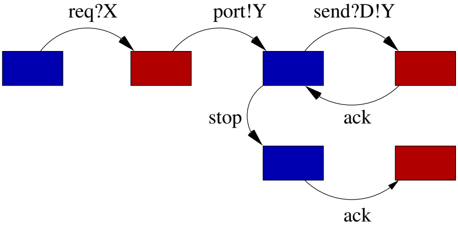
This environment is the *observer state* σ_{obs}.

The function rebind extracts from concrete output events the values and binds them to explicit variables in env. (= σ_{obs})
The predicate rebind only stores occurrences of input-events (marked by ?) in the protocol into the environment; output (!)-occurences were ignored.

consts rebind :: "[vars $\rightharpoonup$ chan, OutEvent_conc] $\Rightarrow$ vars $\rightharpoonup$ chan"
primrec
"rebind env (port n) = env(Y $\mapsto$ n)"
"rebind env OutEvent_conc.ack = env"

In a way, rebind can be viewed as an abstraction of the concrete log produced at runtime.

Revisit the protocol automaton:



Test-purpose specification (= protocol specification) is as follows (we view the enumeration type A=0 as abbreviation).

consts accept' :: "nat $\times$event_abs list $\Rightarrow$bool"
recdef accept' "measure(λ (x,y). length y)"
"accept'(A,(Inl(reqA X))#S) = accept'(B,S)"
"accept'(B,(Inr(portA Y))#S) = accept'(C,S)"
"accept'(C,(Inl(sendA d Y))#S) = accept'(D,S)"
"accept'(D,(Inr(ackA))#S) = accept'(C,S)"
"accept'(C,(Inl(stopA))#S) = accept'(E,S)"
"accept'(E,[Inr(ackA)]) = True"
"accept'(x,y) = False"

constdefs
accept :: "event_abs list $\Rightarrow$bool"

Actually, this is merely an academic exercise - we use for testing merely the subsequent protocol automaton:

We proceed by modeling a subautomaton of the protocol automaton accept.

```
consts   stim_trace' :: "nat ×InEvent_abs list ⇒bool"
recdef   stim_trace' "measure(λ (x,y). length y)"
        "stim_trace'(A,(reqA X)#S) = stim_trace'(C,S)"
        "stim_trace'(C,(sendA d Y)#S) = stim_trace'(C,S)"
        "stim_trace'(C,[stopA])      = True"
        "stim_trace'(x,y)                = False"

constdefs stim_trace :: "InEvent_abs list ⇒bool"
        "stim_trace s ≡stim_trace'(A,s)"
```

consts postcond' :: "((vars $\rightharpoonup$ int)$\times\sigma\times$InEvent_conc$\times$OutEvent_
bool"

recdef postcond' "{}"
"postcond' (env, _, req n, port m) = (m <= n)"
"postcond' (env, _, send z n, ack) = (n = lookup env Y)"
"postcond' (env, _, stop, ack) = True"
"postcond' (env, _, y, z) = False"

constdefs postcond :: "(vars $\rightharpoonup$ int) $\Rightarrow$'σ $\Rightarrow$InEvent_conc $\Rightarrow$OutEv
bool"
"postcond env σy z $\equiv$postcond' (env, σ, y, z)"

test_spec "stim_trace ιs $\Longrightarrow$
(empty[X$\mapsto$ x],())
$\models$(os$\leftarrow$(mbind ιs(observer2 rebind substitute postcond ioprog));
result(length ιs = length os))"

where ioprog is the program under test. The initial state consists of a suitably initialized observer state (the client-controlled X must be initialized), whereas we provide for the server-side state σ, which is nowhere used in the model (in particular not in postcond) and therefore polymorphic, is instantiated by the dummy type unit and its element ()).

1. $([X \mapsto ?X1], ())$
   $\models$(os $\leftarrow$mbind [reqA X,stop] (observer2 rebind substitute po
   result(2=length os)
3. $([X \mapsto ?X2], ())$
   $\models$(os $\leftarrow$mbind [reqA X,sendA Data Y,stop] (observer2 rebind
   result(3 = length os))
5. $([X \mapsto ?X3], ())$
   $\models$(os $\leftarrow$mbind [reqA X, sendA Data Y,sendA Data Y,stop] (ob
   result(4 = length os))
7. $([X \mapsto ?X4], ())$
   $\models$(os $\leftarrow$mbind [reqA X,sendA Data Y,sendA Data Y,sendA Da
   result (5 = length os))
9. ...

where we left out the test hypotheses. The meta-variables serve just as a place-holder for the initial (client-controlled) value for the X.

Outline

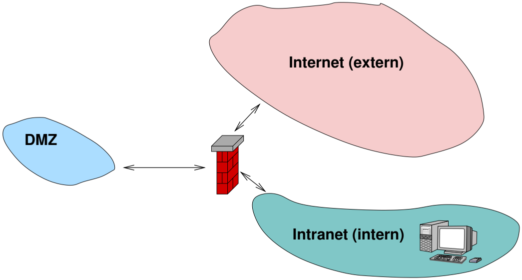
Specification-based Firewall Testing

Objective: test if a firewall configuration implements a given firewall policy

Procedure: as usual:

1. model firewalls (e.g., networks and protocols) and their policies in HOL
2. use HOL-TestGen for test-case generation

A Typical Firewall Policy



$\longrightarrow$	Intranet	DMZ	Internet
Intranet	-	smtp, imap	all protocols except smtp
DMZ	$\emptyset$	-	smtp
Internet	$\emptyset$	http,smtp	-

A Bluffers Guide to Firewalls

- A Firewall is a
  - state-less or
  - state-full

  packet filter.
- The filtering (i.e., either accept or deny a packet) is based on the
  - source
  - destination
  - protocol
  - possibly: internal protocol state

The State-less Firewall Model I

First, we model a packet:

types (α,β) packet = "id $\times$protocol $\times\alpha$src $\times\alpha$dest $\times\beta$content"

where

id: a unique packet identifier, e. g., of type Integer

protocol: the protocol, modeled using an enumeration type (e.g., ftp, http, smtp)

α src (α dest): source (destination) address, e.g., using IPv4:

types
ipv4_ip = "(int $\times$int $\times$int $\times$int)"
ipv4 = "(ipv4_ip $\times$int)"

β content: content of a packet

The State-less Firewall Model II

- A **firewall** (packet filter) either accepts or denies a packet:

  datatype
  α out = accept α| deny

- A **policy** is a map from packet to packet out:

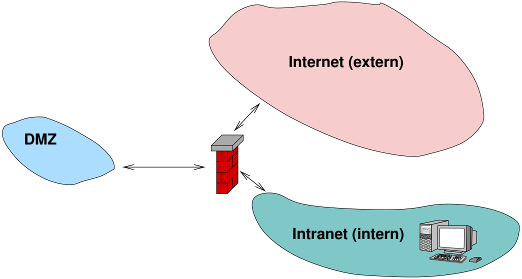
  types
  (α, β) Policy = "(α, β) packet $\rightharpoonup$((α, β) packet) out"

  where $\alpha \rightharpoonup \beta$ is a type synonym for $\alpha \rightarrow \beta$ option modeling partial functions.

- Writing policies is supported by a specialised combinator set

  constdefs
  allow_prot_from_to :: "protocol $\Rightarrow\alpha$::net set set $\Rightarrow\alpha$::net set set $\Rightarrow$(α,β)
  "allow_prot_from_to prot src_net dest_net $\equiv$allow_all |‘
  {pa. src pa $\sqsubset$src_net $\wedge$dest pa $\sqsubset$dest_net $\wedge$protocol pa = prot}"

Testing State-less Firewalls: An Example I



$\longrightarrow$	Intranet	DMZ	Internet
Intranet	-	smtp, imap	all protocols except smtp
DMZ	$\emptyset$	-	smtp
Internet	$\emptyset$	**http,smtp**	-

Testing State-less Firewalls: An Example II

src	dest	protocol	action
Internet	DMZ	http	*accept*
Internet	DMZ	smtp	*accept*
⋮	⋮	⋮	⋮
$*$	$*$	$*$	*deny*

constdefs Internet_DMZ :: "(ipv4, content) Rule"
"Internet_DMZ $\equiv$
(allow_prot_from_to smtp internet dmz) ++
(allow_prot_from_to http internet dmz)"

The policy can be modelled as follows:

constdefs test_policy :: "(ipv4,content) Policy"
"test_policy $\equiv$ deny_all ++ Internet_DMZ ++ ..."

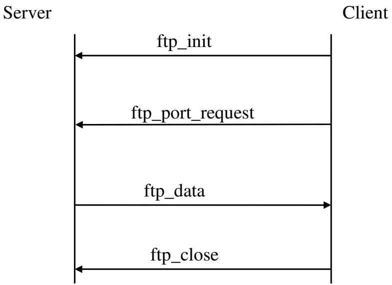
Testing State-less Firewalls: An Example III

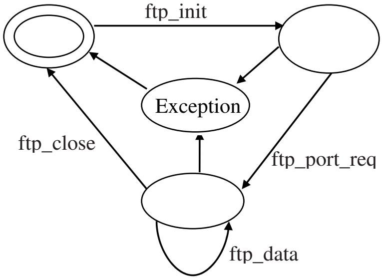
- Using the test specification

  test_spec "FUT x = test_policy x"

- results in test cases like:
  - `FUT (6,smtp,((192,169,2,8),25),((6,2,0,4),2),data) = Some (accept (6,smtp,((192,169,2,8),25),((6,2,0,4),2),data))`
  - `FUT (2,smtp,((192,168,0,6),6),((9,0,8,0),6),data) = Some deny`

State-full Firewalls: An Example (ftp) I





State-full Firewalls: An Example (ftp) II

- based on our state-less model:
  Idea: a firewall (and policy) has an internal state:
- the firewall state is based on the history and the current policy:

  types (α,β,γ) FWState = "$\alpha \times (\beta,\gamma)$ Policy"

- where FWStateTransition maps an incoming packet to a new state

  types (α,β,γ) FWStateTransition =
  "((β,γ) In_Packet $\times (\alpha,\beta,\gamma)$ FWState) $\rightharpoonup$
  ((α,β,γ) FWState)"

State-full Firewalls: An Example (ftp) III

HOL-TestGen generates test case like:

FUT [(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), close),
(6, ftp, ((4, 7, 9, 8), 21), ((192, 168, 3, 1), 3), ftp_data),
(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), port_request
(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), init)] =
([(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), close),
(6, ftp, ((4, 7, 9, 8), 21), ((192, 168, 3, 1), 3), ftp_data),
(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), port_request 3)
(6, ftp, ((192, 168, 3, 1), 10), ((4, 7, 9, 8), 21), init)],
new_policy)

Firewall Testing: Summary

- Successful testing if a concrete configuration of a network firewall correctly implements a given policy
- Non-Trivial Test-Case Generation
- Non-Trivial State-Space (IP Adresses)
- Sequence Testing used for Stateful Firewalls
- Realistic, but amazingly concise model in HOL!

Outline

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**:

$$\mathrm{pre}\, x \longrightarrow \mathrm{post}\, x (prog\; x)$$

- 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**:

$$\text{accept}\, \mathit{trace} \Longrightarrow P(\text{Mfold}\, \mathit{trace}\ \sigma_0 \mathit{prog})$$

- 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**:

$$\mathrm{accept}\ \mathit{trace} \implies P(\mathrm{Mfold}\ \mathit{trace}\ \sigma_0\ (\mathrm{observer}\ \mathrm{observer}\ \mathrm{rebind}\ \mathrm{subst}\ \mathit{prog}))$$

- 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 ...)

- **How to get the system ?**
- Current Version:
  Version HOL-TestGen 1.7 (Isabelle 2011-1)
- `http://www.brucker.ch/projects/hol-testgen`
  Including the example suite . . .

Bibliography

2012

- Achim D. Brucker and Lukas Brügger and Matthias P. Krieger and Burkhart Wolff. ***HOL-TestGen 1.7.0 User Guide***. Laboratoire en Recherche en Informatique (LRI), Université Paris-Sud 11, France, Technical Report 1551, 2012.
  Categories: ,
  (PDF) (BibTeX) (Endnote) (RIS) (Word) ( )
- Achim D. Brucker and Burkhart Wolff. **On Theorem Prover-based Testing**. In *Formal Aspects of Computing*, 2012.
  Categories: ,
  (abstract) (PDF) (BibTeX) (Endnote) (RIS) (Word) (doi:10.1007/s00165-012-0222-y) ( )
- Lukas Brügger. **A Framework for Modelling and Testing of Security Policies**. ETH Zurich,2012.
  (PDF) (BibTeX) (Endnote) (RIS) (Word) ( )

Figure: Recent publcations