Program-based Testing: A Proof-of-Technology

Burkhart Wolff

October 13, 2012

Abstract

In this paper, we present the underlying methods for white box testing in interactive unit test scenarios. HOL-TestGen can also be understood as a unifying technical and conceptual framework for presenting and investigating the variety of unit test techniques in a logically consistent way.
Keywords: symbolic test case generations, black box testing, white box testing, theorem proving, interactive testing

Contents

1 Introduction to White Box Tests

Our framework is not restricted to black box test of side-effect free programs. Using a *logical embedding* (a representation in HOL comprising syntax and semantics) for an imperative language, it can be used to implement and analyze various white-box test techniques.

MORE TO COME (COMMENTED OUT IN LATEX)

2 Appendix

3 Syntax of Commands

theory *Com* **imports** *Main* **begin**

typedecl *loc*
— an unspecified (arbitrary) type of locations (adresses/names) for variables

types
val = *nat* — or anything else, *nat* used in examples
state = *loc* ⇒ *val*
aexp = *state* ⇒ *val*
bexp = *state* ⇒ *bool*
— arithmetic and boolean expressions are not modelled explicitly here,
— they are just functions on states

datatype
com = *SKIP*
| *Assign loc aexp* (- :== - 60)
| *Semi com com* (-; - [60, 60] 10)
| *Cond bexp com com* (*IF* - *THEN* - *ELSE* - 60)
| *While bexp com* (*WHILE* - *DO* - 60)

notation (*latex*)
SKIP (`SKIP`) **and**
Cond (`IF` - `THEN` - `ELSE` - 60) **and**
While (`WHILE` - `DO` - 60)

end

4 Natural Semantics of Commands

theory *Natural* **imports** *Com* **begin**

4.1 Execution of commands

We write $\langle c,s\rangle \longrightarrow_c s'$ for *Statement c, started in state s, terminates in state s'*. Formally, $\langle c,s\rangle \longrightarrow_c s'$ is just another form of saying *the tuple* (c,s,s') *is part of the relation evalc*:

definition
update :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$ (-/[- ::= /-] [*900*,*0*,*0*] *900*) **where**
update = *fun-upd*

notation (*xsymbols*)
update (-/[- $\mapsto$ /-] [*900*,*0*,*0*] *900*)

Disable conflicting syntax from HOL Map theory.

no-syntax
-maplet :: ['*a*, '*a*] => *maplet* (- /|->/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[|->]/ -)
:: *maplet* => *maplets* (-)
-Maplets :: [*maplet*, *maplets*] => *maplets* (-,/ -)
-MapUpd :: ['*a* ~=> '*b*, *maplets*] => '*a* ~=> '*b* (-/'(-') [*900*,*0*]*900*)
-Map :: *maplets* => '*a* ~=> '*b* ((*1*[-]))

The big-step execution relation *evalc* is defined inductively:

inductive
evalc :: [*com*,*state*,*state*] $\Rightarrow$ *bool* ($\langle$-,-$\rangle$/ $\longrightarrow_c$ - [*0*,*0*,*60*] *60*)
where
Skip: $\langle$*SKIP*,*s*$\rangle \longrightarrow_c s$
| *Assign*: $\langle x ::= a,s\rangle \longrightarrow_c s[x\mapsto a\ s]$

| *Semi*: $\langle c0,s\rangle \longrightarrow_c s'' \Longrightarrow \langle c1,s''\rangle \longrightarrow_c s' \Longrightarrow \langle c0;\ c1,\ s\rangle \longrightarrow_c s'$

| *IfTrue*: $b\ s \Longrightarrow \langle c0,s\rangle \longrightarrow_c s' \Longrightarrow \langle$*IF b THEN c0 ELSE c1*, $s\rangle \longrightarrow_c s'$
| *IfFalse*: $\neg b\ s \Longrightarrow \langle c1,s\rangle \longrightarrow_c s' \Longrightarrow \langle$*IF b THEN c0 ELSE c1*, $s\rangle \longrightarrow_c s'$

| *WhileFalse*: $\neg b\ s \Longrightarrow \langle$*WHILE b DO c*,*s*$\rangle \longrightarrow_c s$
| *WhileTrue*: $b\ s \Longrightarrow \langle c,s\rangle \longrightarrow_c s'' \Longrightarrow \langle$*WHILE b DO c*, $s''\rangle \longrightarrow_c s'$
$\Longrightarrow \langle$*WHILE b DO c*, $s\rangle \longrightarrow_c s'$

lemmas *evalc.intros* [*intro*] — use those rules in automatic proofs

The induction principle induced by this definition looks like this:

$[\![\langle x1,x2\rangle \longrightarrow_c x3;\ \bigwedge s.\ P\ \texttt{SKIP}\ s\ s;\ \bigwedge x\ a\ s.\ P\ (x :== a\ )\ s\ (s[x \mapsto a\ s]);$
$\bigwedge c0\ s\ s''\ c1\ s'.$
$[\![\langle c0,s\rangle \longrightarrow_c s'';\ P\ c0\ s\ s'';\ \langle c1,s''\rangle \longrightarrow_c s';\ P\ c1\ s''\ s']\!]$
$\Longrightarrow P\ (c0;\ c1)\ s\ s';$
$\bigwedge b\ s\ c0\ s'\ c1.\ [\![b\ s;\ \langle c0,s\rangle \longrightarrow_c s';\ P\ c0\ s\ s']\!] \Longrightarrow P\ (\texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1)\ s\ s';$
$\bigwedge b\ s\ c1\ s'\ c0.\ [\![\neg\ b\ s;\ \langle c1,s\rangle \longrightarrow_c s';\ P\ c1\ s\ s']\!] \Longrightarrow P\ (\texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1)\ s\ s';$
$\bigwedge b\ s\ c.\ \neg\ b\ s \Longrightarrow P\ (\texttt{WHILE}\ b\ \texttt{DO}\ c)\ s\ s;$
$\bigwedge b\ s\ c\ s''\ s'.$
$[\![b\ s;\ \langle c,s\rangle \longrightarrow_c s'';\ P\ c\ s\ s'';\ \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s''\rangle \longrightarrow_c s';$
$P\ (\texttt{WHILE}\ b\ \texttt{DO}\ c)\ s''\ s']\!]$
$\Longrightarrow P\ (\texttt{WHILE}\ b\ \texttt{DO}\ c)\ s\ s']\!]$
$\Longrightarrow P\ x1\ x2\ x3$

($\bigwedge$ and $\Longrightarrow$ are Isabelle's meta symbols for $\forall$ and $\longrightarrow$)

The rules of *evalc* are syntax directed, i.e. for each syntactic category there is always only one rule applicable. That means we can use the rules in both directions. This property is called rule inversion.

inductive-cases *skipE* [*elim*!]: $\langle \texttt{SKIP},s\rangle \longrightarrow_c s'$
inductive-cases *semiE* [*elim*!]: $\langle c0;\ c1,\ s\rangle \longrightarrow_c s'$
inductive-cases *assignE* [*elim*!]: $\langle x :== a,s\rangle \longrightarrow_c s'$
inductive-cases *ifE* [*elim*!]: $\langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s'$
inductive-cases *whileE* [*elim*]: $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c s'$

The next proofs are all trivial by rule inversion.

inductive-simps
skip: $\langle \texttt{SKIP},s\rangle \longrightarrow_c s'$
and *assign*: $\langle x :== a,s\rangle \longrightarrow_c s'$
and *semi*: $\langle c0;\ c1,\ s\rangle \longrightarrow_c s'$

lemma *ifTrue*:
$b\ s \Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s' = \langle c0,s\rangle \longrightarrow_c s'$
⟨*proof*⟩

lemma *ifFalse*:
$\neg b\ s \Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s' = \langle c1,s\rangle \longrightarrow_c s'$
⟨*proof*⟩

lemma *whileFalse*:
$\neg\ b\ s \Longrightarrow \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c s' = (s' = s)$
⟨*proof*⟩

lemma *whileTrue*:
$b\ s \Longrightarrow$
$\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,\ s\rangle \longrightarrow_c s' =$

$(\exists\, s''.\ \langle c,s\rangle \longrightarrow_c s'' \wedge \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,\ s''\rangle \longrightarrow_c s')$
⟨*proof*⟩

Again, Isabelle may use these rules in automatic proofs:

lemmas *evalc-cases* [*simp*] = *skip assign ifTrue ifFalse whileFalse semi whileTrue*

4.2 Equivalence of statements

We call two statements c and c' equivalent wrt. the big-step semantics when *c started in s terminates in s′ iff c′ started in the same s also terminates in the same s′*. Formally:

definition
equiv-c :: *com* ⇒ *com* ⇒ *bool* (- ∼ - [*56*, *56*] *55*) **where**
$c \sim c' = (\forall\, s\ s'.\ \langle c,\ s\rangle \longrightarrow_c s' = \langle c',\ s\rangle \longrightarrow_c s')$

Proof rules telling Isabelle to unfold the definition if there is something to be proved about equivalent statements:

lemma *equivI* [*intro*!]:
$(\bigwedge s\ s'.\ \langle c,\ s\rangle \longrightarrow_c s' = \langle c',\ s\rangle \longrightarrow_c s') \Longrightarrow c \sim c'$
⟨*proof*⟩

lemma *equivD1*:
$c \sim c' \Longrightarrow \langle c,\ s\rangle \longrightarrow_c s' \Longrightarrow \langle c',\ s\rangle \longrightarrow_c s'$
⟨*proof*⟩

lemma *equivD2*:
$c \sim c' \Longrightarrow \langle c',\ s\rangle \longrightarrow_c s' \Longrightarrow \langle c,\ s\rangle \longrightarrow_c s'$
⟨*proof*⟩

As an example, we show that loop unfolding is an equivalence transformation on programs:

lemma *unfold-while*:
$(\texttt{WHILE}\ b\ \texttt{DO}\ c) \sim (\texttt{IF}\ b\ \texttt{THEN}\ c;\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{ELSE SKIP})$ (**is** *?w* ∼ *?if*)
⟨*proof*⟩

Happily, such lengthy proofs are seldom necessary. Isabelle can prove many such facts automatically.

lemma
$(\texttt{WHILE}\ b\ \texttt{DO}\ c) \sim (\texttt{IF}\ b\ \texttt{THEN}\ c;\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{ELSE SKIP})$
⟨*proof*⟩

lemma *triv-if*:
$(\texttt{IF}\ b\ \texttt{THEN}\ c\ \texttt{ELSE}\ c) \sim c$
⟨*proof*⟩

lemma *commute-if*:
(*IF b1 THEN* (*IF b2 THEN c11 ELSE c12*) *ELSE c2*)
∼
(*IF b2 THEN* (*IF b1 THEN c11 ELSE c2*) *ELSE* (*IF b1 THEN c12 ELSE c2*))
⟨*proof*⟩

lemma *while-equiv*:
⟨*c0*, *s*⟩ −→$_c$ *u* ⟹ *c* ∼ *c′* ⟹ (*c0* = *WHILE b DO c*) ⟹ ⟨*WHILE b DO c′*, *s*⟩ −→$_c$ *u*
⟨*proof*⟩

lemma *equiv-while*:
c ∼ *c′* ⟹ (*WHILE b DO c*) ∼ (*WHILE b DO c′*)
⟨*proof*⟩

Program equivalence is an equivalence relation.

lemma *equiv-refl*:
c ∼ *c*
⟨*proof*⟩

lemma *equiv-sym*:
c1 ∼ *c2* ⟹ *c2* ∼ *c1*
⟨*proof*⟩

lemma *equiv-trans*:
c1 ∼ *c2* ⟹ *c2* ∼ *c3* ⟹ *c1* ∼ *c3*
⟨*proof*⟩

Program constructions preserve equivalence.

lemma *equiv-semi*:
c1 ∼ *c1′* ⟹ *c2* ∼ *c2′* ⟹ (*c1*; *c2*) ∼ (*c1′*; *c2′*)
⟨*proof*⟩

lemma *equiv-if*:
c1 ∼ *c1′* ⟹ *c2* ∼ *c2′* ⟹ (*IF b THEN c1 ELSE c2*) ∼ (*IF b THEN c1′ ELSE c2′*)
⟨*proof*⟩

lemma *while-never*: ⟨*c*, *s*⟩ −→$_c$ *u* ⟹ *c* ≠ *WHILE* (λ*s*. *True*) *DO c1*
⟨*proof*⟩

lemma *equiv-while-True*:
(*WHILE* (λ*s*. *True*) *DO c1*) ∼ (*WHILE* (λ*s*. *True*) *DO c2*)
⟨*proof*⟩

4.3 Execution is deterministic

This proof is automatic.

theorem $\langle c,s\rangle \longrightarrow_c t \Longrightarrow \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = t$
⟨*proof*⟩

The following proof presents all the details:

theorem *com-det*:
assumes $\langle c,s\rangle \longrightarrow_c t$ **and** $\langle c,s\rangle \longrightarrow_c u$
shows $u = t$
⟨*proof*⟩

This is the proof as you might present it in a lecture. The remaining cases are simple enough to be proved automatically:

theorem
assumes $\langle c,s\rangle \longrightarrow_c t$ **and** $\langle c,s\rangle \longrightarrow_c u$
shows $u = t$
⟨*proof*⟩

end

5 Inductive Definition of Hoare Logic

theory *Hoare* **imports** *Natural* **begin**

types *assn* = *state* => *bool*

inductive
hoare :: *assn* => *com* => *assn* => *bool* (|- ({(1-)}/ (-)/ {(1-)}) 50)
where
skip: |- {*P*}`SKIP`{*P*}
| *ass*: |- {%*s*. *P*(*s*[*x*↦*a* *s*])} *x*:==*a* {*P*}
| *semi*: [| |- {*P*}*c*{*Q*}; |- {*Q*}*d*{*R*} |] ==> |- {*P*} *c*;*d* {*R*}
| *If*: [| |- {%*s*. *P* *s* & *b* *s*}*c*{*Q*}; |- {%*s*. *P* *s* & ~*b* *s*}*d*{*Q*} |] ==>
|- {*P*} `IF` *b* `THEN` *c* `ELSE` *d* {*Q*}
| *While*: |- {%*s*. *P* *s* & *b* *s*} *c* {*P*} ==>
|- {*P*} `WHILE` *b* `DO` *c* {%*s*. *P* *s* & ~*b* *s*}
| *conseq*: [| !*s*. *P'* *s* --> *P* *s*; |- {*P*}*c*{*Q*}; !*s*. *Q* *s* --> *Q'* *s* |] ==>
|- {*P'*}*c*{*Q'*}

lemma *strengthen-pre*: [| !*s*. *P'* *s* --> *P* *s*; |- {*P*}*c*{*Q*} |] ==> |- {*P'*}*c*{*Q*}
⟨*proof*⟩

lemma *weaken-post*: [| |- {*P*}*c*{*Q*}; !*s*. *Q* *s* --> *Q'* *s* |] ==> |- {*P*}*c*{*Q'*}
⟨*proof*⟩

lemma *While'*:

assumes $|-\ \{\%s.\ P\ s\ \&\ b\ s\}\ c\ \{P\}$ **and** $ALL\ s.\ P\ s\ \&\ \neg\ b\ s \longrightarrow Q\ s$
shows $|-\ \{P\}$ `WHILE` b `DO` $c\ \{Q\}$
⟨*proof*⟩

lemmas [*simp*] = *skip ass semi If*

lemmas [*intro*!] = *hoare.skip hoare.ass hoare.semi hoare.If*

end

6 Soundness and Completeness wrt Operational Semantics

theory *Hoare-Op* **imports** *Hoare* **begin**

definition
hoare-valid :: [*assn*,*com*,*assn*] => *bool* (|= {(*1*-)}/ (-)/ {(*1*-)} *50*)
where
|= $\{P\}c\{Q\}$ = (!*s t*. $\langle c,s\rangle \longrightarrow_c t$ −−> *P s* −−> *Q t*)

lemma *hoare-sound*: $|-\ \{P\}c\{Q\}$ ==> |= $\{P\}c\{Q\}$
⟨*proof*⟩

definition
wp :: *com* => *assn* => *assn* **where**
wp c Q = (%*s*. !*t*. $\langle c,s\rangle \longrightarrow_c t$ −−> *Q t*)

lemma *wp-SKIP*: *wp* `SKIP` *Q* = *Q*
⟨*proof*⟩

lemma *wp-Ass*: *wp* (*x*:==*a*) *Q* = (%*s*. *Q*($s[x\mapsto a\ s]$))
⟨*proof*⟩

lemma *wp-Semi*: *wp* (*c*;*d*) *Q* = *wp c* (*wp d Q*)
⟨*proof*⟩

lemma *wp-If*:
wp (`IF` *b* `THEN` *c* `ELSE` *d*) *Q* = (%*s*. (*b s* −−> *wp c Q s*) & (~*b s* −−> *wp d Q s*))
⟨*proof*⟩

lemma *wp-While-If*:
wp (`WHILE` *b* `DO` *c*) *Q s* =
wp (*IF b THEN c*;`WHILE` *b* `DO` *c ELSE SKIP*) *Q s*
⟨*proof*⟩

lemma *wp-While-True*: *b s ==>*
wp (`WHILE` *b* `DO` *c*) *Q s* = *wp* (*c*;`WHILE` *b* `DO` *c*) *Q s*
⟨*proof*⟩

lemma *wp-While-False*: ~*b s ==> wp* (`WHILE` *b* `DO` *c*) *Q s* = *Q s*
⟨*proof*⟩

lemmas [*simp*] = *wp-SKIP wp-Ass wp-Semi wp-If wp-While-True wp-While-False*

lemma *wp-is-pre*: |− {*wp c Q*} *c* {*Q*}
⟨*proof*⟩

lemma *hoare-relative-complete*: **assumes** |= {*P*}*c*{*Q*} **shows** |− {*P*}*c*{*Q*}
⟨*proof*⟩

end

7 Verification Conditions

theory *VC* **imports** *Hoare-Op* **begin**

datatype *acom* = *Askip*
| *Aass loc aexp*
| *Asemi acom acom*
| *Aif bexp acom acom*
| *Awhile bexp assn acom*

primrec *awp* :: *acom => assn => assn*
where
awp Askip Q = *Q*
| *awp* (*Aass x a*) *Q* = (λ*s*. *Q*(*s*[*x*↦*a s*]))
| *awp* (*Asemi c d*) *Q* = *awp c* (*awp d Q*)
| *awp* (*Aif b c d*) *Q* = (λ*s*. (*b s*−−>*awp c Q s*) & (~*b s*−−>*awp d Q s*))
| *awp* (*Awhile b I c*) *Q* = *I*

primrec *vc* :: *acom => assn => assn*
where
vc Askip Q = (λ*s*. *True*)
| *vc* (*Aass x a*) *Q* = (λ*s*. *True*)
| *vc* (*Asemi c d*) *Q* = (λ*s*. *vc c* (*awp d Q*) *s* & *vc d Q s*)
| *vc* (*Aif b c d*) *Q* = (λ*s*. *vc c Q s* & *vc d Q s*)
| *vc* (*Awhile b I c*) *Q* = (λ*s*. (*I s* & ~*b s* −−> *Q s*) &
(*I s* & *b s* −−> *awp c I s*) & *vc c I s*)

primrec *astrip* :: *acom => com*
where

```
  astrip Askip = SKIP
| astrip (Aass x a) = (x:==a)
| astrip (Asemi c d) = (astrip c;astrip d)
| astrip (Aif b c d) = (IF b THEN astrip c ELSE astrip d)
| astrip (Awhile b I c) = (WHILE b DO astrip c)


primrec vcawp :: acom => assn => assn × assn
where
  vcawp Askip Q = (λs. True, Q)
| vcawp (Aass x a) Q = (λs. True, λs. Q(s[x↦a s]))
| vcawp (Asemi c d) Q = (let (vcd,wpd) = vcawp d Q;
                            (vcc,wpc) = vcawp c wpd
                        in (λs. vcc s & vcd s, wpc))
| vcawp (Aif b c d) Q = (let (vcd,wpd) = vcawp d Q;
                            (vcc,wpc) = vcawp c Q
                        in (λs. vcc s & vcd s,
                            λs.(b s --> wpc s) & (~b s --> wpd s)))
| vcawp (Awhile b I c) Q = (let (vcc,wpc) = vcawp c I
                           in (λs. (I s & ~b s --> Q s) &
                                   (I s & b s --> wpc s) & vcc s, I))


declare hoare.conseq [intro]


lemma vc-sound: (ALL s. vc c Q s) ⟹ |- {awp c Q} astrip c {Q}
⟨proof⟩


lemma awp-mono:
  (!s. P s --> Q s) ==> awp c P s ==> awp c Q s
⟨proof⟩

lemma vc-mono:
  (!s. P s --> Q s) ==> vc c P s ==> vc c Q s
⟨proof⟩

lemma vc-complete: assumes der: |- {P}c{Q}
  shows (∃ ac. astrip ac = c & (∀ s. vc ac Q s) & (∀ s. P s --> awp
ac Q s))
  (is ? ac. ?Eq P c Q ac)
⟨proof⟩

lemma vcawp-vc-awp: vcawp c Q = (vc c Q, awp c Q)
  ⟨proof⟩

end
```

8 Denotational Semantics of Commands

theory *Denotation* **imports** *Natural* **begin**

types *com-den* = (*state*×*state*)*set*

definition
  Gamma :: [*bexp*,*com-den*] => (*com-den* => *com-den*) **where**
  Gamma b cd = (λ*phi*. {(*s*,*t*). (*s*,*t*) ∈ (*cd O phi*) ∧ *b s*} ∪
                  {(*s*,*t*). *s*=*t* ∧ ¬*b s*})

primrec *C* :: *com* => *com-den*
where
  C-skip: *C* `SKIP` = *Id*
| *C-assign*: *C* (*x* :== *a*) = {(*s*,*t*). *t* = *s*[*x*↦*a*(*s*)]}
| *C-comp*: *C* (*c0*;*c1*) = *C*(*c0*) *O* *C*(*c1*)
| *C-if*: *C* (`IF` *b* `THEN` *c1* `ELSE` *c2*) = {(*s*,*t*). (*s*,*t*) ∈ *C c1* ∧ *b s*} ∪
                                  {(*s*,*t*). (*s*,*t*) ∈ *C c2* ∧ ¬*b s*}
| *C-while*: *C*(`WHILE` *b* `DO` *c*) = *lfp* (*Gamma b* (*C c*))

lemma *Gamma-mono*: *mono* (*Gamma b c*)
  ⟨*proof*⟩

lemma *C-While-If*: *C*(`WHILE` *b* `DO` *c*) = *C*(`IF` *b* `THEN` *c*;`WHILE` *b* `DO` *c* `ELSE SKIP`)
⟨*proof*⟩

lemma *com1*: ⟨*c*,*s*⟩ ⟶$_c$ *t* ⟹ (*s*,*t*) ∈ *C*(*c*)

⟨*proof*⟩

lemma *com2*: (*s*,*t*) ∈ *C*(*c*) ⟹ ⟨*c*,*s*⟩ ⟶$_c$ *t*
⟨*proof*⟩

lemma *denotational-is-natural*: (*s*,*t*) ∈ *C*(*c*) = (⟨*c*,*s*⟩ ⟶$_c$ *t*)
  ⟨*proof*⟩

end

9 A Proof-Of-Technology of Program-based Testing: The Framework

theory
program-based-testing
imports
IMP−2011/VC
IMP−2011/Denotation
Testing

begin

9.1 Unfold and its Correctness

The core of our white box testing function is the following "unwind" function, that "unfolds" while loops and normalizes the resulting program in order to expose it to the operational semantics (i.e. the "natural semantics" *evalc* up to an unwind factor k. Evaluating programs leads to accumulating path-conditions: If a remaining constraint (whose components essentially result from applications of the *If-split* rule), is satisfiable that a path through a program is traceable and results to a certain successor state.

This can be used to test program specifications: Hoare-Triples were checked against for all paths up to a certain depth.

primrec *Append* :: [*com*,*com*] ⇒ *com* (**infixr** @@ *70*)
where
conc-skip : *SKIP* @@ *c* = *c*
| *conc-ass* : (*x*:== *E*) @@ *c* = ((*x*:== *E*); *c*)
| *conc-semi* : (*c*;*d*) @@ *e* = (*c*; *d* @@ *e*)
| *conc-If* : (*IF b THEN c ELSE d*) @@ *e* =
(*IF b THEN c* @@ *e ELSE d* @@ *e*)
| *conc-while*: (*WHILE b DO c*) @@ *e* = ((*WHILE b DO c*);*e*)

lemma *C-skip-cancel1*[*simp*] : *C*(*SKIP*;*c*) = *C*(*c*)
⟨*proof*⟩

lemma *C-skip-cancel2*[*simp*] : *C*(*c*;*SKIP*) = *C*(*c*)
⟨*proof*⟩

lemma *C-If-semi*[*simp*] :
C((*IF x THEN c ELSE d*);*e*) = *C*(*IF x THEN* (*c*;*e*) *ELSE* (*d*;*e*))

⟨*proof*⟩

lemma *comappend-correct* [*simp*]: $C(c\ @@\ d) = C(c;d)$
⟨*proof*⟩

fun *unfold* :: $nat \times com \Rightarrow com$
where
uf-skip : $unfold(n,\ SKIP) \quad = SKIP$
| *uf-ass* : $unfold(n,\ a :== E) = (a :== E)$
| *uf-If* : $unfold(n,\ IF\ b\ THEN\ c\ ELSE\ d) =$
$IF\ b\ THEN\ unfold(n,\ c)\ ELSE\ unfold(n,\ d)$
| *uf-while*: $unfold(n,\ WHILE\ b\ DO\ c) =$
$(if\ 0 < n$
$then\ IF\ b\ THEN\ unfold(n,c)@@unfold(n-\ 1, WHILE\ b\ DO\ c)\ ELSE\ SKIP$
$else\ WHILE\ b\ DO\ unfold(0,\ c))$
| *uf-semi1*: $unfold(n,\ SKIP\ ;\ c) = unfold(n,\ c)$
| *uf-semi2*: $unfold(n,\ c\ ;\ SKIP) = unfold(n,\ c)$
| *uf-semi3*: $unfold(n,\ (IF\ b\ THEN\ c\ ELSE\ d)\ ;\ e) =$
$(IF\ b\ THEN\ (unfold(n,c;e))\ ELSE\ (unfold(n,d;e)))$
| *uf-semi4*: $unfold(n,\ (c\ ;\ d);\ e) = \ (unfold(n,\ c;d))@@(unfold(n,e))$
| *uf-semi5*: $unfold(n,\ c\ ;\ d) = \ (unfold(n,\ c))@@(unfold(n,\ d))$

lemma *unfold-correct-aux1* :
assumes $H : \forall x.\ C\ (unfold\ (x,\ c)) \quad = C\ c$
shows $C(unfold(n, WHILE\ b\ DO\ c)) = C(WHILE\ b\ DO\ c)$
⟨*proof*⟩

declare *uf-while* [*simp del*]

lemma *unfold-correct-aux2* :
$C(unfold(n,c;d)) = C(unfold(n,c)\ ;\ unfold(n,\ d))$
⟨*proof*⟩ **print-cases**
⟨*proof*⟩

lemma *unfold-correct* [*rule-format*]: $\forall\ x.\ (C(unfold(x,c)) = C(c))$
⟨*proof*⟩

lemma *wp-unfold* : $wp\ (c)\ (p) = wp(unfold(n,c))\ (p)$
⟨*proof*⟩

lemma *wp-test* : $\forall \sigma.\ P\ \sigma \longrightarrow wp\ (unfold(k,c))\ Q\ \sigma \implies |\!-\ \{P\}\ c\ \{Q\}$
⟨*proof*⟩

9.2 Symbolic Evaluation Rule-Set

lemma *If-split*:
$[\![\ b\ s \Longrightarrow \langle c0,s\rangle \longrightarrow_c s';\ \neg\ b\ s \Longrightarrow \langle c1,s\rangle \longrightarrow_c s'\ ]\!] \Longrightarrow \langle IF\ b\ THEN\ c0\ ELSE\ c1,s\rangle \longrightarrow_c s'$
⟨*proof*⟩

lemma *If-splitE*:
$[\![\ \langle IF\ b\ THEN\ c\ ELSE\ d,s\rangle \longrightarrow_c s';\ [\![\ b\ s;\ \langle c,s\rangle \longrightarrow_c s'\ ]\!] \Longrightarrow P;\ [\![\ \neg\ b\ s;\ \langle d,s\rangle \longrightarrow_c s'\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$

⟨*proof*⟩

9.3 Splitting Rule for program-based Tests

lemma *symbolic-eval-test* :
$(\ |\!-\ \{Pre\}\ c\ \{Post\}) = (\forall s\ t.\ \langle unfold\ (n,\ c),s\rangle \longrightarrow_c t \longrightarrow Pre\ s \longrightarrow Post\ t)$
⟨*proof*⟩

9.4 Tactic Set-up

⟨*ML*⟩

lemmas *one-point-rules* = *HOL.simp-thms*(*39*) *HOL.simp-thms*(*40*)

lemma *IF-split*:
$\langle IF\ b\ THEN\ c\ ELSE\ d,s\rangle \longrightarrow_c s' = ((b\ s \wedge \langle c\ ,s\rangle \longrightarrow_c s')\ \vee\ (\neg\ b\ s \wedge \langle d\ ,s\rangle \longrightarrow_c s'))$
⟨*proof*⟩

lemma *assign-sequence*:
$\langle a{:=}{=}\ e;\ c,s\rangle \longrightarrow_c s' = \langle c,s[a \mapsto e\ s]\rangle \longrightarrow_c s'$
⟨*proof*⟩

lemmas *symbolic-evaluation* = *IF-split*
Natural.skip Natural.assign
Natural.semi Natural.whileFalse

thm *symbolic-evaluation*

lemmas *symbolic-evaluation2 = IF-split assign-sequence*
Natural.skip Natural.assign
Natural.whileFalse

lemmas *memory-model = Fun.fun-upd-other HOL.simp-thms(8)*
Fun.fun-upd-same Fun.fun-upd-triv

⟨*ML*⟩

end

10 Program-based Testing: The Squareroot-Example.

theory
squareroot-test
imports
../../src/program-based-testing
begin

10.1 The Definition of the Integer-Squareroot Program

definition *squareroot* :: [*loc*,*loc*,*loc*,*loc*] ⇒ *com*
where *squareroot tm sqsum i a* ==
((*tm* :== (λ*s*. *1*));
((*sqsum* :== (λ*s*. *1*));
((*i* :== (λ*s*. *0*));
WHILE (λ*s*. (*s sqsum*) <= (*s a*)) *DO*
((*i* :== (λ*s*. (*s i*) + *1*));
((*tm* :== (λ*s*. (*s tm*) + *2*));
(*sqsum* :== (λ*s*. (*s tm*) + (*s sqsum*)))))))
)

definition *pre* :: *assn* **where** *pre* ≡ λ *x*. *True*
definition *post* :: [*loc*,*loc*] ⇒ *assn*
where *post a i* ≡ λ *s*. (*s i*)∗(*s i*)≤(*s a*) ∧ *s a* < (*s i* + *1*)∗(*s i* + *1*)

definition *inv* :: [*loc*,*loc*,*loc*,*loc*] ⇒ *assn*
where *inv i sqsum tm a* ≡ λ*s*.(*s i* + *1*) ∗ (*s i* + *1*) = *s sqsum*
∧ *s tm* = (*2* ∗ (*s i*) + *1*)
∧ (*s i*) ∗ (*s i*) <= (*s a*)

10.2 Computing Program Paths and their Path-Constraints

lemma *derive-pathconds*:
assumes *no-alias* : *sqsum* ≠ *i* ∧ *i* ≠ *sqsum* ∧ *tm* ≠ *sqsum* ∧
sqsum ≠ *tm* ∧ *sqsum* ≠ *a* ∧ *a* ≠ *sqsum* ∧
tm ≠ *i* ∧ *i* ≠ *tm* ∧ *tm* ≠ *a* ∧ *a* ≠ *tm* ∧
a ≠ *i* ∧ *i* ≠ *a*
shows ⟨*unfold*(*3*, *squareroot tm sqsum i a*), *s*⟩ ⟶$_c$ *s′*

⟨*proof*⟩

Summary: With this approach, one can synthesize paths and their conditions.

10.3 Testing Specifications

thm *symbolic-evaluation2*

Slow Motion Interactive Version (for demonstrations).

lemma *whitebox-test*:
assumes *no-alias*[*simp*] : *sqsum* ≠ *i* ∧ *i* ≠ *sqsum* ∧ *tm* ≠ *sqsum* ∧

sqsum ≠ *tm* ∧ *sqsum* ≠ *a* ∧ *a* ≠ *sqsum* ∧
tm ≠ *i* ∧ *i* ≠ *tm* ∧ *tm* ≠ *a* ∧ *a* ≠ *tm* ∧
a ≠ *i* ∧ *i* ≠ *a*
shows |− {*pre*} *squareroot tm sqsum i a* {*post a i*}

⟨*proof*⟩

Automated Version:

lemma *whitebox-test2*:
assumes *no-alias*[*simp*] : *sqsum* ≠ *i* ∧ *i* ≠ *sqsum* ∧ *tm* ≠ *sqsum* ∧

sqsum ≠ *tm* ∧ *sqsum* ≠ *a* ∧ *a* ≠ *sqsum* ∧
tm ≠ *i* ∧ *i* ≠ *tm* ∧ *tm* ≠ *a* ∧ *a* ≠ *tm* ∧
a ≠ *i* ∧ *i* ≠ *a*
shows |− {*pre*} *squareroot tm sqsum i a* {*post a i*}

⟨*proof*⟩

10.4 An Alternative Approach with an On-The-Fly generated Explicit Test-Hyp.

Recall the rules for the computation of weakest preconditions:

```
Hoare.wp_def: wp ?c ?Q == %s. ALL t. (s, t) : C ?c --> ?Q t

Hoare.wp_If:   wp (IF ?b THEN ?c ELSE ?d) ?Q = (%s. (?b s --> wp ?c ?Q s) &
                                                  (~ ?b s --> wp ?d ?Q s))
Hoare.wp_Semi: wp (?c; ?d) ?Q = wp ?c (wp ?d ?Q)
Hoare.wp_Ass:  wp (?x :== ?a ) ?Q = (%s. ?Q (s[?x ::= ?a s]))
Hoare.wp_SKIP: wp SKIP ?Q = ?Q
```

lemma *path-exploration-test*:
assumes *no-alias* : $sqsum \neq i \wedge i \neq sqsum \wedge tm \neq sqsum \ \wedge$
$sqsum \neq tm \wedge sqsum \neq a \wedge a \neq sqsum \wedge$
$tm \neq i \wedge i \neq tm \wedge tm \neq a \wedge a \neq tm \ \wedge$
$a \neq i \wedge i \neq a$
shows $|-$ $\{pre\}$ *squareroot tm sqsum i a* $\{post\ a\ i\}\langle proof\rangle$

end