ETH Technical Report 449

Symbolic Test Case Generation for Primitive Recursive Functions

Achim D. Brucker* Burkhart Wolff†

June 1, 2004

Information Security
ETH-Zentrum
CH-8092 Zürich
Switzerland

We present a method for the automatic generation of test cases for HOL formulae containing primitive recursive predicates. These test cases may be used for the animation of specifications as well as for black-box-testing of external programs.

Our method is two-staged: first, the original formula is partitioned into test cases by transformation into a Horn-clause normal form (CNF). Second, the test cases are analyzed for ground instances satisfying the premises of the clauses. Particular emphasis is put on the control of test hypothesis' and test hierarchies to avoid intractability.

We applied our method to several examples, including AVL-trees and the red-black implementation in the standard library from SML/NJ.

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

*brucker@inf.ethz.ch
†bwolff@inf.ethz.ch

1 Introduction

Today, essentially two validation techniques for software are used: *software verification* and *software testing*. Whereas verification is rarely used in "real" software development, testing is widely-used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call *Djikstra's verdict* [10, p.6]:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

More recently, three research areas, albeit driven by different motivations, converge and result in a renewed interest in testing techniques:

- *Abstraction Techniques:* model-checking raised interest in techniques to abstract infinite to finite models. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [5, 9].

- *Systematic Testing:* the discussion over *test adequacy criteria* [23], i.e. criteria solving the question "when did we test enough to meet a given test hypothesis", led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [13, 11].

- *Specification Animation:* constructing counter-examples has raised interest also in the theorem proving community, since combined with animations of evaluations, they may help to find modeling errors early and to increase the overall productivity [22, 15, 8, 1].

The first two areas are motivated by the question "are we building the program right?", the latter is focused on the question "are we specifying the right program?". While the first area shows that Dijkstra's Verdict is no longer true under all circumstances, the latter area shows, that it simply does not apply in practically important situations. In particular, if a formal model of the environment of a software system (e.g. based among others on the operation system, middleware or external libraries) must be reverse-engineered, testing ("experimenting") is without alternative (see [7]).

Following standard terminology [23], our approach is a *specification-based unit test*. In general, a test procedure for such an approach can be divided into:

- *Test Case Generation:* for each operation of the pre/post-condition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

- *Test Data Selection:* for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

- *Test Execution:* the implementation is run with the selected test input data in order to determine the test output data.
- *Test Result Verification:* the pair of input/output data is checked against the specification of the test case.

This paper has been inspired by [12], which follows the line of specification animation works. In contrast, we see our contribution in the development of techniques mostly on the first and to a minor extent on the second phase. Building on QuickCheck [8], the work presented in [12] performs essentially random test, potentially improved by hand-programmed external test data generators. It is well-known that random test can be ineffective in many cases; in particular, if pre-conditions of a program based on recursive predicates like "input tree must be balanced" or "input must be a typable abstract syntax tree" rule out most of randomly generated data. In our approach, we will exploit these predicates and other specification data in order to produce adequate data. As a particular feature, the automated deduction-based process can log the underlying test hypothesis made during the test; provided that the test hypothesis are valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the specification.

We proceed as follows: we will introduce to our implementation (built on top of the theorem prover Isabelle/HOL [19]) by a tiny, but classical example (Sec. 2). This demonstration serves as a means to motivate concepts like *testing normal form*, *test cases*, *test statements*, *test animations* and *test scripts*. In Sec. 3, we will discuss the test case generation, in Sec. 4 the (structured) test statement generation. Finally, we apply our technique to a number of non-trivial examples (Sec. 5) involving recursive datatypes and recursive functions over them.

2 Symbolic Test Case Generation: A Guided Tour

Our running example for automatic test case generation is described as follows: given three integers representing the lengths of the sides of a triangle, a small algorithm has to check, whether these integers describe an equilateral, isosceles, scalene triangle, or no triangle at all. First we define an abstract datatype describing the possible results in Isabelle/HOL:

datatype Triangles := equilateral | scalene | isosceles | error

For clarity (and as an example for specification modularization) we define an auxiliary predicate deciding if the three lengths are describing an triangle:

$$\begin{aligned}\textbf{constdefs}\ & \mathit{triangle} :: [\mathrm{nat}, \mathrm{nat}, \mathrm{nat}] \to \mathrm{bool} \\ & \mathit{triangle}\ x\ y\ z \equiv (0 < x) \wedge (0 < y) \wedge (0 < z) \wedge (z < x + y) \\ & \qquad \wedge (x < y + z) \wedge (y < x + z)\end{aligned}$$

Now we define the behavior of the triangle program by initializing a Isabelle proof state with the test specification TS:

$$
\begin{aligned}
\texttt{prog}(x,y,z) = \ & \textbf{if } \textit{triangle } x\ y\ z \textbf{ then} \\
& \quad \textbf{if } x = y \textbf{ then} \\
& \quad\quad \textbf{if } y = z \textbf{ then } \text{equilateral } \textbf{else } \text{isosceles} \\
& \quad \textbf{else if } y = z \textbf{ then } \text{isosceles} \\
& \quad\quad \textbf{else if } x = z \textbf{ then } \text{isosceles } \textbf{else } \text{scalene} \\
& \textbf{else } \text{error}
\end{aligned}
$$

Note that the variable `prog` to is used to label an arbitrary implementation as the current *program under test* that should fulfill the specification.

In the following we show how test package `TestGen` can be applied for the automatic test data generation for the triangle problem. Our method proceeds in the following steps:

1. By applying `gen_test_case_tac` we bring the proof state into *testing normal form* (TNF). In this example, we decided to generate test-cases up to depth 0 (discussed later) and to unfold the *triangle* predicate by its definition (denoted by triangle_def):

   `gen_test_case_tac` 0 [triangle_def] "`prog`"

   In our example, this leads to a formulae with 26 clauses, among them:

   $$
   \begin{aligned}
   [\![0 < z; z < z + z]\!] &\Longrightarrow \texttt{prog}(z,z,z) = \text{equilateral} \\
   \left[\!\!\left[\begin{array}{c} x \neq z; 0 < x; 0 < z; \\ z < x + z; x < z + z; z < x + z \end{array} \right]\!\!\right] &\Longrightarrow \texttt{prog}(x,z,z) = \text{isosceles} \\
   [\![y \neq z; z \neq y; \neg z < z + y]\!] &\Longrightarrow \texttt{prog}(z,y,z) = \text{error} \\
   [\![y \neq z; z \neq y; \neg z < z + y]\!] &\Longrightarrow \texttt{prog}(z,y,z) = \text{error}
   \end{aligned}
   $$

   We call each Horn clause of the proof state a *symbolic test case*. The result of `gen_test_case_tac` is a test theorem describing the list of symbolic test cases. We extract the test theorem of the proof state and insert it into a specific data structure provided by `TestGen`, namely a *test environment* `env`, that captures all data relevant to a test:

   `val env` = ("`prog`", *topthm*()) `add_test_case` *mt_testenv*

   Here, *mt_testenv* is the empty test environment and `add_test_case` the (infix) insertion operation. The Isabelle operation *topthm*() extracts the current proof state as theorem of the form $[\![A_1; \ldots; A_{26}]\!] \Longrightarrow TS$ where the A_i abbreviate the above test cases.

   All symbolic test cases for the triangle example are presented in Tab. 1. The result is roughly equivalent to the one presented in [11] where a disjunctive normal form (DNF) was used. The only difference is, that we allow $0 \in \mathbb{N}$ as input whereas [11] allowed specified triangle over the natural numbers excluding 0.

1. Test Cases for equilateral triangle:

$$0 < z \rightarrow \texttt{prog}(z, z, z) = \text{equilateral}$$

2. Test Cases for isosceles triangle:

$$\begin{aligned}
[\![x \neq z; x < z + z; 0 < x]\!] &\Longrightarrow \texttt{prog}(x, z, z) = \text{isosceles} \\
[\![y \neq z; y < z + z; 0 < y]\!] &\Longrightarrow \texttt{prog}(z, y, z) = \text{isosceles} \\
[\![y \neq z; z < y + y; 0 < z]\!] &\Longrightarrow \texttt{prog}(y, y, z) = \text{isosceles}
\end{aligned}$$

3. Test Cases for scalene triangle:

$$[\![y \neq z; x \neq z; x \neq y; x < y + z; z < x + y; y < x + z]\!] \Longrightarrow \texttt{prog}(x, y, z) = \text{scalene}$$

4. Test Cases for triangle:

a) At least one input is equal to 0 (note, Dick and Faivre are using: $x, y, z \in \mathbb{N}_1$):

$$\begin{aligned}
z = 0 &\Longrightarrow \texttt{prog}(0, 0, 0) = \text{error} \\
[\![0 < x; z = 0]\!] &\Longrightarrow \texttt{prog}(x, 0, 0) = \text{error} \\
[\![0 < y; z = 0]\!] &\Longrightarrow \texttt{prog}(0, y, 0) = \text{error} \\
[\![0 < z; y = 0]\!] &\Longrightarrow \texttt{prog}(0, 0, z) = \text{error} \\
[\![0 < z; x = 0]\!] &\Longrightarrow \texttt{prog}(0, z, z) = \text{error} \\
[\![0 < z; y = 0]\!] &\Longrightarrow \texttt{prog}(z, 0, z) = \text{error} \\
[\![0 < y; z = 0]\!] &\Longrightarrow \texttt{prog}(y, y, 0) = \text{error} \\
[\![0 < z; y \neq z; 0 < y; x = 0]\!] &\Longrightarrow \texttt{prog}(0, y, z) = \text{error} \\
[\![x \neq z; 0 < z; 0 < x; y = 0]\!] &\Longrightarrow \texttt{prog}(x, 0, z) = \text{error} \\
[\![0 < x; 0 < y; x \neq y; z = 0]\!] &\Longrightarrow \texttt{prog}(x, y, 0) = \text{error}
\end{aligned}$$

b) Input not satisfying triangle property:

$$\begin{aligned}
[\![x \neq z; z + z \leq x]\!] &\Longrightarrow \texttt{prog}(x, z, z) = \text{error} \\
[\![y \neq z; z + z \leq y]\!] &\Longrightarrow \texttt{prog}(z, y, z) = \text{error} \\
[\![y \neq z; y + y \leq z]\!] &\Longrightarrow \texttt{prog}(y, y, z) = \text{error} \\
[\![y \neq z; x \neq z; x \neq y; x + y \leq z]\!] &\Longrightarrow \texttt{prog}(x, y, z) = \text{error} \\
[\![y \neq z; x \neq z; x \neq y; y + z \leq x]\!] &\Longrightarrow \texttt{prog}(x, y, z) = \text{error} \\
[\![y \neq z; x \neq z; x \neq y; x + z \leq y]\!] &\Longrightarrow \texttt{prog}(x, y, z) = \text{error}
\end{aligned}$$

Table 1: Test Cases for the Triangle Problem

2. Finally, we compute the concrete test data by grounding the symbolic test cases for "`prog`" in the environment by `genadd_test_data` (the parameter 3 controls the number of generated tests):

$$\texttt{val env} = \texttt{genadd\_test\_data}\ (\text{“}\texttt{prog}\text{”}, 3)\ \texttt{env};$$
$$\texttt{get\_test\_data}(\texttt{env}, \text{“}\texttt{prog}\text{”});$$

   The latter operation selects the generated data from the test environment and shows the list of the so-called *test statements* (excerpt):

$$\texttt{prog}(3,3,3) = \text{equilateral}$$
$$\texttt{prog}(4,6,0) = \text{error}$$

In the triangle example, standard simplification was able to eliminate the assumptions of the (grounded) test cases automatically. In general, assumptions in test statements (also called *constraints*) may remain. Provided that all test statements are executable, clauses with constraints can nevertheless be interpreted as an abstract test program. For its result, three cases may be distinguished: If one of the clauses evaluates to false, the test is *invalid*, otherwise *valid*. A valid test may be *successful*, iff the evaluation of all conclusions (including the call of `prog`) also evaluates to true; otherwise the test contains at least one *failure*. Rephrased in this terminology, the ultimate goal of the test data selection is to construct successful tests, which means that ground substitutions must be found that make the remaining constraints valid.

Coming back to our example, there is a viable alternative for the proceeding above: instead of unfolding *triangle* and trying to generate ground substitutions satisfying the constraints, one may keep *triangle* in the test theorem, treating it as a building block for new constraints. It turns out that an own test theorem and test data (like "$\mathit{triangle}(3,4,5) = \text{True}$") can be generated "once and for all" and inserted before the test data selection phase. Thus, the main state explosion (leading to 26 test cases for this tiny example) is shifted from the test case generation to the test data selection step. We will discuss this approach to modularized test data generation in Section 4.

Having generated a sequence of test statements, the question remains if it is executable, i.e. if the underlying Isabelle code-generator [4] can convert it to SML code. This question boils down to a purely syntactic one: a specification is *executable* iff all constants (except the name the program under test) occurring in the test statements must be either constructors or functions for which primitive or well-founded recursive definitions are available; thus, the Hilbert-operator or unbounded quantifiers are excluded in the test statements.

Further, for executable test statements the `TestGen`-module provides additional methods for:

- *Test Harness Generation:* the `gen_testscript` method generates an SML-based test script where the name of the program under test is mapped to external procedure calls. This allows for testing real implementations. Programs written in other

languages (e.g. Java or C) can be tested by linking them as "external functions" to the SML environment. Our test harness generator is mainly a specific setup of the Isabelle code-generator.

- *Animation:* the `animate` method allows for the evaluation of the test statements in a test environment on a stepwise basis. This may be an alternative way to debug definitions early.

3 Concepts of Test Case Generation

As input of the test case generation phase, namely the test specification, one might expect a special format like pre(x) $\rightarrow$ post x (`prog`(x)). However, this would rule out trivial instances such as $3 <$ `prog`(x) or just `prog`(x) (meaning that `prog` must evaluate to True for x). Therefore, we do not impose any other restriction on a specification other than being executable.

Processing this test specification, our method `gen_test_case_tac` can be separated into the following conceptual phases (in reality, these phases were performed in an interleaved way):

- *Tableaux Normal Form Computation:* via a tableaux calculus (see Tab. 2), the current specification is transformed into Clause Normal Form (CNF).
- *Rewriting Normal Form Computation:* via the standard rewrite rules the current specification is simplified.
- *TNF Computation:* by re-ordering of the clauses, the calls of the program under test are re-arranged such that they only occur in the conclusion, where they must occur at least once.
- *TNF Minimization:* redundancies, e.g. clauses subsumed by others are eliminated.
- *Exploiting Uniformity Hypotheses:* for free variables occurring in recurring argument positions of primitive recursive predicates, a *suitable data separation lemma* is generated and applied (leading to a test hypothesis labeled by *THYP*).
- *Exploiting Regularity Hypothesis:* for all Horn-clauses not representing a test-hypothesis, a uniformity hypothesis is generated and exploited.

After a brief introduction into concepts and use of Isabelle in our setting, we will follow the sequence of these phases and describe them in more detail in the subsequent sections. We will conclude with a discussion of coverage criteria.

3.1 Concepts and Use of Isabelle/HOL

Isabelle [19] is a well-known generic theorem prover system of the LCF prover family; as such, we use the possibility to build on top of the logical core engine own programs performing symbolic computations over formulae in a logically safe (conservative) way:

this is what `TestGen` technically is. Throughout this paper, we will use Isabelle/HOL, the instance for Church's higher-order logic. Isabelle/HOL offers support for data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

Isabelle's proof engine is geared towards Horn clauses (called "subgoals"): $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, written $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$, is viewed as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}". A *proof state* in Isabelle contains an implicitly conjoint sequence of Horn-clauses ϕ_1,...,ϕ_n and a *goal* ϕ. Since a Horn-clause:

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

is logically equivalent to

$$\neg A_1 \vee \cdots \vee \neg A_n \vee A_{n+1},$$

a Horn-clause normal form is a conjunctive normal form. Note, that in order to cope with quantifiers naturally occurring in specifications, we generalize the idea of a Horn-clause to Isabelle's format of a *subgoal*, where variables may be bound by a built-in meta-quantifier:

$$\bigwedge x_1, \ldots, x_m.[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

Subgoals and goals may be extracted via *topthm*() (see previous section) to the theorem $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$; this mechanism is used to generate test theorems. The meta-quantifier $\bigwedge$ is used to capture the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as free variables. Further, Isabelle supports meta-variables (written $?x, ?y, \ldots$), which can be seen as "holes in a term" that can still be substituted. Meta-variables were instantiated by Isabelle's built-in higher-order unification.

3.2 Normal Form Computations

In this section, we describe the tableaux, rewriting and testing normal form computations in more detail. In Isabelle/HOL, the automated proof procedures for HOL formulae heavily depend on tableaux calculi [14] presented as (derived) natural deduction rules. The core tableaux calculus is shown in Tab. 2 in the Appendix. Note, that with the notable exception of the elimination rule for the universal quantifier (see Tab. 2(c)), any rule application leads to a logically equivalent proof state: therefore, all rules (except $\forall$ elimination) are called *safe*. When applied bottom up in backwards reasoning (which may introduce meta-variables explicitly marked in Tab. 2), the technique leads in a deterministic manner to a Horn-clause normal form.

Horn-clauses can be normalized by a number of elementary logical rules (such as False $\Longrightarrow P =$ True), the usual injectivity and distinctness rules for constructors implied by datatypes and computation rules resulting from recursive definitions. Both processes together bring an original specification into Rewriting Horn-clause normal form.[1]

[1] Both processes were covered by the standard `Safe_tac` procedure.

However, these forms do not exclude clauses of the form:

$$[\![\neg(\texttt{prog}\ x = c); \neg(\texttt{prog}\ x = d)]\!] \Longrightarrow A_{n+1}$$

where `prog` is the program under test. However, this clause can be transformed equivalently into:

$$[\![\neg(A_{n+1})]\!] \Longrightarrow \texttt{prog}\ x = c \vee \texttt{prog}\ x = d$$

We call this form of Horn-clauses *testing normal form* (TNF). More formally, a Horn-clause is in TNF (F) for program under test F, iff:

- F does not occur in the constraints,
- F does occur in the conclusion.

Note that not all specifications can be converted to TNF. For example, if the specification does not make a suitably strong constraint over program F, in particular if F does not occur in the specification. In such cases, `gen_test_case_tac` stops with an exception.

3.3 Minimizing TNF

A TNF computation as described so far may result in a proof state with several redundancies. Redundancies in a proof state may result in superfluous test data and should therefore be eliminated. A proof state may have:

1. several occurrences of the identical clauses
2. several occurrences of clauses with subsuming assumption lists; this can be eliminated by the transformation:

$$\frac{[\![P; R]\!] \Longrightarrow A; \quad [\![P; Q; R]\!] \Longrightarrow A;}{[\![P; R]\!] \Longrightarrow A;}$$

3. and in particular, clauses that subsume each other after distribution of $\vee$; this can be eliminated by the transformation:

$$\frac{[\![P; R]\!] \Longrightarrow A; \quad [\![\neg P; Q]\!] \Longrightarrow B; \quad [\![R; Q]\!] \Longrightarrow A \vee B;}{[\![P; R]\!] \Longrightarrow A; \quad [\![\neg P; Q]\!] \Longrightarrow B;}$$

The notation above refers to logical transformations on a subset of clauses within a proof state and not, as usual, on formulae within a clause. Since in backward proof the proof state below is a refinement of the proof state above, the logical implication goes from bottom to top.

We implemented a couple of new tactics that order premises and meta-quantifiers in a clause and erase subsumed clauses. Amazingly, Isabelle 2003 provides only for some rudimentary cases a standard tactic; our normalizations of proof states are therefore reusable for other logics than HOL, too.

3.4 Exploiting Uniformity Hypothesis for Recursive Predicates

In the following, we address the key problem of test case generation in our setting, i.e. recursive predicates occurring in preconditions of a program. As a introductory example, we consider the membership predicate of an element in a list:

$$\begin{array}{lll} \textbf{primrec} & x\ mem\ [] & = \text{False} \\ & x\ mem\ (y\#ys) & = \textbf{if}\ y = x\ \textbf{then}\ \text{True}\ \textbf{else}\ x\ mem\ ys \end{array} \tag{1}$$

which occurs as precondition in an (abstract) program specification:

$$[\![x\ mem\ S]\!] \Longrightarrow \texttt{prog}\ x\ S$$

Gaudel suggested in [13] for the testing of recursive data structures the introduction of a *uniformity hypothesis* as one possible form of a test hypothesis in our sense, a kind of weak induction rule:

$$\frac{\begin{array}{c}[|x| < k] \\ \vdots \\ P\ x\end{array}}{P\ x}$$

This rule formalizes the hypothesis that, provided a predicate P is true for all data x whose *size*, denoted by $|x|$, is less then a given depth k, it is always true. The original rule can be viewed as a meta-notation: In a rule for a concrete data-type, the premises $|x| < k$ can be expanded to a number of premises enumerating constructor-terms. Using uniformity hypothesis' is related to bounded model-checking [5].

For all variables in Horn-clauses that occur as (recurring) arguments of primitive recursive functions, we will use a testing hypothesis of this kind — called *data separation lemma* — in a little exercise in poly-typic theorem proving [21] described in the following.

The Isabelle/HOL datatype package generates from datatype definitions a couple of definitions of poly-typic functions (like case-match and recursors) and derives a number of theorems over them (like induction, distinctness of constructors, etc.). In particular, for any datatype, we can assume the size function and reduction rules allowing to compute $\big|[a,b,c]\big| = 3$, for example. Moreover, there is an *exhaustion*-theorem, which for example has for lists the form:

$$[\![y = [] \Longrightarrow P; \bigwedge x\ xs.y = x\#xs \Longrightarrow P]\!] \Longrightarrow P$$

Now, since we can separate any data x belonging to a datatype τ into:

$$x \in \{z :: \tau.|z| < d\} \vee x \in \{z :: \tau.d \leq |z|\}$$

i.e. x is either in the set of data smaller d or in the remaining set. Note that both sets are infinite in general; the bound for the size produces "data test cases" and not just finite sets of data. Consequently, we can derive for each given type τ and each d a destruction rule that enumerates the data of size $0, 1, \ldots, k-1$. For lists and $d = 2, 3$, it has the form:

$$\begin{array}{l} x : \{z :: \alpha\ \text{list}.|z| < 2\} \rightarrow x = [] \vee (\exists a.x = [a]) \\ x : \{z :: \alpha\ \text{list}.|z| < 3\} \rightarrow x = [] \vee (\exists a.x = [a]) \vee (\exists ab.x = [a,b]) \end{array}$$

Putting the lemma3.4 together with the destruction rule for $d = 2$, instead of the unsafe uniformity hypothesis in the sense of Gaudel we get the (safe) data separation lemma of the form:

$$\frac{\begin{array}{ccccc}[x = []] & & [x = [a]] & & [x = [a,b]] \\ \vdots & & \vdots & & \vdots \\ P(x) & \bigwedge a. & P(x) & \bigwedge ab. & P(x) \end{array} \qquad THYP(2 \leq |x| \rightarrow P(x))}{P(x)}$$

The purpose of this rule in backward proof is to split a statement over a program into several cases, each with an additional assumption that allows to "rewrite-away" the x appropriately. Here, the constant $THYP :: \text{bool} \rightarrow \text{bool}$ (defined as the identity function) is used to label the test hypothesis in the proof state. Since we do not unfold it, formulas labeled by $THYP$ are protected from decomposition by the tableaux rules.

The introduced equalities by this rule of depth $d = 2$ allows for the simplification of the primitive recursive predicate *mem* which leads to further decompositions during the TNF computation. Thus, for our little example:

$$x \ mem \ S \rightarrow \texttt{prog} \ x \ S$$

executing `gen_test_case_tac 3 []` "`prog`" results in the following TNF (`prog`):

1. $\texttt{prog} \ x \ [x]$
2. $\bigwedge b.\texttt{prog} \ x \ [x, b]$
3. $\bigwedge a.a \neq x \rightarrow \texttt{prog} \ x \ [a, x]$
4. $THYP(3 \leq |S| \rightarrow x \ mem \ S \rightarrow \texttt{prog} \ x \ S)$

The simplification of the *mem* predicate along its defining rules (1) leads to nested "**if then else**" constructs. Their decomposition during CNF computation results in the constraint that the lists fulfilling the precondition must have a particular structure. Even the simplest "generate-and-test"-method for test data selection will now produce adequate test statements, while it would have produced mostly test failures when applied directly to the original specification!

The handling of quantifiers ranging over datatypes can be done analogously: since $\forall x.P(x)$ is equivalent to $\forall x : UNIV.P(x)$ and since $UNIV = \{z :: \tau.|z| < d\} \cup \{z :: \tau.d \leq |z|\}$, the universal quantifier can be decomposed in a finite conjunction for the test cases smaller d and a test hypothesis $THYP$ for the rest.

From the above example it follows that the general form of a test theorem (to be extracted by $topthm()$) is $[\![A_1; \ldots; A_n; THYP(H_1); \ldots; THYP(H_m)]\!] \Longrightarrow TS$. Here the A_i represent the test cases, the H_i the test hypothesis and TS the testing specification.

3.5 Exploiting Regularity Hypothesis

After introducing the regularity hypothesis and computing a TNF (except for clauses containing $THYP$'s), we use the Horn-clauses to construct another form of testing hypothesis,

namely the *regularity hypothesis* [13] (sometimes also called *partitioning hypothesis*) for each test case. This kind of hypothesis has the form:

$$THYP(\exists x.P(x) \rightarrow \forall x.P(x))$$

This means 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.

Using a uniformity hypothesis for each (non-*THYP*) clause allows for the replacement of free variables by meta-variables; e.g. for the case of two free variables, we have the following transformation on proof states:

$$\frac{[\![A_1\ x\ y; \ldots; A_n\ x\ y]\!] \Longrightarrow A_{n+1}\ x\ y}{[\![A_1\ ?x\ ?y; \ldots; A_n\ ?x\ ?y]\!] \Longrightarrow A_{n+1} ?x\ ?y; \quad THYP((\exists xy.P\ x\ y) \rightarrow (\forall xy.P\ x\ y))};$$

where $P\ x\ y \equiv A_1\ x\ y \wedge \ldots \wedge A_n\ x\ y \rightarrow A_{n+1}\ x\ y$. This transformation is logically sound.[2] Moreover, the construction introduces individual meta-variables in each clause for the ground instances to be substituted in the test data selection; this representation also paves the way for structured test data reuse (see Sec.4).

3.6 Coverage Criteria: A Discussion

In their seminal work Dick and Faivre [11] propose to transform the original specification into disjunctive normal form (DNF), followed by a case splitting phase converting the disjunctions $A \vee B$ into $A \wedge B$, $\neg A \wedge B$ and $A \wedge \neg B$ and further (logical and arithmetic) simplifications and minimizations on the disjunctions. The resulting cases are also called the *partitions of the specification* or the *(DNF) test cases*. The method suggests the following test adequacy criterion: a set of test data is *partition complete*, iff for any test case there is a test data. Consequently, a program P is tested adequately to partition completeness with respect to a specification S, if it passes a partition complete test data set.

Our notion of *successful test*, see Sec. 2, is a CNF based adequacy criterion; similar criterion are also used e.g. in [16]. DNF and CNF based adequacy result in the same partitioning in many practical cases, as in the triangle-example, while having no clear-cut advantage in others. Since the DNF technique has the disadvantage to produce a double exponential blow-up (the case splitting phase can produce an exponential blow-up in it its own) while CNF computation is simply exponential, and since DNF-computation can be more directly and efficiently implemented in the Isabelle proof engine, we chose the latter.

Interestingly, CNF adequacy subsumes another interesting adequacy criterion under certain conditions, namely *path coverage* with respect to the specification. Path coverage means that in any (mutual) recursive system of functions, all reachable paths, e.g. of the **if** P **then** A **else** B statements, were activated at least once. For a mutual recursive system consisting only of *primitive* recursive functions, (i.e. with each call the size of data will decrease exactly by one), it can be concluded that if the testing depth d is

[2] It consists of `subgoals_tac` and a number of steps re-arranging the quantifiers.

chosen larger than the size of the maximal strong component of the call graph of the recursive system, each function is unfolded at least once. Since the unfolds results in conditionals that were translated to $(P \rightarrow A) \wedge (\neg P \rightarrow B)$, any branch will lead to a test case.

Thus, while `gen_test_case_tac` often produces reasonable results for arbitrarily recursive functions, only for primitive recursions we can assure that the underlying CNF adequacy of our method subsumes path coverage.

4 Structured Test Data Selection

The motivations to separate test data selection from test case generation are both conceptual and technical. Conceptually, test data selection is a process where we would like to admit also more heuristic techniques like random data generation or generate-and-test with the constraints. Test data selection yields sequences of ground theorems (no meta-variables, no type variables); this paves the way for more efficient inference techniques such as evaluation by code generation desperately needed in the unavoidable blow-up in the late stages. Last but not least, there is a purely technical reason to separate test cases from test data statements: since both were represented by the Isabelle data structure *thm*, and since the scope of polymorphic variables is limited to a *thm*, a test case can not be further refined to a conjunction with the type variables differently instantiated.

The generation of a multitude of ground *test statements* from one *test theorem* containing the test cases and the test hypothesis is essentially based on a random-procedure followed by a test of the satisfaction of the constraints (thus: generate and test). For each type, this default procedure may be overwritten in `TestGen` specific generators that may be user defined; thus, the usual heuristics like trying $[0, 1, 2, maxint, maxint + 1]$ can be easily implemented, or the counter-example generation of integrated in Isabelle's arithmetic procedure can be plugged in (which is difficult to control in larger examples in our experience).

Now we will discuss the issue of structured test data generation. Similar to theorem proving, the question of "how many definitions should be unfolded" is crucial: exploiting the structure and organization of a specification is the major weapon against complexity. Our approach of structured test data generation is best described at our running triangle-example. It contains a auxiliary predicate *triangle*; our previous attempt shown in Sec. 2 to generate a test theorem was based on simply unfolding it. The test case generation resulted in 31 cases, which was acceptable since the test theorem had still a reasonable size and since the output was acceptable for an automated test procedure. If we do not unfold it, i.e. just call:

`gen_test_case_tac` 2 [] "`prog`"

the resulting test theorem had only 6 test cases. Thus, a substantial part of the blow-up could be shifted into the test data selection, if we could generate a test theorem for *triangle* as such, generate the test data separately and feed test statements for it into the process for the global computation. The trick can be done as follows: we define a

trivially true proof goal for:

$$\texttt{prog}(x, y, z) = \mathit{triangle}\ x\ y\ z \Longrightarrow \texttt{prog}(x, y, z) = \mathit{triangle}\ x\ y\ z$$

unfold *triangle* and compute TNF(`prog`). When folding back *triangle* via the assumption we get the following test cases:

$$\begin{array}{ll} \neg 0 < x \Longrightarrow \neg \mathit{triangle}\ x\ y\ z & \neg z < x + y \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\ \neg 0 < y \Longrightarrow \neg \mathit{triangle}\ x\ y\ z & \neg x < y + z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\ \neg 0 < z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z & \neg y < x + z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \end{array}$$

$$\left[\!\!\left[\begin{array}{c} 0 < x; 0 < y; 0 < z; \\ z < x + y; x < y + z; y < x + z \end{array} \right]\!\!\right] \Longrightarrow \mathit{triangle}\ x\ y\ z$$

which can be inserted into the current test environment. The resulting test data statements (e.g. $\neg \mathit{triangle}\ 0\ 3\ 5$) can now be resolved directly into the global test theorem for `prog`. Thus, the test cases of previously developed theories can be reused when building up larger units. Of course, when building up test data in a modular way, this comes at a price: since the "local test theorems" are more abstract and have not the same logical information available as their application context in a more global test theorem, the instantiation may result in unsatisfiable constraints. Nevertheless, since the criterion for success of a decomposition is clear — at the very end we want constraint-free test statements achieving a full coverage of the TNF— the implementor of a test has several choices here that can lead to success even in larger problems. In our example, there is no loss at all: the auxiliary predicate is just the right abstraction for the whole test data generation: test data for the local predicate is valid for the global goal, and by construction, the set of test statements is still partition complete.

5 Examples

In this section we will demonstrate `TestGen` on two widely used variants of balanced binary search trees: AVL trees and red-black trees. These case studies were performed using Isabelle 2003 compiled with SML of New Jersey running on Linux with 512 MByte of RAM and an Intel P4 processor with 1.6 GHz.

5.1 AVL Trees

In 1962 Adel'son-Vel'skiĭ and Landis [2] introduced a class of balanced binary search trees that guarantee that a tree with n internal nodes has height $O(\log n)$. This is done by ensuring the invariant that for every node the height of the subtrees differs at most by one. For ensuring this invariant the trees are rebalanced after each insertion by so called rotations. After an insertion $\Theta(\log n)$ rotations may be needed to assure the balancing invariant.

We used an AVL-theory from the Isabelle library to test the following invariant: the insert functions really inserts an element in a given tree. To be more precise: if an

element y is in the tree after insertion of x in the tree t then either $x = y$ holds or y was already stored in t. This can be stated as follows:

$$isin\ y\ (insert\ x\ t) = (y = x \vee isin\ y\ t)$$

Based on the depth $d = 3$, this test specification leads to amazing 236 test cases which were computed in less than 30 seconds.

5.2 Red-Black Trees

Another idea for balanced binary search trees was presented by Bayer [3]. In this data structure, the balancing information is stored in one additional bit per node. This is called "color of a node" (which can either be red or black). A valid Red-Black tree assumes the following properties:

- *Red Invariant:* each red node has a black parent.
- *Black Invariant:* each path from the root to an empty node has the same number of black nodes.

For our case study we used a formalization of the red-black trees of the standard library of SML of New Jersey (based on version 110.44) presented in [18] and used the generated test data to test the real implementation. The implementation of Red-Black trees is mainly based on the following datatype declaration:

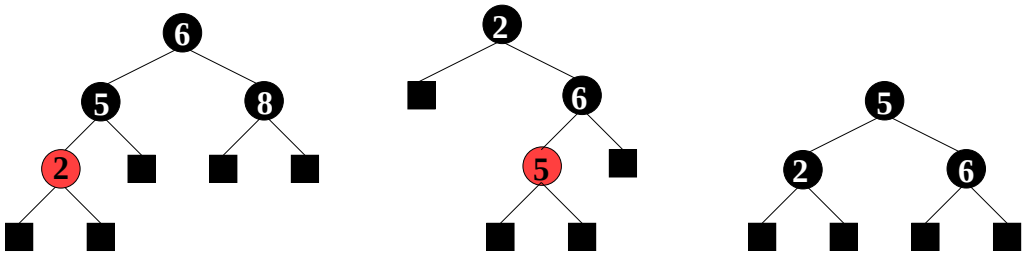
datatype color $= R \mid B$
α tree $= E \mid T$ color (α tree) (α item) (α tree)

In this example we have chosen not only to check if keys are stored or deleted correctly in the trees but also to check if the trees fulfill the balancing invariants. We formalize the red and black invariant by recursive predicates:

consts
redinv :: (α item) tree $\Rightarrow$ bool
blackinv:: (α item) tree $\Rightarrow$ bool

recdef *redinv* "measure ($\lambda t.$(size t))"
"*redinv E* = True"
"*redinv* (*T B a y b*) = (*redinv a* $\wedge$ *redinv b*)"
"*redinv* (*T R* (*T R a x b*) *y c*) = False"
"*redinv* (*T R a x* (*T R b y c*)) = False"
"*redinv* (*T R a x b*) = (*redinv a* $\wedge$ *redinv b*)"

recdef *blackinv* "measure ($\lambda t.$(size t))"
"*blackinvE* = True"
"*blackinv*(*T color a y b*) = ((*blackinv a*) $\wedge$ (*blackinv b*)
$\wedge$((*max_B_height a*) = (*max_B_height b*)))"



(a) pre-state (b) result of SML/NJ (c) correct result

Figure 1: Test Data for deleting a node in a Red-Black tree

As test specification we state:

$$(isin\ x\ t \wedge redinv\ t \wedge blackinv\ t) \rightarrow (redinv\ (delete\ x\ t) \wedge blackinv\ (delete\ x\ t))$$

In other words, for all trees the deletion operation maintains the red and black invariant. The test case generation takes less than two minutes and results in 348 test cases. Test data instantiating *item* with Integers revealed a major error in the standard library of SML of New Jersey. Assuming the red-black tree presented in Fig. 1(a), deleting the node with value 8 results in the tree presented in Fig. 1(b) which obviously violates the black invariant (the expected result is the balanced tree in Fig. 1(c)).

This example shows that specification based testing can test invariants crucial for efficiency; violations against them are hard to detect otherwise. Combinations of insert- and delete operations of the SML implementation lead easily to trees that degenerate to (still sorted) lists. In our case, the revealed flaw has not been detected in the last 12 years, although used, e.g, for implementing finite sets.

6 Conclusion

We have presented the theory and implementation of a test data generator for unit tests of functional programs. It allows for exploiting the structure of specifications both on the level of theories or testing theorems helping to control the case explosion inherent to testing. In our view, test data generation is an activity that clearly needs user interaction and that is therefore best supported by an integration into *interactive* theorem proving environment such as Isabelle. Our structure exploiting approach offers also the advantage to be applicable to regression testing.

The underlying rationale of our test case generation approach is to separate a specification P into a part P_{fin} which can be shown by a finite evaluation and a test hypothesis part $\mathit{THYP}(P_{\text{fin}} \rightarrow P)$. In this view, "The Art of Formal Testing" consists in finding tractable methods for this separation, minimizing both the number of test cases, test hypothesis and introducing as weak test hypothesis as possible. The kind of test hy-

pothesis admissible for a test case generation is of course a crucial point in any formally founded systematic test method.

As a hidden agenda of this paper, we believe that there is another line of criticism against Dijkstra's verdict. A successful test together with explicitly stated test hypothesis is not fundamentally different from any other scientific discipline, in particular program verification itself: here, also all sorts of modeling assumptions were made over the target machine (deterministic), the programming language (correctly implemented), the verification logics (consistent) and its implementation (the prover is correct). The nature and trustworthiness of these assumptions may be different, but a clear-cut line between testing and verification does not exist.

6.1 Further Work

We see the following lines of extension of our work:

1. *investigating test hypothesis*: new test hypothesis (like congruence hypothesis on data, for example) may dramatically improve the viability of the approach. Furthermore, it should be explored if the verification of test hypothesis for a given abstract program offers new lines of automation.

2. *better control of the process*: at the moment, our implementation can only be controlled by very globally applied parameters such as depth. The approach could be improved by generating the test hypothesis and the test data depending from the local context within the test theorems.

3. *integration tests*: integrating/combining our framework into behavioral modeling leads to the generation of *test sequences* as in [17, 20],

4. *generating test data for many-valued logics* such as HOL-OCL [6] should make our approach applicable to formal methods more accepted in industry.

References

[1] Programatica: Integrating programming, properties, and validation. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[4] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *TYPES 2000*, volume 2277 of *LNCS*, pages 24–40. Springer-Verlag, 2002.

[5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking.* Number 58 in Advances In Computers. 2003.

[6] A. D. Brucker and B. Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In C. Muñoz, S. Tahar, and V. Carreño, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.

[7] A. D. Brucker and B. Wolff. A case study of a formalized security architecture. In *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier Science Publishers, 2003.

[8] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[10] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing.* Academic Press, London, 3 edition, 1972.

[11] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *FME 93*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, 1993.

[12] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 188–203. Springer-Verlag, Rome, Italy, 2003.

[13] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *LNCS*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.

[14] R. Hähnle. Tableaux for many-valued logics. In M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 529–580. Kluwer, Dordrecht, 1999.

[15] S. Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.

[16] R. M. Hierons. Testing from a z specification. *The Journal of Software Testing, Verification, and Reliability*, 1(7), 1997.

[17] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - a tool for distributed systems specification. In *FTRTFT 96*, volume 1135 of *LNCS*, pages 467–470. Springer-Verlag, 1996.

[18] A. Kimmig. Red-black trees of smlnj. Studienarbeit, Universität Freiburg, Freiburg, 2003.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

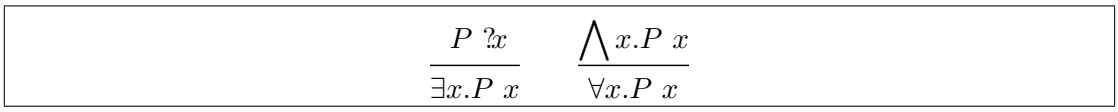
[20] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In E. Brinksma and J. Tretmans, editors, *Proc. Formal approaches to testing of software*, pages 47–60. BRICS, 2001.

[21] K. Slind and J. Hurd. Applications of polytypism in theorem proving. In D. Basin and B. Wolff, editors, *TPHOLs*, number 2758 in LNCS, pages 103–119. Springer-Verlag, Rome, Italy, 2003.

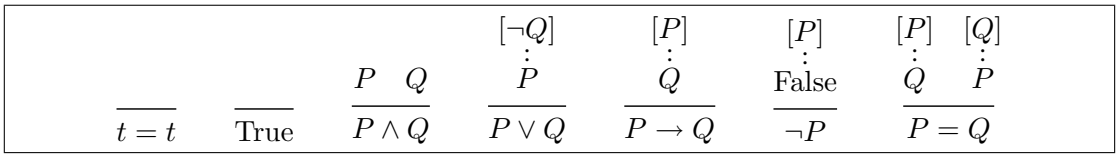
[22] The ZETA system, 2001. `http://uebb.cs.tu-berlin.de/zeta/`.

[23] H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

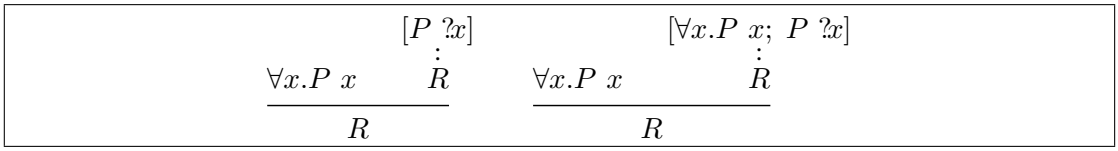
Appendix



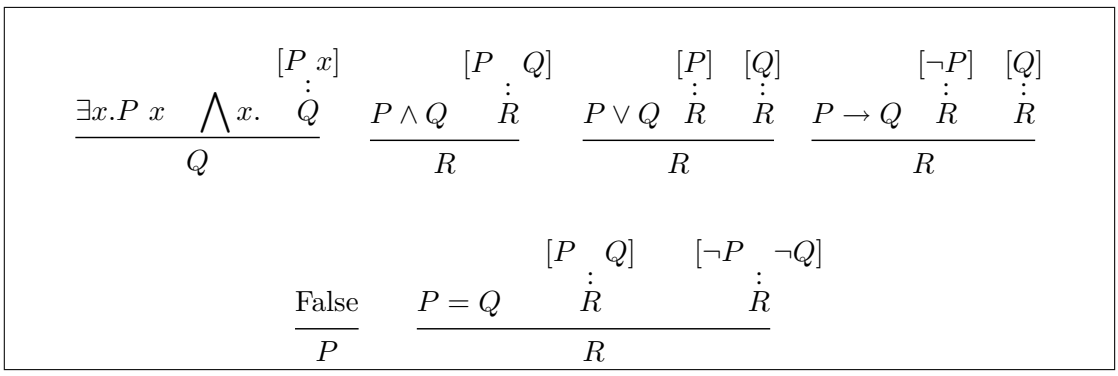
(a) quantifier introduction rules



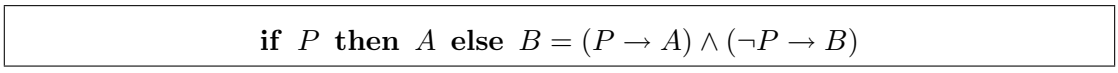
(b) safe introduction rules



(c) unsafe elimination rules



(d) safe elimination rules



(e) rewrites

Table 2: The Standard Tableaux Calculus for HOL