Verifying Test-Hypotheses

An Experiment in Test and Proof

Achim D. Brucker[1]

SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany

Lukas Brügger[2]

Information Security, ETH Zurich, 8092 Zurich, Switzerland

Burkhart Wolff[3]

Universität des Saarlandes, 66041 Saarbrücken, Germany

Abstract

HOL-TestGen is a specification and test case generation environment extending the interactive theorem prover Isabelle/HOL. The HOL-TestGen method is two-staged: first, the original formula, called *test specification*, is partitioned into *test cases* by transformation into a normal form called *test theorem*. Second, the test cases are analyzed for ground instances (the *test data*) satisfying the constraints of the test cases. Test data were used in an automatically generated test-driver running the program under test. Particular emphasis is put on the control of explicit test hypotheses which can be proven over concrete programs. As such, explicit test hypotheses establish a logical link between a validation by test and a validation by proof. Since HOL-TestGen generates explicit test hypotheses and makes them amenable to formal proof, the system is in a unique position to explore the relations between them at an example.

Keywords: symbolic test case generations, black box testing, theorem proving, formal verification, Isabelle/HOL

1 Introduction

Today, essentially two software validation techniques are used: *software verification* and *software testing*. As far as symbolic verification methods and model-based testing techniques are concerned, the interest among researchers in the mutual fertilization of these fields is growing.

The HOL-TestGen system [4,3,2] is designed to explore and exploit the complementary assets of these approaches. Built on top of a widely-used interactive

[1] Email: achim.brucker@sap.com

[2] Email: lukas.bruegger@inf.ethz.ch

[3] Email: wolff@wjpserver.cs.uni-sb.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

theorem prover, it provides automatic procedures for test case generation and test data selection, as well as interactive means to perform logical massages of the intermediate results by derived rules. The core of HOL-TestGen is a test case generation procedure that decomposes a *test specification* (TS), i. e., a test-property over a program under test, into a semantically equivalent *test theorem* of the form:

$$[\![\mathrm{TC}_1; \ldots; \mathrm{TC}_n; \mathrm{THYP}\ H_1; \ldots; \mathrm{THYP}\ H_m]\!] \Longrightarrow \mathrm{TS}$$

where the TC_i ($i \in 1...n$) are the *test cases* (i. e., test input that still contains, possibly constrained, variables) and THYP is a constant (semantically defined as identity) used to mark the explicit *test hypotheses* H_j ($j \in 1...m$) that are underlying this test. Thus, a test theorem has the following meaning:

> If the program under test passes the tests with a witness for all test cases TC_i successfully, and if it satisfies all test hypotheses TH_j, it is correct with respect to the test specification TS.

In this sense, the test theorem bridges the gap between test and verification. Furthermore, testing can be viewed as systematic weakening of specifications.

Establishing a formal link between test and proof, explicit hypotheses are an ideal candidate for addressing some key-questions:

(i) What is the *nature* of the relation between test and proof? Do standard test hypotheses make sense?

(ii) Does a test *approximate* a full-blown verification? Is the underlying test-method complete in this sense?

(iii) Can tests *contribute* or even *facilitate* proofs?

This paper consists of two parts: In part one, we introduce HOL-TestGen to make this paper self-contained, and show its explicit test hypotheses generation using a small example. In part two, we perform the standard workflow of HOL-TestGen on a standard algorithm (insertion sort), and verify the resulting test hypotheses by formal Isabelle/HOL proofs, and evaluate them by some empirical data.

2 Foundations

2.1 Isabelle

Isabelle [9] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and Higher-order logic (HOL), which we choose as framework for HOL-TestGen.

While Isabelle/HOL is usually denoted as a proof assistant, we use it as symbolic computation environment. Implementations on Isabelle/HOL can re-use existing powerful deduction mechanisms such as higher-order resolution and rewriting, and the overall environment provides a large collection of components ranging from documentation generators and code generators to (generic) decision procedures for datatypes and Presburger Arithmetic.

Isabelle can easily be controlled by a programming interface on its implementation level in SML in a logically safe way, as well as on the Isar level, i. e., a tactic proof language in which interactive and automated proofs can be mixed arbitrarily. Documents in the Isar format, enriched by the commands provided by HOL-TestGen, can be processed incrementally within Proof General (see Section 3) as well as in batch mode. These documents can be seen as a formal and technically checked test plan of a program under test.

Isabelle processes rules and theorems of the form $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, also denoted as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$. They can be understood as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}." Further, Isabelle provides a built-in meta-quantifier $\bigwedge x_1, \ldots, x_m.\ [\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$ for representing "fresh free variables not occurring elsewhere" thus avoiding the usual provisos on logical rules. In particular, the presentation of subgoals uses this format. We will refer to assumptions A_i also as *constraints* in this paper.

2.2 Higher-order Logic

Higher-order logic (HOL) [6,1] is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, since e. g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

Isabelle/HOL provides also a large collection of theories like sets, lists, multisets, orderings, and various arithmetic theories. Furthermore, it provides the means for defining datatypes and recursive function definitions over them in a style similar to a functional programming language.

3 The HOL-TestGen System: An Overview

HOL-TestGen is an *interactive* (semi-automated) test tool for specification based tests. Its theory and implementation has been described elsewhere [4,2]; here, we briefly review the main concepts and outline the standard workflow. The latter is divided into four phases: writing the *test specification* TS, generation of *test cases* TC (which contain, possibly constrained, variables) along with a *test theorem* for the TS, generation of *test data* TD, i. e., constraint-free instances of test cases where all variables have been replaced by ground instances, and the *test execution* (*result verification*) phase involving runs of the "real code" of the program under test; Figure 1 illustrates the overall workflow. Once a test theory is completed, documents can be generated that represent a formal test plan. The test plan containing the test theory, test specifications, configurations of the test data and test script generation commands, possibly extended by proofs for rules that support the overall process, is written in an extension of the Isar language. It can be processed in batch mode, but also using the Proof General interface interactively (see upper window in Figure 2). This interface allows for interactively stepping through a test theory in the upper sub-window while the sub-window below shows the corresponding system state. This may be a proof state in a test theorem development, a list of generated test data

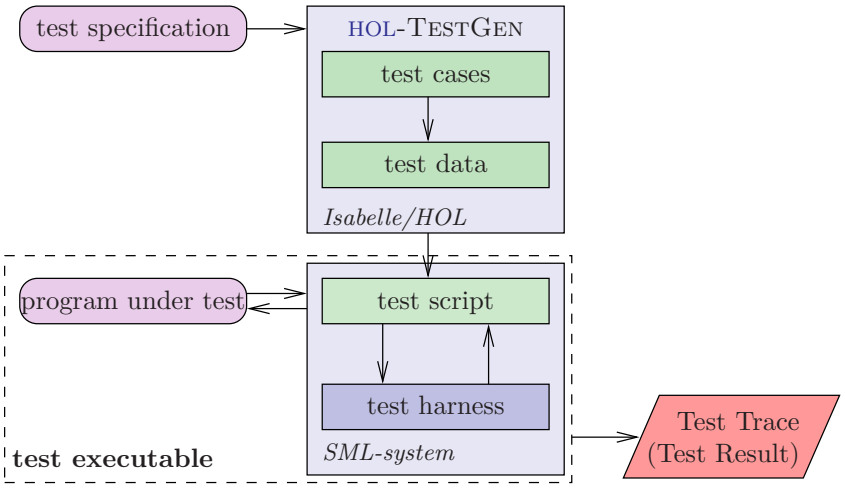

Figure 1. Overview of the Standard Workflow of HOL-TestGen

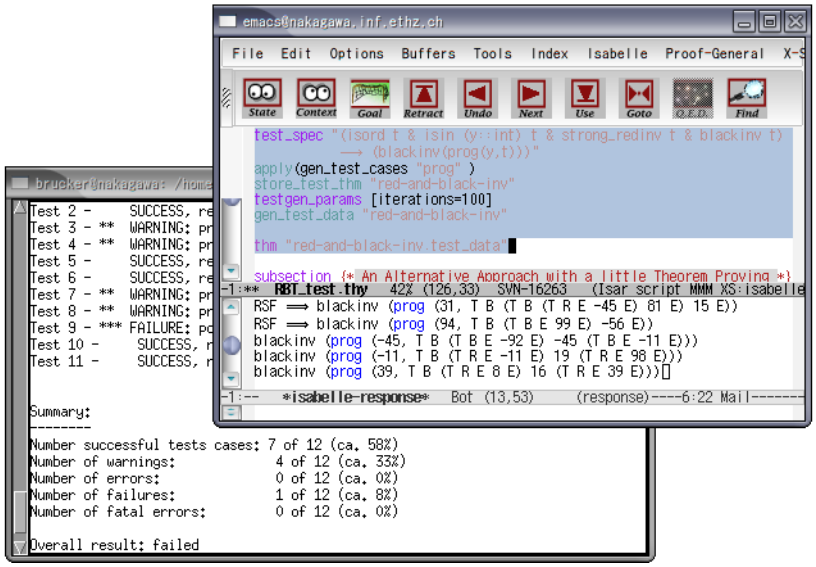

Figure 2. A HOL-TestGen Session Using Proof General

or a list of test hypotheses. After test data generation, HOL-TestGen produces a *test script* driving the test using the provided *test harness*. The test script together with the test harness stimulate the code for the program under test built into the *test executable*. Executing the *test executable* runs the test and yields a *test trace* showing errors in the implementation (see lower window in Figure 2).

4 Test Case Generation with Explicit Test-Hypotheses

In this section, we describe the test case generation procedure of HOL-TestGen. It is driven by an exhaustive backward-application of a standard tableaux calculus combined with certain normal-form computations eliminating redundancy. Interleaved with this partitioning process (similar to the DNF-based approach of Dick and Faivre [7]), test hypothesis rules are generated on the fly and applied to certain subgoals in a backward manner. In the following, we present two well-known

kinds of test hypotheses. Following the terminology of Gaudel [8], these are called *uniformity* and *regularity* hypotheses.

4.1 Inserting Uniformity Hypotheses

Uniformity hypotheses have the form:

$$\text{THYP}(\exists x_1 \ldots x_n.\ P\ x_1, \ldots, x_n \rightarrow \forall x_1 \ldots x_n.\ P\ x_1 \ldots x_n)$$

where THYP is a constant defined as the identity; this constant is used as a marker to protect this type of formulae from other decomposition steps in the generation procedure. Semantically, this kind of hypothesis expresses that whenever there is a successful test for a test case, it is assumed that the program will behave correctly for *all* data of this test case.

The derived rule in natural deduction format, expressing this kind of test theorem transformation, reads as follows:

$$\frac{P\ ?X_1 \ldots ?X_n \quad \text{THYP}(\exists x_1 \ldots x_n.\ P\ x_1 \ldots x_n \rightarrow \forall x_1 \ldots x_n.\ P\ x_1 \ldots x_n)}{\forall x_1 \ldots x_n.\ P\ x_1 \ldots x_n}$$

Here, the $?X_i$ are just meta variables, i. e., place-holders for arbitrary terms. This rule can also be applied for arbitrary formulae just containing free variables since universal quantifiers may be introduced for them beforehand.

Tactically, these hypotheses are introduced *at the end* of the test case generation process, i. e., when all other rules can no longer be applied. Using a uniformity hypothesis for each (non-THYP) clause allows for the replacement of free variables by meta-variables which can be instantiated by ground terms during the test data selection phase later. This transformation is logically sound.

For example, assume the following test specification:

if x < 0 then PUT x else PUT − x

where PUT is a place-holder for the program under test. The case generation produces the following test theorem:

test : if 0 ≤x then PUT x else PUT −x
1: 0 ≤ ?X1 ⟹ PUT ?X1
2: THYP((∃ x. 0 ≤x ⟶ PUT x)
⟶ (∀ x. 0 ≤x ⟶ PUT x))\\
3: ?X2 < 0 ⟹ PUT −X2\\
4: THYP((∃ x. x < 0 ⟶ PUT −x)
⟶ (∀ x. x<0 ⟶ PUT −x))

The test data selection phase will easily generate instances of the test cases, e. g., PUT 3 and PUT (−(−4)), (satisfying the constraints) to be used in a black-box test. If we have the implementation of PUT in our hands, we could also verify the test hypotheses; provided that execution paths in the concrete program correspond to classes of test cases, we gain knowledge from the test for the verification.

4.2 Inserting Regularity Hypotheses

In the following, we address the problem of test case generation for quantifiers (or, equivalently: free variables) ranging over recursive datatypes such as lists or trees. As an introductory example, we consider the datatype for lists which is defined as follows in Isabelle/HOL:

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

This statement is part of the Isabelle/HOL library and represents (together with automatically derived theorems like l = [] ∨ ∃ a, l '. l = a#l' or the induction theorem) the (background) *test theory* for the current example. Moreover, there are ways to define the alternative syntax [x1, x2, x3] for (Cons x1 (Cons x2 (Cons x3 []))) or x1#x2#x3.

Now assume we want to test a program `PUT` running over lists, i. e., the *test specification* looks as follows:

```
PUT (l :: int  list )
```

When generating the test cases for recursive data structures, HOL-TestGen generates instead of a regularity hypothesis (following the terminology introduced by Gaudel [8]), a so called *data exhaustion theorem*. This theorem is generated on-the-fly, its form depends on the structure of the corresponding datatype. The intuitive meaning of such a regularity hypothesis is: assuming that a predicate P is true for all data x whose *size* (denoted by $|x|$) is less than a given depth k, P is always true. For the user-defined value $k = 2$ and for the type list, we get:

$$\frac{\begin{array}{cccccc}[x = [\,]] & & [x = [a]] & & [x = [a, b]] & \\ \vdots & & \vdots & & \vdots & \\ P(x) & \bigwedge a. & P(x) & \bigwedge a\ b. & P(x) & \mathrm{THYP}(\forall x.\ 2 < |x| \rightarrow P(x))\end{array}}{P(x)}$$

The equalities introduced by this rule lead to the following test theorem (we omit the uniformity hypotheses insertion here):

```
test :  PUT l
 1: PUT []
 2: PUT [?X1]
 3: PUT [?X2,?X3]
 4: THYP(∀ x. 2 < |x| ⟶P(x))
 5: ...
```

and, again, it is an easy game for a random-based test data selection method to provide instances (i. e., *test data*) for these constraint free *test cases*.

5 Test of our Running Example

In the following we proceed with our example of a standard sorting algorithm by the usual workflow for HOL-TestGen: we give the test theory, the test specification,

generate the test theorem, and extract test data.

The recursive definition of the predicate " is_sorted " can be given in HOL similarly as in a functional programming language:

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

We proceed by the test specification and the subsequent test case generation. The test specification simply says, that whatever list we give PUT, it should yield a sorted list in the sense given above. [4]

```
test_spec "is_sorted(PUT (l :: int list ))"
  apply(gen_test_cases "PUT")
```

The test case generation, based on the (implicit) default value $k = 3$ (depth of the data exhaustion theorem), results in the test theorem:

```
1: is_sorted (PUT [])
2: is_sorted (PUT [?X1])
3: THYP ((∃ x. is_sorted (PUT [x])) ⟶ (∀ x. is_sorted (PUT [x])))
4: is_sorted (PUT [?X2, ?X3])
5: THYP ((∃ x xa. is_sorted (PUT [xa, x]))
    ⟶ (∀ x xa. is_sorted (PUT [xa, x])))
6: is_sorted (PUT [?X4, ?X5, ?X6])
7: THYP ((∃ x xa xb. is_sorted (PUT [xb, xa, x]))
    ⟶ (∀ x xa xb. is_sorted (PUT [xb, xa, x])))
8: THYP (3 < |l| ⟶ is_sorted (PUT l))
```

Since all test cases are unconstrained, the test data selection phase picks just arbitrary integer values for the meta-variables ?X1, . . . , ?X6.

It turns out that uniformity and regularity hypotheses are an amazingly flexible form of systematic weakening of specifications. Our procedure can in particular handle them by using refined constraint solving techniques, test specifications that involve preconditions like:

```
test_spec "is_sorted l ⟶ is_sorted (PUT (a, l :: int list ))"
```

That means that tests which are constrained to lists l which are sorted. Thus, HOL-TestGen has been used for substantial case studies involving the unit tests of red-black tree library implementations as well as firewalls [3,5]. In the latter, regularity hypotheses can be used to establish coverage of an automaton accepting a (protocol) language.

[4] This test specification is not complete, e. g., it does not require that the result is sorted *permutation* of the input.

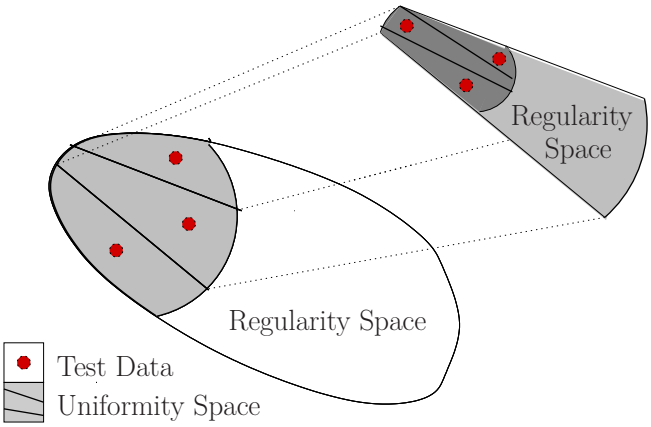

Figure 3. Refining Test Data Spaces by Testing Hypothesis.

6 Validating Explicit Hypotheses

Now we will focus on the following questions:

(i) What is the *nature* of the relation between test and proof? Do standard test hypotheses make "sense"?

(ii) Does a test *approximate* a full-blown verification? Is the underlying test-method complete in this sense?

(iii) Can tests *contribute* or even *facilitate* proofs?

We address these questions by an attempt to *test* the hypotheses, and an attempt to *formally verify* them. With respect to the latter, we will specify the insertion-sort algorithm and use it in a (post-hoc) white-box setting. This verification will shed some light on the role of tests and proofs.

6.1 Refining Test-Classes by Testing Test-Hypotheses

Re-feeding explicit test hypotheses into the testing process is easy in principle: just remove the THYP operator, which protects the formula inside from further decomposition during test case generation, and generate another test theorem from it. Figure 3 illustrates the case of a refinement of a uniformity hypothesis. Refining a specific uniformity hypothesis, i. e., a specific partition of the uniformity space, results in both more fine-grained uniformity spaces (and thus test data) and a new partition of the regularity space (right side of Figure 3).

While the approach leads to the construction of more test cases and therefore more distinct test data in principle, the presented example will not work for HOL-TestGen since our system treats test classes induced by basic types like integer as atomic. A list of integer of length two is therefore not further separated. This kind of incompleteness of HOL-TestGen, however, is merely a weakness than a serious limitation and can be overcome easily by, for example, adding case splits over integer variables via $x < k \vee k \leq x$ where k is a random value. Nevertheless, the approach works with the existing HOL-TestGen in the red-black tree example [3], where trees were refined in each test case.

6.2 Verifying Test-Hypothesis by Formal Proof

As a prerequisite, we have to give a program for our test: this reflects the change to a white-box testing scenario. As in all white-box test procedures, we make the meta-assumption that the program under test *is* the same function as its presentation inside the tool—and thus amenable to analysis on this presentation.

In our case, the functional program can be easily defined by:

```
consts ins :: "[int, int list] ⇒ int list"
primrec "ins x [] = [x]"
        "ins x (y#ys) = (if (x < y) then x#(ins y ys) else (y#(ins x ys)))"

consts sort :: "int list ⇒ int list"
primrec "sort [] = []"
        "sort (x#xs) = ins x (sort xs)"
```

The proof of the uniform hypotheses (where `PUT` is now instantiated with sort, i. e., our presentation of the program) is straightforward and actually automatic. For the sake of this paper, we present the essential proof steps for one of the test hypotheses of the test theorem shown in Section 5 in detail. For example, we prove with Isabelle the second uniformity hypothesis (c.f. line 5 in the test theorem for is_sorted shown on page 7) as follows:

```
lemma uniformity_2_verified: "THYP ((∃ x xa. is_sorted (sort [xa, x]))
                                    ⟶ (∀ x xa. is_sorted (sort [xa, x])))"
```

We standardize the test-hypothesis to the core and get:

$$\bigwedge x\ xa\ x'\ xa''.\ \text{is\_sorted}\ (\text{sort}\ [xa', x']) \Longrightarrow \text{is\_sorted}\ (\text{sort}\ [xa, x])$$

The only way to proceed is by discarding the assumption (see discussion below):

$$\bigwedge x\ xa.\ \text{is\_sorted}\ (\text{sort}\ [xa, x])$$

Unfolding sort yields:

$$\bigwedge x\ xa.\ \text{is\_sorted}\ (\text{ins}\ xa\ (\text{ins}\ x\ []))$$

and after unfolding of ins we get:

$$\bigwedge x\ xa.\ \text{is\_sorted}\ (\text{if}\ xa < x\ \text{then}\ [xa, x]\ \text{else}\ [x, xa])$$

Case-splitting results in:

1: $\bigwedge x\ xa.\ xa < x \Longrightarrow \text{is\_sorted}\ [xa, x]$
2: $\bigwedge x\ xa.\ \neg\, xa < x \Longrightarrow \text{is\_sorted}\ [x, xa]$

Evaluation of is_sorted yields:

$$\begin{aligned}1:\ \bigwedge x\ xa.\ & xa < x \\ & \Longrightarrow \text{case}\ [x]\ \textbf{of}\ []\ \Rightarrow \text{True} \\ & \quad \mid y\ \#\ ys \Rightarrow (xa < y \vee xa = y) \\ & \quad\ \wedge (\text{case}\ []\ \textbf{of}\ []\ \Rightarrow \text{True} \\ & \quad\quad \mid y\ \#\ ys \Rightarrow (x < y \vee x = y) \wedge \text{True})\end{aligned}$$

2: $\bigwedge$ x xa. $\neg$ xa < x
$\Longrightarrow$ case [xa] **of** [] $\Rightarrow$ True
| y # ys $\Rightarrow$ x < y $\vee$ x = y)
$\wedge$ (case [] **of** [] $\Rightarrow$ True
| y # ys $\Rightarrow$ (xa < y $\vee$ xa = y) $\wedge$ True)

which can be reduced to:

1: $\bigwedge$ x xa. $\neg$ xa < x $\Longrightarrow$ x < xa $\vee$ x = xa

which results by arithmetic reasoning in True.

The proof reveals that the test is in itself irrelevant for the proof of uniformity: the existential part has to be discarded since it leads to nowhere. Only in the exceptional case that the quantifier ranges over a singleton set and therefore $x = x'$ and $xa = xa'$ the assumption can be used; in this case, the test is just the verification. In all other cases, the assumption ranges over *different* variables than the conclusion. This fact is inherently related to the scheme of uniformity hypothesis and not specific to our example.

The three uniformity test hypotheses together can be combined to

```
lemma separation_for_sort:
    "∀ l. |l| <= 3 ⟶ is_sorted (sort l)"
```

which states that the depth parameter of the data separation theorem is in fact *exhausted* by the uniformity statements; this result is independent from the definition of sort and could be generated by HOL-TestGen together with the data separation theorem.

Altogether, we can now *verify* the regularity hypothesis. Without explaining the tactical Isabelle-commands in detail, we show the full straightforward induction proof:

```
lemma regularity_verified: "THYP (3 < |l| ⟶ is_sorted (sort l)) "
proof –
  have anchor:
    "⋀a l. |l| = 3 ⟹ is_sorted (ins a (sort l))"
    by(auto intro!: separation_for_sort [THEN spec, THEN mp]
                    is_sorted_invariant_ins )
  have step:
    "⋀a l. is_sorted (sort l) ⟹ is_sorted (ins a (sort l))"
    by(erule is_sorted_invariant_ins )
```

This introduces two sub-lemmas called “anchor” and “step,” establishing that for all lists of size three, the desired property holds and under the assumption that a sub-list is sorted, the desired property hold for arbitrary lists. As the names of theses two sub-lemmas suggest, they represent the anchor and the step of the main induction. In the following we turn to the proof of the main hypothesis:

```
show ?thesis
  apply(simp only: THYP_def)
```

The proof which results in:

1: $3 < |l| \longrightarrow$ is_sorted (sort l)

We continue the proof by induction l:

apply(induct l, auto)

resulting in:

1: $\bigwedge$a l. $[\![2 < |l|; \neg 3 < |l|]\!] \Longrightarrow$ is_sorted (ins a (sort l))
2: $\bigwedge$a l. $[\![2 < |l|;$ is_sorted (sort l)$]\!] \Longrightarrow$ is_sorted (ins a (sort l))

finally, we use the sub-lemmas "anchor" and "step" and conclude our proof:

apply(subgoal_tac "|l| = 3")
apply(auto elim!: anchor step)
done

Overall, this script follows the structure that can be expected in an informal proof sketch. Here, the lemma is_sorted_invariant_ins is just the invariant over the inner loop of the sorting algorithm:

lemma is_sorted_invariant_ins[rule_format]:
"is_sorted l $\longrightarrow$ is_sorted (ins a l)"

which is just established by another straightforward induction.

To complete the comparison, we briefly show the *direct proof* of the test specification:

lemma testspec_proven: "is_sorted (sort l)"
apply(induct l, simp_all)
apply(*erule* is_sorted_invariant_ins)
done

7 Conclusion

We have presented the verification of our concept of explicit test hypotheses as generated by our HOL-TestGen system at a small but paradigmatic example. It shows how tests and (post-hoc) verifications can work seamlessly together.

With respect to our three initial questions, we can give the following summary: Test hypotheses establish a logical link between individual test data and disjoint test cases. Test hypotheses can be seen as a kind of proof obligation that is proven in later stages of validation if needed. Test hypotheses can give the test engineer a further means to control the quality of a test, an experience that is well confirmed in several larger case studies [3,5] done with our system.

Uniformity is often criticized to be an unsound concept. But it is amazingly easy to be verified in a concrete situation, and plays the role of an induction anchor.

The good news is that testing test hypotheses can indeed be used to approximate verification—our methodology is therefore complete in this sense. The bad news is, that our example offers no hope for the desire to *use* tests to simplify proofs. We be-

lieve that our example proof stands here for a wide class of similar, e. g., recursively defined, problems: It can be expected that uniformity will *always* be established independently from a test, and regularity will boil down to an induction, where uniformity clauses are indeed relevant for establishing the anchor, but contribute nothing to the step.

References

[1] P. B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002.

[2] A. D. Brucker and B. Wolff. HOL-TestGen 1.0.0 user guide. Technical Report 482, ETH Zurich, Apr. 2005.

[3] 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, 2005.

[4] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, 2005.

[5] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *Test and Proof 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

[6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284, Heidelberg, Apr. 1993. Springer-Verlag.

[8] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, TAPSOFT *'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.