Program-based Testing

A Proof-of-Technology

Achim D. Brucker and Burkhart Wolff

October 14, 2012

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
Further readings: This theory is an extended and updated version of the work described in the paper "Interactive testing using HOL-TestGen.", LNCS 2997, Springer Verlag, 2005 [4].

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.

1.1 The Language IMP: An Overview

The Isabelle distribution comes already with various logical embeddings: IMP, IMPP, NanoJava, or MicroJava, and more are available in the literature. For the sake of this presentation, we chose the simplest one, IMP, which is intended as formalization of a textbook on programming language semantics [10, 8], and provides as such a particularly clean and complete collection of several semantics of IMP (natural semantics, transition semantics, denotational semantics, axiomatic semantics), proofs of their relations (e.g., denotational is equivalent to natural) and proofs of crucial meta-properties (axiomatic semantics is sound and relative complete).

The basic concepts of IMP are *values* val (just natural numbers, for example), and *states* state = loc ⇒ val. *Boolean expressions* bexp and *atomic expressions* (aexp) are represented as functions from state to val or bool. Thus, IMP has in fact no syntax of its own, but just inherits the expression language of HOL at this place[1]. The syntax of IMP commands com is then defined as data type:

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)

where the text in the parenthesis are just pragmas for the powerful Isabelle syntax engine to allow the usual infix/mixfix notation.

One of the operational semantics of IMP is a relation of triples evalc :: (com ×state × state) set ((cm,s,s') ∈ evalc is denoted ⟨cm,s⟩ $\xrightarrow[c]{}$ s') which is inductively defined as follows:

inductive evalc **intros**
"⟨SKIP,s⟩ $\xrightarrow[c]{}$ s"
"⟨x :== a,s⟩ $\xrightarrow[c]{}$ s[x:=(a s)]"
"⟦ ⟨c_0,s⟩ $\xrightarrow[c]{}$ s_1; ⟨cs_1,s_1⟩ $\xrightarrow[c]{}$ s_2 ⟧ ⟹ ⟨c_0;cs_1, s⟩ $\xrightarrow[c]{}$ s_2"
"⟦ b s; ⟨c_0,s⟩ $\xrightarrow[c]{}$ s_1 ⟧ ⟹ ⟨ IF b THEN c_0 ELSE c_1, s⟩ $\xrightarrow[c]{}$ s_1"
"⟦ ¬b s; ⟨c_1,s⟩ $\xrightarrow[c]{}$ s_1 ⟧ ⟹ ⟨ IF b THEN c_0 ELSE c_1, s⟩ $\xrightarrow[c]{}$ s_1"
"⟦¬b s⟧ ⟹ ⟨ WHILE b DO c, s⟩ $\xrightarrow[c]{}$ s"
"⟦ b s; ⟨c,s⟩$\xrightarrow[c]{}$ s_1; ⟨ WHILE b DO c,s_1⟩$\xrightarrow[c]{}$ s_2 ⟧ ⟹ ⟨ WHILE b DO c, s⟩ $\xrightarrow[c]{}$ s_2"

[1] This technique is also called a "shallow embedding"

The usual notation s[x:=v] is defined by λ y. if y=x then v else s y. For these inductive rules, an alternative rule set is derived that can be processed by the efficient Isabelle rewriter directly:

⟨ SKIP,s⟩ $\underset{c}{\rightarrow}$ s' = (s' = s)
⟨ x :== a, s⟩ $\underset{c}{\rightarrow}$ s' = (s' = s[x := a s])
b s ⟹ ⟨ IF b THEN d ELSE e,s⟩ $\underset{c}{\rightarrow}$ s' = ⟨ d,s⟩ $\underset{c}{\rightarrow}$ s'
. . .

We omit the definition of the denotational semantics reflecting the partial correctness C :: com ⇒(state × state) set (see [2] for details), but it is linked to the operational semantics via the theorem ((s, t) ∈ C c) = ⟨c,s⟩ $\underset{c}{\rightarrow}$ t. On the denotational level, program transformation rules relevant for the next section can be shown easily:

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)
C(WHILE b DO c) = C(IF b THEN c; WHILE b DO c ELSE SKIP)

On the level of the denotational semantics, the usual notion of "valid Hoare triple" is formalized as:

|= {P} c {Q} ≡ ∀s t. (s, t) ∈ C c ⟶ P s ⟶ Q t

where P, Q are *assertions*, i.e., functions from state to bool.

1.2 Unwinding IMP Programs

To perform white box tests in the style of Pathfinder [9], SpecExplorer [7], or Korat [3], it is necessary to make the program paths explicit in the program representation and amenable to the rules of the operational semantics. Therefore, a pre-processing step is necessary that unfolds all WHILE-loops up to a certain limit, the *unwind-factor* k. This principle can also be applied in a language extension with procedure calls such as IMPP, also available in the Isabelle distribution. Additionally, the program should be transformed into a certain normal form to be efficiently processed (left associative sequential compositions must be avoided since they lead to an existentially quantified intermediate states which are more difficult to process in the symbolic computation). We define two recursive functions on com-terms that perform both these normalizations as well as the unwinding up to k. Note, that we will not program this function outside the logic as (tactic), i.e., a control program in SML, but inside HOL, such that we can also prove its correctness with respect to the IMP semantics:

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 THEN c @@ e ELSE d @@ e)"
"(WHILE b DO c) @@ e = ((WHILE b DO c);e)"

consts unwind :: "nat × com ⇒com"
recdef unwind "less_than <*lex*> measure(λ s. size s)"

```
"unwind(n, SKIP)      = SKIP"
"unwind(n, a :== E) = (a :== E)"
"unwind(n,  IF  b THEN c ELSE d) =  IF  b THEN unwind(n,c) ELSE unwind(n,d)"
"unwind(n,  WHILE  b DO  c) =
        (if  0 < n
         then  IF  b THEN unwind(n,c)@@unwind(n− 1,WHILE b DO c) ELSE SKIP
         else  WHILE  b DO  unwind(0, c))"
"unwind(n, SKIP  ;  c) = unwind(n, c)"
"unwind(n, c  ;  SKIP) = unwind(n, c)"
"unwind(n, ( IF  b THEN c ELSE d)  ;  e) =
             ( IF  b THEN  (unwind(n,c;e))  ELSE  (unwind(n,d;e)))"
"unwind(n, (c  ;  d);  e) =  (unwind(n, c;d))@@(unwind(n,e))"
"unwind(n, c  ;  d) =  (unwind(n, c))@@(unwind(n, d))"
```

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. The more tricky unwind function unfolds `WHILE`-loops as long as the unwind factor is positive and performs the program normal form computation along the program equivalences as discussed in Sec. 1.1.

The Isabelle Recursion Package adopts a "First Fit" pattern matching strategy (similar to SML). This means that in overlapping cases, the first is taken into account with higher priority—this is reflected on the level of the rewrite rule set generated from this definition. Thus, the last equation in the recursive definition is a catch-all rule for sequential composition.

Now we derived the following facts over these definitions:

Lemma 1 (Termination:)**.** *Both functions terminate.*

Proof. In the case of @@ this is trivial due to machine checked primitive recursion; in case of unwind a proof has to be performed that the lexicographic composition of the standard ordering $\_ < \_$ and the standard term ordering is well-founded and respected by the inner calls in this recursive definition. This proof is done fully automatically. □

Lemma 2 (Correctness:)**.** *C(c @@ d)= C(c;d) and C(unwind(n,c))= C(c)*

Proof. For @@, a straight-forward induction suffices. As for unwind, the proof is non-trivial, but routine (generalization over n, induction over c, intricate case splitting, application of semantic equivalences of Sec. 1.1). □

1.3 Generating Path Conditions

As example program, we chose a little program that computes the square-root of a natural number. In Isabelle/IMP syntax, we can define it as follows:

```
constdefs squareroot  ::  "[loc, loc, loc, loc]  ⇒ com"
        "squareroot  tm sum i a ≡ ((   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)))))))
```

where the locations (references) are the input into the program to express semantically constraints on them, as we will see later. The shallow embedding of the expressions has the consequence that program variable accesses must be represented as explicit application of the state s (at this program point) to a location representing this variable. Hence, we implicitly require a pre-parser that makes these bindings of program variables explicit.

We need one further derived rule If_split , which is necessary to expand the case splits produced for each path:

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

Putting everything together, we can now formulate the generation of symbolic states for the program squareroot as follows:

lemma derive_test_cases: **assumes** no_alias : . . .
shows "⟨unwind(3, squareroot tm sum i a), s⟩ $\xrightarrow[c]{}$ s'"

where the omitted technical side-condition no_alias specifies that the locations tm,sum,i,a are pairwise disjoint. Now, the canonical tactic script:

apply(simp add: squareroot_def)
apply(*rule* If_split , simp_all add: update_def no_alias)+

unfolds the definition of squareroot, and then enters in a loop that performs the computation of unwind (including path normalization), the case splitting along the If_split rule discussed above, the evaluation of state constraints and the simplification of the arithmetic constraints until no further changes can be achieved. The resulting proof-state consists of the following goals:[2]

```
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)⟩ →c s'
2. [[4 ≤s a; 8 < s a ]]     ⟹ s' = s (i := 2, tm := 5, sum := 9)
3. [[ 1 ≤s a; s a < 4]]     ⟹ s' = s (i := 1, tm := 3, sum := 4)
4.  s a = 0                 ⟹ s' = s(tm := 1, sum := 1, i := 0)
```

The resulting proof state enumerates the possible symbolic states including their path conditions.[3]

1.4 Treating Assertions and Test Hypothesises

Traditional pre and post conditions can be expressed via the validity relation for Hoare Triple, e.g.: |= {pre} squareroot tm sum i a {post a i} where pre is just λx. True and post a i is λ s. (s i)∗(s i)≤(s a) ∧ s a < (s i + 1)∗(s i + 1).

[2] the presentation has been slightly syntactically simplified

[3] The computing time for unwind-factor 10 based on this simplistic tactic remains under a few seconds, including pretty-printing.

The setup of a specification based white box test is now produced by the derived rule:

$\models\{P\}\ c\ \{Q\} = \forall s\ t.\quad \langle \mathsf{unwind}\ (n,\ c), s\rangle \underset{c}{\rightarrow} t \longrightarrow P\ s \longrightarrow Q\ t$

The result of this rule application is piped into the previous process which conjoins the preconditions with the path conditions and attempts to solve them; the post condition is then constructed over the post state constructed by the natural semantics.

Assertions can be introduced into our language as follows: First, we declare an uninterpreted constant `STOP` as command of the language. Then, a construct like `ASSERT` b c can be introduced as abbreviation for `ASSERT` b c $\equiv$ `IF` b `THEN` c `ELSE STOP`, and further constructs like an annotated while loop `AWHILE` b inv c are introduced analogously.

It remains to show how white box testing fits *methodically* into our framework, where we try to generate *test hypothesis* that make the "logical difference" between a test and the verification of the test specification explicit. Obviously, the only new element related to white box test is the unwinding parameter; if exhausted, this leads to program fragments that represent the "set of untested execution paths" of a program under test. In our running example, this lead to the first sub-goal in the final proof state. Turned into an explicit *unwinding k test hypothesis*, this condition for resulting from the test theorem: $\models${pre} squareroot tm sum i a {post a i} looks as follows:

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

Testing a program in this setting means that all symbolic state transitions including their path conditions must satisfy the post condition whenever the pre condition holds. This is the case in our example, and the system will find the satisfiability of the generated constraints without need for random solving in this case. The only remaining assumption is the test hypothesis shown above which reflects that we have tested the program and not verified it.

To sum up, we described a symbolic computation process for white box tests in the language IMP, that generates from a given, potentially annotated program a test theorem including the test hypotheses automatically. This test theorem can be fed into the test data generation phase to find ground instances for particular paths as before.

1.5 Blowing up IMP

The reader might object that the language IMP, having only Boolean and arithmetic side-effect free expressions and non-recursive, macro-like procedures, is too academic to be of practical importance. In contrast, we argue that IMP is a reasonable core language which can be "blown-up" fairly easy to larger languages, in large parts without adding further complexity to the symbolic computation process presented so far.

We discuss three extensions of IMP, two more straight-forward, one more involved, to give an impression over the potential of our approach:

1. *Mutual recursion:* Just apply our approach to embedding IMPP.
2. *Arbitrary expressions:* exchange val in the IMP semantics by a universe which is a sum of the HOL data types.
3. *Objects:* Extend our approach to an embedding like NanoJava.

In more detail, extension 2 requires that program variables must be presented as triples $(\mathrm{loc}, \mathrm{emb}::\alpha \longrightarrow \mathrm{val}, \mathrm{proj}::\mathrm{val} \longrightarrow \alpha)$ consisting of the traditional location, and a pair of functions (representing the typing of the program and variable) that allow for injection and projection of HOL-values into the val universe of IMP. Program variable accesses, which has been encoded by s a so far, will be s!a where s!(a,emb,prj) is defined by prj(s a). The assignment semantics of IMP must be adopted analogously. This technique paves the way for lists, options, strings, and further user-defined data types. Expressions over user-defined HOL data-types can now be processed by the gen_test_cases-method which is at the heart of HOL-TestGen. As a result of these extensions, we have an SML-like language with data-types and HOL-expressions inside.

The extension 3 involves sub-classing, method calls with late-binding and object creation; as such, a lot more machinery is therefore involved whose tactical control will be feasible in our opinion, but require substantial more work.

2 Conclusion

We have shown the pragmatics of our Isabelle/HOL-based testing tool HOL-TestGen [1, 6] gained from previous experiences for specification based black box tests. While some aspects of the symbolic computations are fully automatic (like data separation lemma generation, generation of test hypothesis, TNF-computations, test data generations and solving), other aspects like constraint solving may profit from some theorem proving and experiments with "appropriate" formulations of test specifications/test theorems. We have also developed a method to use HOL-TestGen for specification based white box tests.

The symbolic computation process is fully presented inside HOL, so no tool integration and conversion issues are involved which may be critical both for correctness and efficiency. Since the necessary symbolic transformation processes can be based on derived rules,[4] HOL-TestGen can be used as a tool for a seamless *conceptual study* of these techniques including formal correctness proofs, their *prototypical implementation* and even their *industry strength implementation*. The latter, will require substantial effort in tactic programming and tool integration.

Although the example for imperative white-box test is based on a conceptual language and therefore merely a proof of concept than a proof of technology, we believe that the approach can scale up with respect to size of the supported language while maintaining reasonable efficiency of the underlying symbolic computations. Thus, we believe that HOL-TestGen can be seen as unifying framework in which a wide range of unit test techniques can be presented in a mathematically clean way.

[4] Inserting such rules as *axioms* is trivial, but endangers correctness, of course

3 Appendix

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

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

3.1 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* ⇒ '*b*) ⇒ '*a* ⇒ '*b* ⇒ ('*a* ⇒ '*b*) (-/[- ::= /-] [900,0,0] 900) **where**
update = *fun-upd*

notation (*xsymbols*)
update (-/[- ↦ /-] [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*] ⇒ *bool* (⟨-,-⟩/ $\longrightarrow_c$ - [*0*,*0*,*60*] *60*)
where
Skip: ⟨*SKIP*,*s*⟩ $\longrightarrow_c$ *s*
| *Assign*: ⟨*x* :== *a*,*s*⟩ $\longrightarrow_c$ *s*[*x*↦*a* *s*]

| *Semi*: ⟨*c0*,*s*⟩ $\longrightarrow_c$ *s''* ⟹ ⟨*c1*,*s''*⟩ $\longrightarrow_c$ *s'* ⟹ ⟨*c0*; *c1*, *s*⟩ $\longrightarrow_c$ *s'*

| *IfTrue*: *b s* ⟹ ⟨*c0*,*s*⟩ $\longrightarrow_c$ *s'* ⟹ ⟨*IF b THEN c0 ELSE c1*, *s*⟩ $\longrightarrow_c$ *s'*
| *IfFalse*: ¬*b s* ⟹ ⟨*c1*,*s*⟩ $\longrightarrow_c$ *s'* ⟹ ⟨*IF b THEN c0 ELSE c1*, *s*⟩ $\longrightarrow_c$ *s'*

| *WhileFalse*: ¬*b s* ⟹ ⟨*WHILE b DO c*,*s*⟩ $\longrightarrow_c$ *s*
| *WhileTrue*: *b s* ⟹ ⟨*c*,*s*⟩ $\longrightarrow_c$ *s''* ⟹ ⟨*WHILE b DO c*, *s''*⟩ $\longrightarrow_c$ *s'*
⟹ ⟨*WHILE b DO c*, *s*⟩ $\longrightarrow_c$ *s'*

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

The induction principle induced by this definition looks like this:

⟦⟨*x1*,*x2*⟩ $\longrightarrow_c$ *x3*; ⋀*s*. *P SKIP s s*; ⋀*x a s*. *P* (*x* :== *a*) *s* (*s*[*x* ↦ *a s*]);
⋀*c0 s s'' c1 s'*.
⟦⟨*c0*,*s*⟩ $\longrightarrow_c$ *s''*; *P c0 s s''*; ⟨*c1*,*s''*⟩ $\longrightarrow_c$ *s'*; *P c1 s'' s'*⟧
⟹ *P* (*c0*; *c1*) *s s'*;
⋀*b s c0 s' c1*. ⟦*b s*; ⟨*c0*,*s*⟩ $\longrightarrow_c$ *s'*; *P c0 s s'*⟧ ⟹ *P* (*IF b THEN c0 ELSE c1*) *s s'*;
⋀*b s c1 s' c0*. ⟦¬ *b s*; ⟨*c1*,*s*⟩ $\longrightarrow_c$ *s'*; *P c1 s s'*⟧ ⟹ *P* (*IF b THEN c0 ELSE c1*) *s s'*;
⋀*b s c*. ¬ *b s* ⟹ *P* (*WHILE b DO c*) *s s*;
⋀*b s c s'' s'*.
⟦*b s*; ⟨*c*,*s*⟩ $\longrightarrow_c$ *s''*; *P c s s''*; ⟨*WHILE b DO c*,*s''*⟩ $\longrightarrow_c$ *s'*;
P (*WHILE b DO c*) *s'' s'*⟧
⟹ *P* (*WHILE b DO c*) *s s'*⟧
⟹ *P x1 x2 x3*

(⋀ and ⟹ are Isabelle's meta symbols for ∀ 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*!]: ⟨*SKIP*,*s*⟩ $\longrightarrow_c$ *s'*
inductive-cases *semiE* [*elim*!]: ⟨*c0*; *c1*, *s*⟩ $\longrightarrow_c$ *s'*
inductive-cases *assignE* [*elim*!]: ⟨*x* :== *a*,*s*⟩ $\longrightarrow_c$ *s'*
inductive-cases *ifE* [*elim*!]: ⟨*IF b THEN c0 ELSE c1*, *s*⟩ $\longrightarrow_c$ *s'*
inductive-cases *whileE* [*elim*]: ⟨*WHILE b DO c*,*s*⟩ $\longrightarrow_c$ *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*:
$b\ s \Longrightarrow \langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,\ s\rangle \longrightarrow_c s' = \langle c0,s\rangle \longrightarrow_c s'$
by *auto*

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

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

lemma *whileTrue*:
$b\ s \Longrightarrow$
$\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,\ s\rangle \longrightarrow_c s' =$
$(\exists\, s''.\ \langle c,s\rangle \longrightarrow_c s'' \wedge \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,\ s''\rangle \longrightarrow_c s')$
by *auto*

Again, Isabelle may use these rules in automatic proofs:

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

3.2 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*!]:
$(\bigwedge s\ s'.\ \langle c,\ s\rangle \longrightarrow_c s' = \langle c',\ s\rangle \longrightarrow_c s') \Longrightarrow c \sim c'$
by (*unfold equiv-c-def*) *blast*

lemma *equivD1*:
$c \sim c' \Longrightarrow \langle c,\ s\rangle \longrightarrow_c s' \Longrightarrow \langle c',\ s\rangle \longrightarrow_c s'$
by (*unfold equiv-c-def*) *blast*

lemma *equivD2*:
$c \sim c' \Longrightarrow \langle c',\ s\rangle \longrightarrow_c s' \Longrightarrow \langle c,\ s\rangle \longrightarrow_c s'$
by (*unfold equiv-c-def*) *blast*

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

lemma *unfold-while*:
(`WHILE` b `DO` c) $\sim$ (`IF` b `THEN` c; `WHILE` b `DO` c `ELSE SKIP`) (**is** *?w* $\sim$ *?if*)
proof −
— to show the equivalence, we look at the derivation tree for
— each side and from that construct a derivation tree for the other side
{ fix s s' **assume** w: $\langle ?w, s\rangle \longrightarrow_c s'$
— as a first thing we note that, if b is *False* in state s,
— then both statements do nothing:
hence $\neg b\ s \Longrightarrow s = s'$ **by** *blast*
hence $\neg b\ s \Longrightarrow \langle ?if, s\rangle \longrightarrow_c s'$ **by** *blast*
moreover
— on the other hand, if b is *True* in state s,
— then only the *WhileTrue* rule can have been used to derive $\langle ?w, s\rangle \longrightarrow_c s'$
{ assume b: $b\ s$
with w **obtain** s'' **where**
$\langle c, s\rangle \longrightarrow_c s''$ **and** $\langle ?w, s''\rangle \longrightarrow_c s'$ **by** (*cases set*: *evalc*) *auto*
— now we can build a derivation tree for the `IF`
— first, the body of the True-branch:
hence $\langle c; ?w, s\rangle \longrightarrow_c s'$ **by** (*rule Semi*)
— then the whole `IF`
with b **have** $\langle ?if, s\rangle \longrightarrow_c s'$ **by** (*rule IfTrue*)
}
ultimately
— both cases together give us what we want:
have $\langle ?if, s\rangle \longrightarrow_c s'$ **by** *blast*
}
moreover
— now the other direction:
{ fix s s' **assume** *if*: $\langle ?if, s\rangle \longrightarrow_c s'$
— again, if b is *False* in state s, then the False-branch
— of the `IF` is executed, and both statements do nothing:
hence $\neg b\ s \Longrightarrow s = s'$ **by** *blast*
hence $\neg b\ s \Longrightarrow \langle ?w, s\rangle \longrightarrow_c s'$ **by** *blast*
moreover
— on the other hand, if b is *True* in state s,
— then this time only the *IfTrue* rule can have be used
{ assume b: $b\ s$
with *if* **have** $\langle c; ?w, s\rangle \longrightarrow_c s'$ **by** (*cases set*: *evalc*) *auto*
— and for this, only the Semi-rule is applicable:
then obtain s'' **where**
$\langle c, s\rangle \longrightarrow_c s''$ **and** $\langle ?w, s''\rangle \longrightarrow_c s'$ **by** (*cases set*: *evalc*) *auto*
— with this information, we can build a derivation tree for the `WHILE`
with b
have $\langle ?w, s\rangle \longrightarrow_c s'$ **by** (*rule WhileTrue*)
}
ultimately
— both cases together again give us what we want:

```
  have ⟨?w, s⟩ ⟶c s' by blast
 }
 ultimately
 show ?thesis by blast
qed
```

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

```
lemma
 (WHILE b DO c) ∼ (IF b THEN c; WHILE b DO c ELSE SKIP)
by blast

lemma triv-if:
 (IF b THEN c ELSE c) ∼ c
by blast

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))
by blast

lemma while-equiv:
 ⟨c0, s⟩ ⟶c u ⟹ c ∼ c' ⟹ (c0 = WHILE b DO c) ⟹ ⟨WHILE b DO c', s⟩ ⟶c u
by (induct rule: evalc.induct) (auto simp add: equiv-c-def)

lemma equiv-while:
 c ∼ c' ⟹ (WHILE b DO c) ∼ (WHILE b DO c')
by (simp add: equiv-c-def) (metis equiv-c-def while-equiv)
```

Program equivalence is an equivalence relation.

```
lemma equiv-refl:
 c ∼ c
by blast

lemma equiv-sym:
 c1 ∼ c2 ⟹ c2 ∼ c1
by (auto simp add: equiv-c-def)

lemma equiv-trans:
 c1 ∼ c2 ⟹ c2 ∼ c3 ⟹ c1 ∼ c3
by (auto simp add: equiv-c-def)
```

Program constructions preserve equivalence.

```
lemma equiv-semi:
 c1 ∼ c1' ⟹ c2 ∼ c2' ⟹ (c1; c2) ∼ (c1'; c2')
by (force simp add: equiv-c-def)

lemma equiv-if:
```

c1 $\sim$ *c1'* $\Longrightarrow$ *c2* $\sim$ *c2'* $\Longrightarrow$ (`IF` *b* `THEN` *c1* `ELSE` *c2*) $\sim$ (`IF` *b* `THEN` *c1'* `ELSE` *c2'*)
by (*force simp add*: *equiv-c-def*)

lemma *while-never*: $\langle c, s\rangle \longrightarrow_c u \Longrightarrow c \neq$ `WHILE` ($\lambda s.$ *True*) `DO` *c1*
apply (*induct rule*: *evalc.induct*)
apply *auto*
done

lemma *equiv-while-True*:
(`WHILE` ($\lambda s.$ *True*) `DO` *c1*) $\sim$ (`WHILE` ($\lambda s.$ *True*) `DO` *c2*)
by (*blast dest*: *while-never*)

3.3 Execution is deterministic

This proof is automatic.

theorem $\langle c,s\rangle \longrightarrow_c t \Longrightarrow \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = t$
by (*induct arbitrary*: *u rule*: *evalc.induct*) *blast+*

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 $u = t$
using *assms*
proof (*induct arbitrary*: *u set*: *evalc*)
fix *s u* **assume** $\langle$`SKIP`$,s\rangle \longrightarrow_c u$
thus $u = s$ **by** *blast*
next
fix *a s x u* **assume** $\langle x :== a,s\rangle \longrightarrow_c u$
thus $u = s[x \mapsto a\ s]$ **by** *blast*
next
fix *c0 c1 s s1 s2 u*
assume *IH0*: $\bigwedge u.\ \langle c0,s\rangle \longrightarrow_c u \Longrightarrow u = s2$
assume *IH1*: $\bigwedge u.\ \langle c1,s2\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume $\langle c0;c1,\ s\rangle \longrightarrow_c u$
then obtain *s'* **where**
c0: $\langle c0,s\rangle \longrightarrow_c s'$ **and**
c1: $\langle c1,s'\rangle \longrightarrow_c u$
by *auto*

from *c0 IH0* **have** *s'=s2* **by** *blast*
with *c1 IH1* **show** *u=s1* **by** *blast*
next
fix *b c0 c1 s s1 u*
assume *IH*: $\bigwedge u.\ \langle c0,s\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume *b s* **and** $\langle$`IF` *b* `THEN` *c0* `ELSE` *c1*$,s\rangle \longrightarrow_c u$
hence $\langle c0,\ s\rangle \longrightarrow_c u$ **by** *blast*
with *IH* **show** $u = s1$ **by** *blast*

next
fix *b c0 c1 s s1 u*
assume *IH*: $\bigwedge u.\ \langle c1,s\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume $\neg b\ s$ **and** $\langle \texttt{IF}\ b\ \texttt{THEN}\ c0\ \texttt{ELSE}\ c1,s\rangle \longrightarrow_c u$
hence $\langle c1,\ s\rangle \longrightarrow_c u$ **by** *blast*
with *IH* **show** $u = s1$ **by** *blast*
next
fix *b c s u*
assume $\neg b\ s$ **and** $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c u$
thus $u = s$ **by** *blast*
next
fix *b c s s1 s2 u*
assume IH_c: $\bigwedge u.\ \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = s2$
assume IH_w: $\bigwedge u.\ \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s2\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume *b s* **and** $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c u$
then obtain s' **where**
c: $\langle c,s\rangle \longrightarrow_c s'$ **and**
w: $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s'\rangle \longrightarrow_c u$
by *auto*

from *c* IH_c **have** $s' = s2$ **by** *blast*
with *w* IH_w **show** $u = s1$ **by** *blast*
qed

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 $u = t$
using *assms*
proof (*induct arbitrary*: *u*)
— the simple `SKIP` case for demonstration:
fix *s u* **assume** $\langle \texttt{SKIP},s\rangle \longrightarrow_c u$
thus $u = s$ **by** *blast*
next
— and the only really interesting case, `WHILE`:
fix *b c s s1 s2 u*
assume IH_c: $\bigwedge u.\ \langle c,s\rangle \longrightarrow_c u \Longrightarrow u = s2$
assume IH_w: $\bigwedge u.\ \langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s2\rangle \longrightarrow_c u \Longrightarrow u = s1$

assume *b s* **and** $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s\rangle \longrightarrow_c u$
then obtain s' **where**
c: $\langle c,s\rangle \longrightarrow_c s'$ **and**
w: $\langle \texttt{WHILE}\ b\ \texttt{DO}\ c,s'\rangle \longrightarrow_c u$
by *auto*

from *c* IH_c **have** $s' = s2$ **by** *blast*

with w IH_w **show** $u = s1$ **by** *blast*
qed *blast*+ — prove the rest automatically

end

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}
by (*blast intro*: *conseq*)

lemma *weaken-post*: [| |- {P}c{Q}; !s. Q s --> Q' s |] ==> |- {P}c{Q'}
by (*blast intro*: *conseq*)

lemma *While'*:
assumes |- {%s. P s & b s} c {P} **and** *ALL s*. P s & ¬ b s ⟶ Q s
shows |- {P} WHILE b DO c {Q}
by(*rule weaken-post*[*OF While*[*OF assms*(*1*)] *assms*(*2*)])

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

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

end

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

definition
hoare-valid :: [*assn*,*com*,*assn*] => *bool* (|= {(1-)}/ (-)/ {(1-)} 50) **where**
|= {P}c{Q} = (!s t. ⟨c,s⟩ ⟶c t --> P s --> Q t)

lemma *hoare-sound*: |- {P}c{Q} ==> |= {P}c{Q}
proof(*induct rule*: *hoare.induct*)
case (*While P b c*)

```
  { fix s t
    assume ⟨WHILE b DO c,s⟩ ⟶c t
    hence P s ⟶ P t ∧ ¬ b t
    proof(induct WHILE b DO c s t)
      case WhileFalse thus ?case by blast
    next
      case WhileTrue thus ?case
        using While(2) unfolding hoare-valid-def by blast
    qed

  }
  thus ?case unfolding hoare-valid-def by blast
qed (auto simp: hoare-valid-def)

definition
  wp :: com => assn => assn where
  wp c Q = (%s. !t. ⟨c,s⟩ ⟶c t --> Q t)

lemma wp-SKIP: wp SKIP Q = Q
by (simp add: wp-def)

lemma wp-Ass: wp (x:==a) Q = (%s. Q(s[x↦a s]))
by (simp add: wp-def)

lemma wp-Semi: wp (c;d) Q = wp c (wp d Q)
by (rule ext) (auto simp: wp-def)

lemma wp-If:
 wp (IF b THEN c ELSE d) Q = (%s. (b s --> wp c Q s) &  (~b s --> wp d Q s))
by (rule ext) (auto simp: wp-def)

lemma wp-While-If:
 wp (WHILE b DO c) Q s =
  wp (IF b THEN c;WHILE b DO c ELSE SKIP) Q s
unfolding wp-def by (metis equivD1 equivD2 unfold-while)

lemma wp-While-True: b s ==>
  wp (WHILE b DO c) Q s = wp (c;WHILE b DO c) Q s
by(simp add: wp-While-If wp-If wp-SKIP)

lemma wp-While-False: ~b s ==> wp (WHILE b DO c) Q s = Q s
by(simp add: wp-While-If wp-If wp-SKIP)

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(induct c arbitrary: Q)
  case SKIP show ?case by auto
next
```

```
  case Assign show ?case by auto
next
  case Semi thus ?case by(auto intro: semi)
next
  case (Cond b c1 c2)
  let ?If = IF b THEN c1 ELSE c2
  show ?case
  proof(rule If)
    show |- {λs. wp ?If Q s ∧ b s} c1 {Q}
    proof(rule strengthen-pre[OF - Cond(1)])
      show ∀ s. wp ?If Q s ∧ b s ⟶ wp c1 Q s by auto
    qed
    show |- {λs. wp ?If Q s ∧ ¬ b s} c2 {Q}
    proof(rule strengthen-pre[OF - Cond(2)])
      show ∀ s. wp ?If Q s ∧ ¬ b s ⟶ wp c2 Q s by auto
    qed
  qed
next
  case (While b c)
  let ?w = WHILE b DO c
  have |- {wp ?w Q} ?w {λs. wp ?w Q s ∧ ¬ b s}
  proof(rule hoare.While)
    show |- {λs. wp ?w Q s ∧ b s} c {wp ?w Q}
    proof(rule strengthen-pre[OF - While(1)])
      show ∀ s. wp ?w Q s ∧ b s ⟶ wp c (wp ?w Q) s by auto
    qed
  qed
  thus ?case
  proof(rule weaken-post)
    show ∀ s. wp ?w Q s ∧ ¬ b s ⟶ Q s by auto
  qed
qed

lemma hoare-relative-complete: assumes |= {P}c{Q} shows |- {P}c{Q}
proof(rule strengthen-pre)
  show ∀ s. P s ⟶ wp c Q s using assms
    by (auto simp: hoare-valid-def wp-def)
  show |- {wp c Q} c {Q} by(rule wp-is-pre)
qed

end

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 = (λs. Q(s[x↦a s]))
| awp (Asemi c d) Q = awp c (awp d Q)
| awp (Aif b c d) Q = (λs. (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 = (λs. True)
| vc (Aass x a) Q = (λs. True)
| vc (Asemi c d) Q = (λs. vc c (awp d Q) s & vc d Q s)
| vc (Aif b c d) Q = (λs. vc c Q s & vc d Q s)
| vc (Awhile b I c) Q = (λs. (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(induct c arbitrary: Q)
```

```
  case (Awhile b I c)
  show ?case
  proof(simp, rule While')
    from ‹∀ s. vc (Awhile b I c) Q s›
    have vc: ALL s. vc c I s and IQ: ALL s. I s ∧ ¬ b s ⟶ Q s and
        awp: ALL s. I s & b s --> awp c I s by simp-all
    from vc have |- {awp c I} astrip c {I} using Awhile.hyps by blast
    with awp show |- {λs. I s ∧ b s} astrip c {I}
      by(rule strengthen-pre)
    show ∀ s. I s ∧ ¬ b s ⟶ Q s by(rule IQ)
  qed
qed auto


lemma awp-mono:
  (!s. P s --> Q s) ==> awp c P s ==> awp c Q s
proof (induct c arbitrary: P Q s)
  case Asemi thus ?case by simp metis
qed simp-all

lemma vc-mono:
  (!s. P s --> Q s) ==> vc c P s ==> vc c Q s
proof(induct c arbitrary: P Q)
  case Asemi thus ?case by simp (metis awp-mono)
qed simp-all

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)
using der
proof induct
  case skip
  show ?case (is ? ac. ?C ac)
  proof show ?C Askip by simp qed
next
  case (ass P x a)
  show ?case (is ? ac. ?C ac)
  proof show ?C(Aass x a) by simp qed
next
  case (semi P c1 Q c2 R)
  from semi.hyps obtain ac1 where ih1: ?Eq P c1 Q ac1 by fast
  from semi.hyps obtain ac2 where ih2: ?Eq Q c2 R ac2 by fast
  show ?case (is ? ac. ?C ac)
  proof
    show ?C(Asemi ac1 ac2)
      using ih1 ih2 by simp (fast elim!: awp-mono vc-mono)
  qed
next
  case (If P b c1 Q c2)
```

```
  from If.hyps obtain ac1 where ih1: ?Eq (%s. P s & b s) c1 Q ac1 by fast
  from If.hyps obtain ac2 where ih2: ?Eq (%s. P s & ~b s) c2 Q ac2 by fast
  show ?case (is ? ac. ?C ac)
  proof
    show ?C(Aif b ac1 ac2)
      using ih1 ih2 by simp
  qed
next
  case (While P b c)
  from While.hyps obtain ac where ih: ?Eq (%s. P s & b s) c P ac by fast
  show ?case (is ? ac. ?C ac)
  proof show ?C(Awhile b P ac) using ih by simp qed
next
  case conseq thus ?case by(fast elim!: awp-mono vc-mono)
qed

lemma vcawp-vc-awp: vcawp c Q = (vc c Q, awp c Q)
  by (induct c arbitrary: Q) (simp-all add: Let-def)

end

theory Denotation imports Natural begin

types com-den = (state×state)set

definition
  Gamma :: [bexp,com-den] => (com-den => com-den) where
  Gamma b cd = (λphi. {(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)
  by (unfold Gamma-def mono-def) fast

lemma C-While-If: C(WHILE b DO c) = C(IF b THEN c;WHILE b DO c ELSE SKIP)
apply simp
apply (subst lfp-unfold [OF Gamma-mono])  — lhs only
apply (simp add: Gamma-def)
```

done

lemma *com1*: $\langle c,s\rangle \longrightarrow_c t \Longrightarrow (s,t) \in C(c)$

apply (*induct set*: *evalc*)
apply *auto*

apply (*unfold Gamma-def*)
apply (*subst lfp-unfold*[*OF Gamma-mono*, *simplified Gamma-def*])
apply *fast*
apply (*subst lfp-unfold*[*OF Gamma-mono*, *simplified Gamma-def*])
apply *auto*
done

lemma *com2*: $(s,t) \in C(c) \Longrightarrow \langle c,s\rangle \longrightarrow_c t$
apply (*induct c arbitrary*: *s t*)
apply *auto*
apply *blast*

apply (*erule lfp-induct2* [*OF - Gamma-mono*])
apply (*unfold Gamma-def*)
apply *auto*
done

lemma *denotational-is-natural*: $(s,t) \in C(c) \;=\; (\langle c,s\rangle \longrightarrow_c t)$
by (*fast elim*: *com2 dest*: *com1*)

end

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

begin

3.4 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)
  by (simp add: Denotation.C.simps Id-O-R R-O-Id)


lemma C-skip-cancel2[simp] : C(c;SKIP) = C(c)
  by (simp add: Denotation.C.simps Id-O-R R-O-Id)

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

lemma comappend-correct [simp]: C(c @@ d) = C(c;d)
  apply(induct c)
  apply(simp-all only: C-If-semi conc-If)
  apply(simp-all add: Relation.O-assoc)
  done

fun unfold :: nat × com ⇒ com
where
  uf-skip : unfold(n, SKIP)     = 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* : ∀ *x*. *C* (*unfold* (*x*, *c*)) = *C c*
shows *C*(*unfold*(*n*,*WHILE b DO c*)) = *C*(*WHILE b DO c*)
proof (*induct n*)
case *0* **then show** *?case*
by(*simp add*: *Denotation.C.simps H*)
next
case (*Suc n*) **then show** *?case*
apply(*subst uf-while*,*subst if-P*, *simp*)
apply(*rule-tac s* = *n* **and** *t* = *Suc n* − *1* **in** *subst*,*arith*)
apply(*simp only*: *Denotation.C.simps comappend-correct*)
apply(*simp only*: *Denotation.C.simps* [*symmetric*] *H*)
apply(*simp only*: *Denotation.C-While-If*)
done
qed

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

lemma *unfold-correct-aux2* :
C(*unfold*(*n*,*c*;*d*))= *C*(*unfold*(*n*,*c*) ; *unfold*(*n*, *d*))
proof (*induct c*) **print-cases**
case *SKIP* **then show** *?case* **by**(*simp*)
next
case (*Assign loc E*) **then show** *?case* **by**(*case-tac d*, *simp-all*)
next
case (*Semi c1 c2*) **then show** *?case* **by**(*case-tac d*, *simp-all*)
next
case (*Cond cond then-branch else-branch*) **then show** *?case*
apply (*case-tac d*, *simp-all*)
apply (*simp-all only*: *C.simps*[*symmetric*] *C-If-semi*)
by *auto*
next
case (*While cond body*) **then show** *?case* **by**(*case-tac d*, *simp-all*)
qed

lemma *unfold-correct* [*rule-format*]: ∀ *x*. (*C*(*unfold*(*x*,*c*)) = *C*(*c*))
proof(*induct c*)
case *SKIP* **then show** *?case* **by** *simp*
next
case (*Assign loc E*) **then show** *?case* **by** *simp*

next
case (*Semi c1 c2*) **then show** *?case*
by (*cases c1*, *cases c2*,
simp-all add: *unfold-correct-aux2*)
next
case (*Cond cond then-branch else-branch*) **then show** *?case* **by** *simp*
next
case (*While cond body*) **then show** *?case*
by (*intro allI unfold-correct-aux1*,*auto*)
qed

lemma *wp-unfold* : *wp* (*c*) (*p*) = *wp*(*unfold*(*n*,*c*)) (*p*)
by(*simp add*: *wp-def unfold-correct denotational-is-natural*[*symmetric*])

lemma *wp-test* : $\forall \sigma.\ P\ \sigma \longrightarrow wp\ (unfold(k,c))\ Q\ \sigma \Longrightarrow |-\ \{P\}\ c\ \{Q\}$
apply (*rule Hoare.strengthen-pre*)
apply (*simp add*: *wp-unfold*[*symmetric*])
apply (*rule wp-is-pre*)
done

3.5 Symbolic Evaluation Rule-Set

lemma *If-split*:
$[\![\ b\ s \Longrightarrow \langle c0,s\rangle \longrightarrow_c s';$
$\neg\ b\ s \Longrightarrow \langle c1,s\rangle \longrightarrow_c s'\ ]\!]$
$\Longrightarrow \langle IF\ b\ THEN\ c0\ ELSE\ c1,s\rangle \longrightarrow_c s'$
by (*cases b s*, *simp-all*)

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

by(*cases b s*, *simp-all*)

3.6 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 −
have *hoare-sound-complete* : (|- {*Pre*} *c* {*Post*}) = (|= {*Pre*} *c* {*Post*})
by(*auto intro*!: *hoare-sound hoare-relative-complete*)
show *?thesis*
by(*simp only*: *hoare-sound-complete hoare-valid-def*
denotational-is-natural[*symmetric*] *unfold-correct*)

qed

3.7 Tactic Set-up

ML⟪

fun contains-eval n thm =
    let fun T(Natural.evalc,-) = true | T - = false
    in Term.exists-Const T (term-of(cprem-of thm n)) end

⟫
ML⟪
local open TestGen in

fun thyp-ify-partial-evaluations ctxt =
    (COND' contains-eval (thyp-ify ctxt) (K all-tac))

end

⟫

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

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

lemma *assign-sequence*:
$\langle a{:=}{=}\ e;\ c,s\rangle \longrightarrow_c s' = \langle c,s[a \mapsto e\ s]\rangle \longrightarrow_c s'$
by(*simp only*:*Natural.semi Natural.assign one-point-rules*)

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⟪

```
local open TestGen HOLogic in

fun generate-program-splitter ctxt simps depth no thm =
    let val thy = theory-of-thm thm
        val @{term Hoare.hoare} $ PRE $ PROG $ - = dest-Trueprop(term-of(cprem-of thm
1));
        val S = (Thm.trivial (cterm-of thy (mk-Trueprop
                        (@{term Hoare.hoare} $ PRE $ PROG $
                         Free(POSTCONDITION,
                         @{typ (Com.loc ⇒ nat) ⇒ bool}))))
        val S = S |$> (res-inst-tac ctxt (∗ [(n1, Int.toString depth)] ∗)
                                    [((n,1),Int.toString depth)]
                         (@{thm symbolic-eval-test} RS iffD2) no)
                  |$> (safe-tac ctxt)
                  |$> (asm-full-simp-tac((global-simpset-of thy)
                          addsimps
                         (@{thms Append.simps} @
                          @{thms unfold.simps} @
                          [@{thm uf-while}])) 1)
                  |$> (asm-full-simp-tac( HOL-ss
                          addsimps
                         (@{thms symbolic-evaluation2} @
                          @{thms memory-model} @ simps @
                          [@{thm update-def}])) 1)
                  |$> (safe-tac ctxt)
                  |$> (ALLGOALS(COND' contains-eval (thyp-ify ctxt) (K all-tac)))

    in thm |> (rtac (Drule.export-without-context S) 1)
  end

end (∗ local ∗)
```

⟫

end

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

3.8 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*)

3.9 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′*

apply(*simp add*: *squareroot-def uf-while*)
apply(*rule If-split*,*simp-all add*: *update-def no-alias*)+

The resulting proof state capturing the test hypothesis as well as the resulting 4 evaluation paths (no entry into loop, 1 pass, 2 passes and 3 passes through the loop) looks as follows:

1. *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))) ≤ *s a* ⟹
⟨*WHILE* λ*s*. *s sqsum*
≤ *s a* *DO* *i* :== λ*s*. *Suc* (*s i*) ; (*tm* :== λ*s*.
Suc (*Suc* (*s tm*)) ; *sqsum* :== λ*s*. *s tm* + *s sqsum*),*s*
(*i* := *Suc* (*Suc* (*Suc 0*)),
tm := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))),
sqsum :=
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))))))))))⟩
⟶$_c$ *s′*
2. ⟦*Suc* (*Suc* (*Suc* (*Suc 0*))) ≤ *s a*;
¬ *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))) ≤ *s a*⟧
⟹ *s′* = *s*

$(i := Suc\ (Suc\ 0),\ tm := Suc\ (Suc\ (Suc\ (Suc\ 0))),$
$sqsum := Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))))$
3. $[\![Suc\ 0 \leq s\ a;\ \neg\ Suc\ (Suc\ (Suc\ 0))) \leq s\ a]\!]$
$\Longrightarrow s' = s$
$(i := Suc\ 0,\ tm := Suc\ (Suc\ 0)),$
$sqsum := Suc\ (Suc\ (Suc\ (Suc\ 0))))$
4. $\neg\ Suc\ 0 \leq s\ a \Longrightarrow s' = s(tm := Suc\ 0,\ sqsum := Suc\ 0,\ i := 0)$

oops

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

3.10 Testing Specifications

thm *symbolic-evaluation2*

Slow Motion Interactive Version (for demonstrations).

lemma *whitebox-test*:
assumes *no-alias*[*simp*] : $sqsum \neq i \wedge i \neq sqsum \wedge tm \neq sqsum \wedge$
$sqsum \neq tm \wedge sqsum \neq a \wedge a \neq sqsum \wedge$
$tm \neq i \wedge i \neq tm \wedge tm \neq a \wedge a \neq tm \wedge$
$a \neq i \wedge i \neq a$
shows $|-$ {*pre*} *squareroot tm sqsum i a* {*post a i*}

apply(*simp add*: *squareroot-def pre-def*)
apply(*rule-tac n1 = 3* **in** *iffD2*[*OF symbolic-eval-test*])

apply(*safe*, *simp add*: *Append.simps unfold.simps uf-while*)
apply(*simp only*: *symbolic-evaluation2*
memory-model no-alias update-def,
safe)
apply(*tactic ALLGOALS*(*TestGen.COND' contains-eval*
(*TestGen.thyp-ify* @{*context*})
(*K all-tac*)))

apply(*simp-all*)

sorry

Automated Version:

lemma *whitebox-test2*:
assumes *no-alias*[*simp*] : $sqsum \neq i \wedge i \neq sqsum \wedge tm \neq sqsum \wedge$
$sqsum \neq tm \wedge sqsum \neq a \wedge a \neq sqsum \wedge$
$tm \neq i \wedge i \neq tm \wedge tm \neq a \wedge a \neq tm \wedge$
$a \neq i \wedge i \neq a$
shows $|-$ {*pre*} *squareroot tm sqsum i a* {*post a i*}

apply(*simp add*: *squareroot-def pre-def*)
apply(*tactic generate-program-splitter* @{*context*} (@{*thms no-alias*}) *3 1*)

apply(*simp-all*)

The resulting proof state captures the essence of this white box test:

1. THYP
(∀ *x xa xb xc xd xe xf*.
⟨*WHILE* λ*s*. *s x*
≤ *s xa DO xb* :== λ*s*. *Suc* (*s*
xb) ; (*xc* :== λ*s*. *Suc* (*Suc* (*s xc*)) ; *x* :== λ*s*. *s xc* + *s x*),*xf*
(*xb* := *Suc* (*Suc* (*Suc* 0)),
xc := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))))),
x := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0))))))))))))))⟩
⟶$_c$ *xe* ⟶
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))))))) ≤ *xf xa* ⟶
xd xe)
2. ⋀*s*. ⟦*Suc* (*Suc* (*Suc* (*Suc* 0))) ≤ *s a*;
¬ *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))))))) ≤ *s a*⟧
⟹ *post a i*
(*s*(*i* := *Suc* (*Suc* 0), *tm* := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))),
sqsum :=
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))))))))
3. ⋀*s*. ⟦*Suc* 0 ≤ *s a*; ¬ *Suc* (*Suc* (*Suc* (*Suc* 0))) ≤ *s a*⟧
⟹ *post a i*
(*s*(*i* := *Suc* 0, *tm* := *Suc* (*Suc* (*Suc* 0)),
sqsum := *Suc* (*Suc* (*Suc* (*Suc* 0)))))
4. ⋀*s*. ¬ *Suc* 0 ≤ *s a* ⟹ *post a i* (*s*(*tm* := *Suc* 0, *sqsum* := *Suc* 0, *i* := 0))

Now testing all paths for compliance to post condition:

apply(*simp-all add*: *no-alias post-def*)

In this special case—arithmetic constraints—the system can even **verify** these constraints, i.e. the simplifier shows that all postconditions follow from the initial constraints and the computed relation between pre-state and post state.

1. THYP
(∀ *x xa xb xc xd xe xf*.
⟨*WHILE* λ*s*. *s x*
≤ *s xa DO xb* :== λ*s*. *Suc* (*s*
xb) ; (*xc* :== λ*s*. *Suc* (*Suc* (*s xc*)) ; *x* :== λ*s*. *s xc* + *s x*),*xf*
(*xb* := *Suc* (*Suc* (*Suc* 0)),
xc := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))))),
x := *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0))))))))))))))⟩
⟶$_c$ *xe* ⟶
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* 0)))))))) ≤ *xf xa* ⟶
xd xe)

To say it loud and clearly: The white box test decomposes the original specification into a test hypothesis for cases with $3^3 = 9 <= sa$ and all other cases (e.g. $2^2 = 4 <= sa \wedge sa < 9$). The latter have been proven automatically.

oops

3.11 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* $\{post\ a\ i\}$

We fire the basic white-box scenario:

apply (*rule wp-test* [*of - 3*])

Given the concrete unfolding factor and the concrete program term, standard normalization yields an "Path Exhaustion Theorem" with the explicit test hypothesis:

apply(*auto simp*: *squareroot-def update-def no-alias uf-while*)
apply(*tactic ALLGOALS*(*TestGen.COND' contains-eval*
(*TestGen.thyp-ify* @{*context*})
(*K all-tac*)))

and we reach the following instantiation of a white-box test-theorem (with explicit test-hypothesis for the uncovered paths):

1. $\bigwedge\sigma$. $[\![pre\ \sigma;\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))) \leq \sigma\ a]\!]$
$\Longrightarrow wp\ (\texttt{WHILE}\ \lambda s.\ s\ sqsum$
$\leq s\ a\ \texttt{DO}\ i :== \lambda s.$
$Suc\ (s\ i)\ ;\ (tm :== \lambda s.\ Suc\ (Suc\ (s\ tm))\ ;\ sqsum :== \lambda s.\ s\ tm + s\ sqsum\ ))$
$(post\ a\ i)$
$(\sigma(i := Suc\ (Suc\ (Suc\ 0)),$
$tm := Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))),$
$sqsum :=$
$Suc\ (Suc\ (Suc\ (Suc\ (Suc$
$(Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))))))))))))$

2. $\bigwedge\sigma$. $[\![pre\ \sigma;\ Suc\ (Suc\ (Suc\ (Suc\ 0))) \leq \sigma\ a;$
$\neg\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))) \leq \sigma\ a]\!]$
$\Longrightarrow post\ a\ i$
$(\sigma(i := Suc\ (Suc\ 0),\ tm := Suc\ (Suc\ (Suc\ (Suc\ 0)))),$
$sqsum :=$
$Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))))$
3. $\bigwedge\sigma$. $[\![pre\ \sigma;\ Suc\ 0 \leq \sigma\ a;\ \neg\ Suc\ (Suc\ (Suc\ (Suc\ 0))) \leq \sigma\ a]\!]$
$\Longrightarrow post\ a\ i$
$(\sigma(i := Suc\ 0,\ tm := Suc\ (Suc\ (Suc\ 0)),$
$sqsum := Suc\ (Suc\ (Suc\ (Suc\ 0)))))$
4. $\bigwedge\sigma$. $[\![pre\ \sigma;\ \neg\ Suc\ 0 \leq \sigma\ a]\!]$
$\Longrightarrow post\ a\ i\ (\sigma(tm := Suc\ 0,\ sqsum := Suc\ 0,\ i := 0))$

Now we allso perform the "tests" by symbolic execution:

apply(*auto simp*: *no-alias pre-def post-def*)

which leaves us just with test-hypothesis case; for all paths *not* leading to a remaining while, the program is correct.

1. $\bigwedge\sigma$. $Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))) \leq \sigma\ a \Longrightarrow$
$wp\ (\texttt{WHILE}\ \lambda s.\ s\ sqsum$
$\leq s\ a\ \texttt{DO}\ i :== \lambda s.\ Suc\ (s$
$i)\ ;\ (tm :== \lambda s.\ Suc\ (Suc\ (s\ tm))\ ;\ sqsum :== \lambda s.\ s\ tm + s\ sqsum\ ))$
$(\lambda s.\ s\ i * s\ i \leq s\ a \wedge s\ a < Suc\ (s\ i + (s\ i + s\ i * s\ i)))$
$(\sigma(i := Suc\ (Suc\ (Suc\ 0)),$
$tm := Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))),$
$sqsum :=$
$Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc$
$(Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))))))))))))))$

oops

end

References

[1] HOL-TestGen. http://www.brucker.ch/projects/hol-testgen/.

[2] IMP. http://isabelle.in.tum.de/library/HOL/IMP/Denotation.html.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the international symposium on Software testing and analysis*, pages 123–133. ACM Press, 2002.

[4] A. D. Brucker and B. Wolff. Interactive testing using HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Formal Approaches to Testing of Software*,

number 3997 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2005.

[5] A. D. Brucker and B. Wolff. Extensible universes for object-oriented data models. In J. Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, number 5142 in Lecture Notes in Computer Science, pages 438–462. Springer-Verlag, Heidelberg, 2008.

[6] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, Heidelberg, 2009.

[7] W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, 2004.

[8] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[9] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003.

[10] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.