# Random testing in Isabelle/HOL

Stefan Berghofer and Tobias Nipkow
Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
`http://www.in.tum.de/~{berghofe,nipkow}/`

## Abstract

*When developing non-trivial formalizations in a theorem prover, a considerable amount of time is devoted to "debugging" specifications and conjectures by failed proof attempts. To detect such problems early in the proof and save development time, we have extended the Isabelle theorem prover with a tool for testing specifications by evaluating propositions under an assignment of random values to free variables. Distribution of the test data is optimized via mutation testing. The technical contributions are an extension of earlier work with inductive definitions and a generic method for randomly generating elements of recursive datatypes.*

## 1. Introduction

When developing non-trivial formalizations in a theorem prover, a considerable amount of time is devoted to "debugging" specifications and theorems. Typically, incorrect specifications or theorems are discovered during failed proof attempts. This is an expensive form of debugging. Therefore it is often useful to *test* conjectures before embarking on a proof. A possible way of doing this is to assign random values to the free variables of the conjecture and then evaluate it. This approach has already been successfully used in the functional programming community and is implemented e.g. in the *QuickCheck* library [3] for testing Haskell programs. We describe an implementation of such a testing tool for the theorem prover Isabelle/HOL. It is important to note that QuickCheck is essentially a framework for writing random test case generators, where the implementation of generators for specific datatypes is left to the user. In contrast, our testing tool automatically derives test case generators from datatype definitions in a canonical way, using a technique reminiscent of *polytypic programming* [8, 12].

Roughly speaking, Isabelle/HOL is a functional programming language augmented with predicate logic. For example, we can define inductive datatypes such as the datatype of lists

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

and define recursive functions such as *take* and *drop*:

**consts** *take*:: *nat* ⇒ *'a list* ⇒ *'a list*
**primrec**
  *take n* [] = []
  *take n* (*x* # *xs*) = (*case n of*
    *0* ⇒ [] | *Suc m* ⇒ *x* # *take m xs*)

**consts** *drop*:: *nat* ⇒ *'a list* ⇒ *'a list*
**primrec**
  *drop n* [] = []
  *drop n* (*x* # *xs*) = (*case n of*
    *0* ⇒ *x* # *xs* | *Suc m* ⇒ *drop m xs*)

We can now try to formulate a commutation property for *take* and *drop*:

**theorem** *take j* (*drop i xs*) = *drop i* (*take j xs*)
  **quickcheck**

Before attempting to prove such a statement, it is a good idea to run a counterexample generator on it. This is done using the **quickcheck** command of Isabelle shown above, which produces the following output:

```
Test data size: 0
Test data size: 1
Test data size: 2

Counterexample found:
```
*i* = *Suc 0*
*j* = *Suc 0*
*xs* = [*1*, −*1*]

This shows that our above statement was wrong, since

*take* (*Suc 0*) (*drop* (*Suc 0*) [*1*, −*1*]) = [−*1*]  and
*drop* (*Suc 0*) (*take* (*Suc 0*) [*1*, −*1*]) = []

Fortunately, this error can easily be corrected. Thus, we abort our failed proof attempt and prove a slightly modified version of the above statement:

**theorem** $\bigwedge i\, j.\ take\ j\ (drop\ i\ xs) = drop\ i\ (take\ (i+j)\ xs)$

## 2. Related work

Testing is a huge field, even when limited to formal specifications. Even model checking can be viewed as a clever form of exhaustive testing. In the theorem proving field testing has long had a bad name — after all, isn't testing the very thing theorem proving is trying to replace? Nevertheless, the idea of searching for finite (counter)models of first-order formulae has been around for some time (e.g. [11, 14]). Thus our work should be viewed as a new application of mostly known techniques. In particular, we have made use of the following ideas:

- Executing HOL specifications, which we described earlier [1], and which is reminiscent of functional-logic programming [6].
- Random testing *à la QuickCheck*, which we lift from the purely functional to the functional-logic level.
- Mutation testing to determine suitable parameters of our test framework.

An approach closely related to ours has been proposed by Dybjer, Qiao and Takeyama [5] in the context of the *Agda* proof assistant, which implements Martin-Löf type theory and can roughly be viewed as an extension of Haskell with dependent types. Their work uses ideas from the *QuickCheck* library, too, but is restricted to recursive functions, while our approach covers inductively defined predicates as well. Moreover, in their framework, test data generators are defined and executed *inside* the language of Agda, rather than programming them in Haskell. On the one hand, this has the advantage that properties of test case generators can be *proved* inside the system, but on the other hand has the possible drawback of slower execution speed when applied to larger specifications.

A very influential tool for debugging formal specifications is the Alloy Analyzer [7] which enumerates small finite models of the specification to find counterexamples to given conjectures. Essentially, Alloy specifications are first-order formulae and the search for finite models is performed by an external SAT solver. Tjark Weber has carried this idea over to HOL [13]. The two forms of counter-example search are quite complementary: the *QuickCheck* approach is limited to executable formulae but is not very sensitive to the size of the specification. Searching for finite models can handle non-executable constructs like quantifiers but is very sensitive to the size and complexity of the formulae involved.

The fact that computers have made testing much easier than proving has not escaped mathematicians either. In 1992 the journal *Experimental Mathematics* was founded to allow the publication of conjectures formulated on the basis of experiments, i.e. testing.

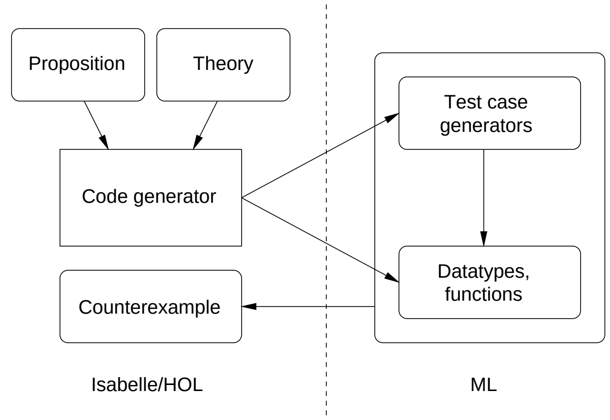


**Figure 1. Architecture of testing framework**

## 3. Overview

Testing specifications involves the evaluation of expressions. In principle, this could be done using Isabelle's built-in term rewriting engine, the so-called *simplifier*. However, this would involve the *construction* of a proof in equational logic, which is too slow for processing large amounts of test cases. Since a large subset of Isabelle/HOL specifications is actually *executable*, the approach taken in this work is therefore to *compile* specifications to functional programs in the ML programming language that are efficiently executable. The executable fragment of Isabelle/HOL contains the following constructs:

- Inductive datatypes
- Recursive functions on datatypes
- Inductive predicates

Compiling inductive datatypes and recursive functions to ML is fairly straightforward, and we do not explain it here. The compilation of inductive predicates is more challenging, and is discussed in more detail in §5. Recursive functions and predicates may also be *intermixed*, i.e. a recursive function may be called from within an inductive predicate, and vice versa.

$$mk\text{-}gen\ \Gamma\ i\ \alpha = gen_\alpha$$

$$mk\text{-}gen\ \Gamma\ i\ (\tau_1,\ldots,\tau_n)t = \begin{cases} gen_t\ (mk\text{-}gen\ \Gamma\ i\ \tau_1)\ \cdots\ (mk\text{-}gen\ \Gamma\ i\ \tau_n) & \text{if } t \notin \Gamma \\ gen'_t\ (mk\text{-}gen\ \Gamma\ i\ \tau_1)\ \cdots\ (mk\text{-}gen\ \Gamma\ i\ \tau_n)\ i & \text{if } t \in \Gamma \end{cases}$$

```
fun gen'_t gen_α1 ··· gen_αk i j = frequency
  [(i, fn () => one_of
     [fn () => C_1 (mk-gen {t} (i-1) τ_1^1 j) ··· (mk-gen {t} (i-1) τ_1^r1 j),
        ⋮,
      fn () => C_m (mk-gen {t} (i-1) τ_m^1 j) ··· (mk-gen {t} (i-1) τ_m^rm j)] ()),
   (1, fn () => one_of
     [fn () => D_1 (mk-gen {t} (i-1) σ_1^1 j) ··· (mk-gen {t} (i-1) σ_1^s1 j),
        ⋮,
      fn () => D_n (mk-gen {t} (i-1) σ_n^1 j) ··· (mk-gen {t} (i-1) σ_n^sn j)] ())] ()
and gen_t gen_α1 ··· gen_αk i = gen'_t gen_α1 ··· gen_αk i i
```

**Figure 2. General construction scheme for random data generators**

The overall architecture of the implemented tool is shown in Fig. 1. From a proposition $\varphi$ with free variables $x_1, \ldots, x_n$ given by the user, which is to be interpreted relative to a specific theory, the code generator produces ML code for the actual proposition, as well as the datatypes and functions used in it. For this to work, the proposition is interpreted as a function

$$f = \lambda x_1 \ldots x_n.\ \varphi :: \tau_1 \Rightarrow \cdots \tau_n \Rightarrow bool$$

Moreover, a test case generator is constructed for each datatype. These generators are then invoked by a test driver, which generates random values $r_1, \ldots, r_n$ of type $\tau_1, \ldots, \tau_n$ and then evaluates $f\ r_1 \ldots r_n$. If the result of the evaluation is *false*, the arguments $r_i$ are returned as a counterexample.

## 4. Test Data Generators

As a basis for the definition of random data generators, we assume the following functions:

```
random: int -> int -> int
one_of: 'a list -> 'a
frequency: (int * 'a) list -> 'a
```

Function `random l h` generates a random integer number $r$ with equal distribution, where $\mathtt{l} \leq r \leq \mathtt{h}$. Function `one_of xs` chooses one of the elements of the list `xs`, where each element has the same probability of being chosen. Function `frequency` takes a list of pairs `[(`$k_1$`, `$x_1$`), ..., (`$k_n$`, `$x_n$`)]` and chooses one of the elements $x_1, \ldots, x_n$. Here, the integer values $k_i$ are interpreted as *weights*, i.e. the probability of $x_i$ to be chosen is

$$P_i = \frac{k_i}{\sum_{j=1}^{n} k_j}$$

A *generator* for a type $\tau$ is a function of type $int \Rightarrow \tau$, where the integer argument of the function specifies the *size* of the test data to be generated. Using the above functions, we can define generators for the basic types of booleans and integers as follows:

```
fun gen_bool i = one_of [false, true];
fun gen_int i = one_of [~1, 1] * random 0 i;
```

We now come to a more general description of the construction of generators for arbitrary datatypes. Types in Isabelle can either be *type variables*, which we denote by greek letters $\alpha$, $\beta$, $\gamma$, …, or *complex type expressions*, which have the form $(\tau_1, \ldots, \tau_n)t$, where $t$ is a *type constructor* and $\tau_i$ are the *argument types*. Each $n$-ary type constructor $t$ is associated with a function

$$gen_t :: (int \Rightarrow \alpha_1) \Rightarrow \cdots \Rightarrow (int \Rightarrow \alpha_n) \Rightarrow int \Rightarrow (\alpha_1, \ldots, \alpha_n)t$$

that, given generators for the types $\alpha_i$, yields a generator for the type $(\alpha_1, \ldots, \alpha_n)t$. For example, the generator `gen_list` for the type $\alpha$ *list* is defined as follows:

```
fun gen_list' aG i j = frequency
  [(i, fn () => aG j :: gen_list' aG (i-1) j),
   (1, fn () => [])] ()
and gen_list aG i = gen_list' aG i i;
```

where `aG` is a generator for elements of type $\alpha$. The actual test data generation is done by the function `gen_list'` that takes two *size* arguments instead of just one. The first of these arguments is decremented with every recursive call, while the other is left unchanged and is simply passed on to the generator `aG` for the argument type $\alpha$. The probability that `gen_list aG i` generates a list with $k$ elements, where $0 \leq k \leq \mathtt{i}$, is

$$\frac{\mathtt{i}}{\mathtt{i}+1} \cdot \frac{\mathtt{i}-1}{\mathtt{i}} \cdots \frac{\mathtt{i}-(k-1)}{\mathtt{i}-(k-1)+1} \cdot \frac{1}{\mathtt{i}-k+1} = \frac{1}{\mathtt{i}+1}$$

This is independent of the value of $k$, i.e. the length of the generated lists is equally distributed.

For a complex type expression, a suitable generator can be constructed by recursion on the structure of the type. This is accomplished by the function *mk-gen* shown in Fig. 2. For example, the generator for a list of lists of integers can be constructed as follows:

$$\textit{mk-gen}\ \{\}\ \texttt{i}\ (\textit{int list list}) = \texttt{gen\_list (gen\_list gen\_int) i}$$

The additional argument $\Gamma$ of function *mk-gen* is a set of type constructors describing the context in which the invocation of the generator function takes place. If a generator for datatype $t_i$ is called recursively from within the definition of the generators for the mutually recursive datatypes $t_1, \ldots, t_n$, then $\Gamma = \{t_1, \ldots, t_n\}$, and therefore *mk-gen* will produce a call to the auxiliary function $gen'_{t_i}$ with an additional size argument. In contrast, if $t_i$ is called from elsewhere, *mk-gen* will produce a call to $gen_{t_i}$. We now consider the general recursive datatype definition

$$\begin{array}{l} \textbf{datatype}\ (\alpha_1, \ldots, \alpha_k)t = \\ \quad C_1\ \tau_1^1\ \cdots\ \tau_1^{r_1} \mid \cdots \mid C_m\ \tau_m^1\ \cdots\ \tau_m^{r_m} \\ \quad \mid D_1\ \sigma_1^1\ \cdots\ \sigma_1^{s_1} \mid \cdots \mid D_n\ \sigma_n^1\ \cdots\ \sigma_n^{r_n} \end{array}$$

The constructors $C_1, \ldots, C_m$ are *recursive*, i.e.

$$\forall 1 \le i \le m.\ \exists 1 \le j \le r_i.\ \tau_i^j = (\alpha_1, \ldots, \alpha_k)t$$

whereas $D_1, \ldots, D_n$ are *non-recursive*, i.e.

$$\forall 1 \le i \le n.\ \forall 1 \le j \le s_i.\ \sigma_i^j \ne (\alpha_1, \ldots, \alpha_k)t$$

The scheme for constructing a generator for this type is shown in Fig. 2. The first choice the generator makes is whether a recursive or a non-recursive constructor should be selected. In either case, one of the available constructors is selected with equal probability. The probability for a recursive constructor to be chosen is initially higher than for a non-recursive constructor, and decreases with every recursive call. As in the above definition of `gen_list`, this makes sure that the size of the generated test data is equally distributed and does not exceed the limit specified by the user.

## 5. Inductive predicates

Inductively defined predicates (or sets) are used in many areas of computer science. An inductive definition specifies the *smallest* set *closed* under a list of inference rules, which are called the *introduction* rules of the predicate. For example, programming language semantics or type systems are often presented in the form of such inference rules. Introduction rules are Prolog-style *Horn Clauses*, which have the form

$$(t_1^1, \ldots, t_{n_1}^1) \in p_1 \Longrightarrow \cdots \Longrightarrow (t_1^m, \ldots, t_{n_m}^m) \in p_m \Longrightarrow (t_1^0, \ldots, t_{n_0}^0) \in p_0$$

where $p_0, \ldots, p_m$ are inductively defined predicates. In addition to premises of the form $(t_1^i, \ldots, t_{n_i}^i) \in p_i$, we also allow other *side conditions*, i.e. executable boolean expressions, which can be thought of as a kind of "filter" for the computed results. To simplify the exposition, we will treat side conditions only informally in this section.

### 5.1. Mode analysis

In contrast to Prolog, whose execution model is based on *unification* and *resolution*, inductive predicates in Isabelle are executed by translating them into functional programs. This is done by performing a *dataflow analysis*, which assigns a set of possible dataflow directions to each inductive predicate. These dataflow directions, which are also called *modes*, partition arguments of inductive predicates into *input* and *output* arguments. In the sequel, we will denote a mode by the set of indices of the input arguments. Usually, there is more than one possible direction of dataflow for a predicate. For example, the predicate

$$\begin{array}{l} (\textit{Nil}, \textit{ys}, \textit{ys}) \in \textit{app} \\ (\textit{xs}, \textit{ys}, \textit{zs}) \in \textit{app} \Longrightarrow (\textit{Cons}\ x\ \textit{xs}, \textit{ys}, \textit{Cons}\ x\ \textit{zs}) \in \textit{app} \end{array}$$

may be given two lists $xs = [1,\ 2]$ and $ys = [3,\ 4]$ as input, the output being the list $zs = [1,\ 2,\ 3,\ 4]$. We may as well give a list $zs = [1,\ 2,\ 3,\ 4]$ as an input, the output being a sequence of pairs of lists $xs$ and $ys$, where $zs$ is the result of appending $xs$ and $ys$, namely $xs = [1,\ 2,\ 3,\ 4]$ and $ys = []$, or $xs = [1,\ 2,\ 3]$ and $ys = [4]$, or $xs = [1,\ 2]$ and $ys = [3,\ 4]$, etc. Another possibility would be to give all three lists $xs$, $ys$ and $zs$ as an input, the output being either *True* (encoded by the singleton sequence consisting only of the nullary tuple) if $zs$ is the result of appending $xs$ and $ys$, and *False* (encoded by the empty sequence) otherwise.

In order to execute the above clause of predicate $p_0$, a suitable order of execution needs to be chosen for the predicates $(t_1^i, \ldots, t_{n_i}^i) \in p_i$ $(1 \le i \le m)$ in the clause body, such that all variables appearing in the *input* arguments of a predicate to be executed appear either in the *output* arguments of previously executed predicates or in *input* arguments in the clause head. This makes sure that the values of all input arguments of a predicate are *known* at the point of execution. When executing a *side condition*, the values of *all* variables occurring in it must be known. More formally, the above clause for $p_0$ is said to be *well-moded* if there is a permutation $\pi$ and modes $M_i \subseteq \{1, \ldots, n_i\}$ such that for all $1 \le i \le m$

$$\begin{array}{lll} \mathsf{Vars}(in_{\pi(i)}) & \subseteq & \mathsf{Vars}(in_0) \cup \bigcup_{1 \le j < i} \mathsf{Vars}(out_{\pi(j)}) \quad \text{and} \\ \mathsf{Vars}(out_0) & \subseteq & \mathsf{Vars}(in_0) \cup \bigcup_{1 \le j \le m} \mathsf{Vars}(out_j) \end{array}$$

where $in_i$ and $out_i$ denote the list of input and output arguments of predicate $p_i$ with respect to mode $M_i$:

$$\begin{array}{rcl} in_i & = & \left[t_j^i | 1 \leq j \leq n_i \wedge j \in M_i\right] \\ out_i & = & \left[t_j^i | 1 \leq j \leq n_i \wedge j \notin M_i\right] \end{array}$$

An inductive predicate is well-moded if all its clauses are well-moded.

### 5.2. Code generation

Well-moded inductive predicates can easily be translated into functional programs that only use the built-in *pattern matching* mechanism of the underlying functional programming lanuguage instead of *unification.* To account for possible nondeterminism, the function generated from an inductive predicate returns a sequence of output values for a given input value. Since the number of output values can also be infinite, lazy lists have to be used. Lazy lists are represented by the type `'a seq`, for which we assume the following operations:

```
Seq.empty  : 'a seq
Seq.single : 'a -> 'a seq
Seq.append : 'a seq * 'a seq -> 'a seq
Seq.map    : ('a -> 'b) -> 'a seq -> 'b seq
Seq.flat   : 'a seq seq -> 'a seq
```

In the sequel, we will write `s1 ++ s2` instead of `Seq.append (s1, s2)`. In addition, we define the operator

```
fun s :-> f = Seq.flat (Seq.map f s)
```

that will be used to compose subsequent calls of predicates. Using the above operators, the predicate $p_0$ can be translated to ML as follows:

```
fun p₀ inp =
  Seq.single inp :->
    (fn in₀ => p_π(1) in_π(1) :->
        (fn out_π(1) => p_π(2) in_π(2) :->
            ⋱
                (fn out_π(m) => Seq.single out₀
                  | _ => Seq.empty)
            ⋮
          | _ => Seq.empty)
      | _ => Seq.empty)
  ++
  ...;
```

Side conditions, which are just boolean expressions, can easily be embedded into this translation scheme with the help of the following combinator:

```
fun ?? b = if b then Seq.single () else Seq.empty
```

The purpose of this combinator is to make a boolean expression behave like an inductive predicate with no output arguments. An expression evaluating to *False* corresponds to the empty sequence, whereas an expression evaluating to *True* corresponds to a singleton sequence consisting only of the nullary tuple.

### 5.3. Inductive characterization of predicate logic operators

The usual way of evaluating a propositional logic formula such as $\varphi \wedge \varphi'$ is to evaluate the subformulae $\varphi_1$ and $\varphi_2$, and then compute the value of the formula using the *truth-table semantics* for $\wedge$. Unfortunately, this approach does not extend to formulae of predicate logic such as $\exists x.\ \varphi\ x$ or $\forall x.\ \varphi\ x$, unless the domain of quantification is finite. An alternative approach, which also allows predicate logic formulae to be given a computational interpretation, is to phrase such formulae in the form of an inductive definition. A predicate logic formula of the form

$$\begin{array}{ll} & (\exists \vec{x}_1.\ \varphi_1^1\ \vec{x}_1\ \vec{y} \wedge \ldots \wedge \varphi_1^{n_1}\ \vec{x}_1\ \vec{y}) \\ \vee & \ldots \\ \vee & (\exists \vec{x}_m.\ \varphi_m^1\ \vec{x}_m\ \vec{y} \wedge \ldots \wedge \varphi_m^{n_m}\ \vec{x}_m\ \vec{y}) \end{array}$$

with free variables $\vec{y}$ corresponds to an inductive predicate $R$, which is characterized by the clauses

$$\begin{array}{l} \varphi_1^1\ \vec{x}_1\ \vec{y} \Longrightarrow \cdots \Longrightarrow \varphi_1^{n_1}\ \vec{x}_1\ \vec{y} \Longrightarrow \vec{y} \in R \\ \quad\vdots \\ \varphi_m^1\ \vec{x}_m\ \vec{y} \Longrightarrow \cdots \Longrightarrow \varphi_m^{n_m}\ \vec{x}_m\ \vec{y} \Longrightarrow \vec{y} \in R \end{array}$$

As is common in logic programming languages such as Prolog, free variables in clauses are implicitly *universally quantified.* Thus, free variables only occurring in the body of a clause are implicitly *existentially quantified.* Inductive encodings of logical operators in a theorem prover have first been proposed by Paulin-Mohring [10], who used them in the *Coq* system based on the *Calculus of Inductive Constructions.*

In order to transform an arbitrary predicate logic formula into a formula of the above form, the following rewrite rules have to be applied in a preprocessing step:

$$\begin{array}{c} (\forall x.\ P\ x) = (\neg\exists x.\ \neg P\ x) \qquad (\neg\forall x.\ P\ x) = (\exists x.\ \neg P\ x) \\ (P \longrightarrow Q) = (\neg P \vee Q) \\ \neg\neg P = P \\ (\exists x.\ P\ x \vee Q\ x) = (\exists x.\ P\ x) \vee (\exists x.\ Q\ x) \\ (\exists x.\ P\ x) \wedge Q = (\exists x.\ P\ x \wedge Q) \\ \neg(P \vee Q) = \neg P \wedge \neg Q \qquad \neg(P \wedge Q) = \neg P \vee \neg Q \\ P \wedge (Q \vee R) = P \wedge Q \vee P \wedge R \\ (P \vee Q) \wedge R = P \wedge R \vee Q \wedge R \end{array}$$

We now describe how the above translation scheme can be applied to formulae with preconditions, a notoriously problematic case for testing. Consider the formula

$$(x, y) \in I \longrightarrow P\ x\ y$$

where $I$ is an inductive predicate with the modes $\{1\}$ and $\{1, 2\}$. A naive strategy for testing this formula would be to evaluate it under an assignment of random values to the variables $x$ and $y$ (where the mode $\{1, 2\}$ is used for $I$). However, if $I$ represents a non-trivial property, it is quite unlikely that a random data generator produces suitable combinations of values for both $x$ and $y$ such that $(x, y) \in I$ will evaluate to *True*. Therefore, the precondition $(x, y) \in I$ will evaluate to *False* for most values of $x$ and $y$, and so the whole formula will evaluate to *True*. Thus, this strategy is unlikely to find counterexamples. A better approach is to generate values for $y$ in a more *goal-directed* way: If we add an explicit quantification over $y$, the resulting formula $\forall y.\ (x, y) \in I \longrightarrow P\ x\ y$ can be transformed as follows:

$$\begin{array}{rl} & \forall y.\ (x, y) \in I \longrightarrow P\ x\ y \\ = & \forall y. \neg (x, y) \in I \vee P\ x\ y \\ = & \neg \exists y.\ (x, y) \in I \wedge \neg P\ x\ y \end{array}$$

After introducing an inductively defined auxiliary predicate $R$ with the introduction rule

$$(x, y) \in I \Longrightarrow \neg P\ x\ y \Longrightarrow x \in R$$

the above formula can be rephrased as $\neg x \in R$. This time, when evaluating the auxiliary predicate $R$, the predicate $I$ will be evaluated with mode $\{1\}$, i.e. for a given value of $x$, values of $y$ will be generated such that $(x, y) \in I$ holds. This transformation easily generalizes to formulae

$$\forall \vec{y_1} \ldots \vec{y_n}.\ (\vec{x_1}, \vec{y_1}) \in I_1 \longrightarrow \cdots \longrightarrow (\vec{x_n}, \vec{y_n}) \in I_n \longrightarrow P\ \vec{x_1} \ldots \vec{x_n}\ \vec{y_1} \ldots \vec{y_n}$$

with several inductive predicates as premises. By iterated application of the translation rules shown above, we can turn this formula into

$$\neg \exists \vec{y_1} \ldots \vec{y_n}.\ (\vec{x_1}, \vec{y_1}) \in I_1 \wedge \cdots \wedge (\vec{x_n}, \vec{y_n}) \in I_n \wedge \neg P\ \vec{x_1} \ldots \vec{x_n}\ \vec{y_1} \ldots \vec{y_n}$$

## 6. Case studies

In this section, we demonstrate the applicability of our testing framework by two case studies: The formalization of a small programming language and red-black trees.

### 6.1. A programming language with parallelism and nondeterminism

While the behaviour of sequential programs is often relatively easy to grasp, it is quite hard to develop an intuitive understanding of programs involving parallelism and nondeterminism. Parallel programs are substantially harder to verify than sequential ones, since parts of a program running in parallel may interfere with each other. Testing and counterexample generation is therefore particularly helpful when reasoning about such programs. This section demonstrates the applicability of our testing framework to the operational semantics of a programming language with parallelism and nondeterminism. Programs operate on a state, which is a mapping from *addresses* to *values*. Both addresses and values are encoded as natural numbers (type *nat*). States are encoded as lists of natural numbers[1], where the $i$th element of the list denotes the value stored at address $i$.

**types** *state* = *nat list*

Moreover, our programming language contains arithmetic and boolean expressions, represented by the datatypes *aexp* and *bexp*, respectively.

**datatype** *aexp* = *PLUS aexp aexp* (**infixl** $\oplus$ *65*)
 | *MINUS aexp aexp* (**infixl** $\ominus$ *65*)
 | *V nat* | *C nat*

**datatype** *bexp* = *AND bexp bexp* | *NOT bexp*
 | *LE aexp aexp* (**infix** $\lhd$ *50*)

where $V\ n$ denotes a variable, $C\ n$ a constant, and $\lhd$ means "less than". The definitions of the evaluation functions

**consts**
 *evala* :: *state* $\Rightarrow$ *aexp* $\Rightarrow$ *nat*
 *evalb* :: *state* $\Rightarrow$ *bexp* $\Rightarrow$ *bool*

for expressions are fairly standard, and we omit them here. The datatype

**datatype** *com* = *SKIP*
 | *Assign nat aexp* (- := - *60*)
 | *Semi com com* (-; - [*60*, *60*] *10*)
 | *Cond bexp com com* (*IF* - *THEN* - *ELSE* - *60*)
 | *Par com com* (- ‖ - [*8*, *7*] *7*)
 | *Choice com com* (- ++ - [*6*, *5*] *5*)

of commands consists of the *SKIP* command that does nothing, as well as the usual operators := and ; for assignment and sequential composition. To simplify matters, we take *IF* - *THEN* - *ELSE* - as the only control structure and omit *WHILE* to avoid non-termination issues. The most important ingredients are the operators ‖ and ++ for parallel composition and nondeterministic choice. Fig. 3 shows the inductive defini-

1 It might seem more abstract to encode states as functions from addresses to values, but this would render equality between states undecidable.

**inductive** *evalc1*
**intros**
*Skip*: $\langle SKIP, s\rangle \longrightarrow_1 \langle s\rangle$
*Assign*: $\langle x := a, s\rangle \longrightarrow_1 \langle s[x{:=}evala\ s\ a]\rangle$

*Semi1*: $\langle c0{,}s\rangle \longrightarrow_1 \langle s'\rangle \Longrightarrow \langle c0;c1{,}s\rangle \longrightarrow_1 \langle c1{,}s'\rangle$
*Semi2*: $\langle c0{,}s\rangle \longrightarrow_1 \langle c0'{,}s'\rangle \Longrightarrow \langle c0;c1{,}s\rangle \longrightarrow_1 \langle c0';c1{,}s'\rangle$

*Par1*: $\langle c0, s\rangle \longrightarrow_1 \langle c0', s'\rangle \Longrightarrow \langle c0 \parallel c1, s\rangle \longrightarrow_1 \langle c0' \parallel c1, s'\rangle$
*Par1'*: $\langle c0, s\rangle \longrightarrow_1 \langle s'\rangle \Longrightarrow \langle c0 \parallel c1, s\rangle \longrightarrow_1 \langle c1, s'\rangle$
*Par2*: $\langle c1, s\rangle \longrightarrow_1 \langle c1', s'\rangle \Longrightarrow \langle c0 \parallel c1, s\rangle \longrightarrow_1 \langle c0 \parallel c1', s'\rangle$
*Par2'*: $\langle c1, s\rangle \longrightarrow_1 \langle s'\rangle \Longrightarrow \langle c0 \parallel c1, s\rangle \longrightarrow_1 \langle c0, s'\rangle$

*Choice1*: $\langle c0 \mathbin{++} c1, s\rangle \longrightarrow_1 \langle c0, s\rangle$
*Choice2*: $\langle c0 \mathbin{++} c1, s\rangle \longrightarrow_1 \langle c1, s\rangle$

*IfTrue*: $evalb\ s\ b \Longrightarrow \langle IF\ b\ THEN\ c1\ ELSE\ c2{,}s\rangle \longrightarrow_1 \langle c1{,}s\rangle$
*IfFalse*: $\neg\ evalb\ s\ b \Longrightarrow \langle IF\ b\ THEN\ c1\ ELSE\ c2{,}s\rangle \longrightarrow_1 \langle c2{,}s\rangle$

**inductive** *evalc1-tr*
**intros**
*tr1*: $(c, s) \longrightarrow_1{}^* (c, s)$
*tr2*: $(c, s) \longrightarrow_1 (c', s') \Longrightarrow (c', s') \longrightarrow_1{}^* (c'', s'') \Longrightarrow (c, s) \longrightarrow_1{}^* (c'', s'')$

**Figure 3. Inductive definition of operational semantics**

tion of the small-step semantics for the above programming language, consisting of a single-step execution relation $\longrightarrow_1$ as well as its transitive closure $\longrightarrow_1{}^*$. The execution relation operates on a *configuration* that is either a pair $\langle c, s\rangle$ consisting of a residual command $c$ and a state $s$, if the execution is not fully completed, or just a (final) state $\langle s\rangle$ reached after complete execution of a command.

As a first "theorem", one might try to prove that the behaviour of the parallel composition of a program $c$ with a program incrementing the variable *0* twice by *1* can be simulated by the parallel composition of $c$ with a program incrementing the variable *0* by two in one step:

**theorem**
$\langle (0 := V\,0 \oplus C\,1;\ 0 := V\,0 \oplus C\,1) \parallel c, s\rangle \longrightarrow_1{}^* \langle s'\rangle \Longrightarrow$
$\langle 0 := V\,0 \oplus C\,2 \parallel c, s\rangle \longrightarrow_1{}^* \langle s'\rangle$
**quickcheck**

This is obviously wrong, and quickcheck easily finds the following counterexample of size *1*:

$c = 0 := C\,0$
$s = [Suc\ 0]$
$s' = [Suc\ 0]$

Even though this is a formula with a precondition, the counterexample can be found without applying the transformation described in §5.3. There are only few states with at most one address and values $\leq 1$, and almost any command $c$ that assigns a value to the variable *0* will make the statement wrong. In contrast, the other direction

**theorem** $\langle 0 := V\,0 \oplus C\,2 \parallel c, s\rangle \longrightarrow_1{}^* \langle s'\rangle \Longrightarrow$
$\langle (0 := V\,0 \oplus C\,1;\ 0 := V\,0 \oplus C\,1) \parallel c, s\rangle \longrightarrow_1{}^* \langle s'\rangle$

is correct, and quickcheck does not find a counterexample. As a more complex example, we try to prove that parallel composition can be simulated by nondeterministic choice:

**theorem** $\langle c1 \parallel c2, s\rangle \longrightarrow_1{}^* \langle s'\rangle \Longrightarrow$
$\langle (c1;\ c2) \mathbin{++} (c2;\ c1), s\rangle \longrightarrow_1{}^* \langle s'\rangle$

This time, finding a counterexample directly by generating random values for all free variables is hopeless: There are too many degrees of freedom. We therefore apply the transformation technique from §5.3 and introduce the following auxiliary predicate:

**consts** *test* :: (*com* × *com* × *state*) *set*
**inductive** *test*
**intros**
$\langle c1 \parallel c2, s\rangle \longrightarrow_1{}^* \langle s'\rangle \Longrightarrow$
$\neg\ \langle (c1;\ c2) \mathbin{++} (c2;\ c1), s\rangle \longrightarrow_1{}^* \langle s'\rangle \Longrightarrow (c1, c2, s) \in test$

Using *test*, we can reformulate our goal as follows[2]:

**theorem** $\neg\ (c1, c2, s) \in test$
**quickcheck**

2 It should be noted that this transformation need not be done manually by the user, but is performed by the system behind the scenes. It is only shown here for illustration.

For this goal, quickcheck can find the following counterexample:

$$\begin{array}{lcl} c1 & = & IF\ C\ 0 \lhd V\ 0 \\ & & THEN\ 0 := C\ (Suc\ 0)\ ELSE\ SKIP \\ c2 & = & 0 := C\ 0 \\ s & = & [Suc\ 0] \end{array}$$

To see why this is a counterexample, consider the following execution sequence for $c1 \parallel c2$: First, the condition $C\ 0 \lhd V\ 0$ of the *IF* statement in *c1* is evaluated. Since the value of $V\ 0$ is currently *Suc 0*, the first branch $0 := C\ (Suc\ 0)$ of the *IF* statement is chosen. However, before its execution, control is handed over to *c2*, which sets variable *0* to *0*. Now control switches back to *c1* again, and the branch of the *IF* statement is executed, which resets variable *0* to *Suc 0*. This behaviour cannot be simulated by $(c1;\ c2) +\!+ (c2;\ c1)$. When executing $(c1;\ c2)$, variable *0* is left unchanged by *c1* and then set to *0* by *c2*. In contrast, when executing $(c2;\ c1)$ variable *0* is set to *0* by *c2* and again left unchanged by *c1*.

### 6.2. Red-Black trees

As a second case study, we reconsider a formalization of a functional implementation of red-black trees in Isabelle/HOL done by the software engineering group at the University of Freiburg [9]. The formalization was based on the library that is part of the *Standard ML of New Jersey* distribution.

Red-black trees are binary trees, whose nodes have an extra *colour* attribute, which can be either *red* or *black*. Assuming that the data items to be stored in the tree are *integers*, we can define the datatype of red-black trees as follows:

**datatype** $colour = R \mid B$
**datatype** $tree = E \mid T\ colour\ tree\ int\ tree$

Red-black trees must satisfy two important *invariants*. The *red invariant* says that the two children of a *red* node must be *black*, whereas the *black invariant* says that the number of black nodes must be the same for all paths from the root of the tree to the leaves. During the formalization, it turned out that the implementation of the *delete* function in the *Standard ML* library contained an error, which led to the violation of one of the above invariants. Interestingly, **quickcheck** can find a relatively small counterexample that exhibits this error. The specification that the delete function is supposed to satisfy can be expressed as follows:

**theorem** $isord\ t \wedge isin\ x\ t \wedge redinv\ t \wedge blackinv\ t \longrightarrow$
$redinv\ (delete\ x\ t) \wedge blackinv\ (delete\ x\ t)$

More informally, given an ordered tree $t$ that satisfies the *red* and *black* invariants, and an element $x$ that is contained in $t$, the tree obtained after deletion of the element must again satisfy the *red* and *black* invariants. This is not always the case, as the following counterexample found by **quickcheck** shows:

$$\begin{array}{l} t = T\ B\ (T\ B\ E\ {-1}\ E)\ 0\ (T\ B\ E\ 1\ E) \\ x = 0 \end{array}$$

Deleting the element at the root of $t$ yields the tree

$T\ B\ (T\ B\ E\ {-1}\ E)\ 1\ E$

that violates the *black* invariant, since one path from the root to the leaves contains two black nodes, while the other contains just one. However, when deleting a different element, everything works as expected. For example, deleting the element *1* yields

$T\ B\ (T\ R\ E\ {-1}\ E)\ 0\ E$

which satisfies the *black* invariant, since the node containing $-1$ has changed its colour to *red*.

## 7. Evaluation

This section is concerned with an assessment of the quality of the testing strategy described in the previous sections. In particular, we examine how the performance of the testing strategy is influenced by its parameters, namely the *size* of the test data and the number of *iterations*, i.e. the number of times a test is run with a particular test data size. A technique for analyzing test case generators, which is frequently used in software engineering, is *mutation testing* [4, 2]. From a valid theorem, such as[3]

$(ys\ @\ xs = zs\ @\ xs) = (ys = zs)$

which is taken from the database of the theorem prover, several variants (so-called *mutants*) are generated by applying certain syntactic transformations (so-called *mutation strategies*) on the term representing the proposition of the theorem. These mutants are then analyzed by the testing tool. Some of these mutated propositions may still be true, but many of them are likely to be false. The number of mutants that are detected to be false can be taken as a measure for the quality of the testing strategy. In this section, we focus on the following two mutation strategies:

1. Two *subterms* of a term having the same type are *exchanged* with each other. Since we want to generate as many false mutants as possible, we ex-

3 Note that we use = to denote equality on both lists and booleans (i.e. "if and only if"). Also, like in Standard ML, @ is the operator for *appending* two lists.

| size \ iterations | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1509 | 1291 | 1251 | 1201 | 1178 | 1168 | 1159 | 1149 | 1145 | 1138 |
| 2 | 1149 | 998 | 947 | 895 | 874 | 858 | 854 | 839 | 829 | 839 |
| 3 | 1063 | 930 | 901 | 862 | 845 | 822 | 814 | 806 | 803 | 799 |
| 4 | 1011 | 908 | 877 | 829 | 825 | 820 | 805 | 792 | 788 | 795 |
| 5 | 1015 | 894 | 865 | 835 | 824 | 805 | 804 | 793 | 794 | 773 |
| 6 | 986 | 905 | 867 | 835 | 819 | 795 | 793 | 800 | 791 | 786 |
| 7 | 971 | 898 | 851 | 824 | 818 | 820 | 796 | 793 | 786 | 780 |
| 8 | 982 | 885 | 842 | 834 | 823 | 806 | 798 | 791 | 791 | 783 |
| 9 | 972 | 875 | 855 | 838 | 814 | 809 | 789 | 795 | 797 | 792 |
| 10 | 960 | 890 | 851 | 818 | 816 | 813 | 802 | 798 | 794 | 786 |

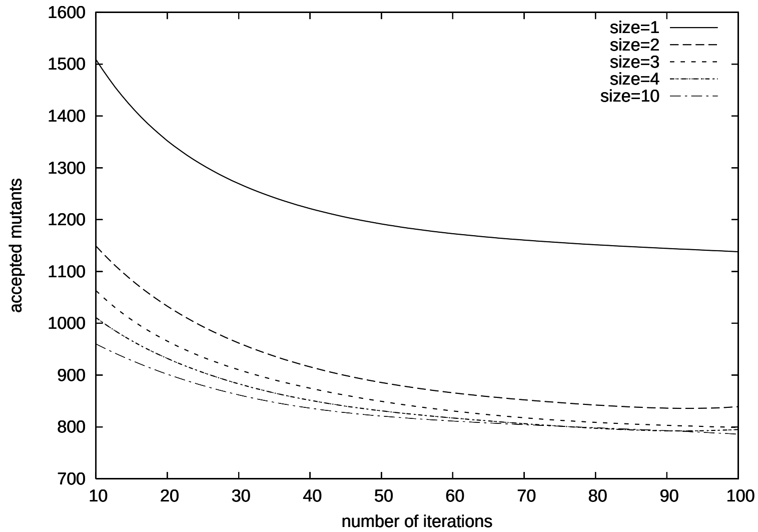


**Figure 4. Dependency of number of accepted mutants on size of test data and number of iterations**

clude mutants that are obtained by just exchanging the two arguments of a *commutative* operator such as $=$. A way to achieve this is to *canonize* the mutated terms by putting the arguments $t$ and $u$ of a commutative operator $\odot$ into a canonical order wrt. a term ordering $\prec$, i.e. rewrite $t \odot u$ to $u \odot t$ if $u \prec t$. From the above theorem, 16 different mutants can be obtained by exchanging two subterms. For example, the canonical form of the first 4 mutants is

$(ys = zs) = (xs \mathbin{@} zs = ys \mathbin{@} xs)$
$(ys = zs) = (xs = zs \mathbin{@} ys \mathbin{@} xs)$
$(ys = zs) = (xs = ys \mathbin{@} zs \mathbin{@} xs)$
$(ys = ys \mathbin{@} xs) = (zs = zs \mathbin{@} xs)$

The first 3 of these mutants are false, whereas the last one is true.

2. A *constant* occurring in a term is *replaced* by a constant of the same type taken from a given signature. Using this strategy, we can produce the following mutants from the above theorem:

$(ys \mathbin{@} xs = zs \mathbin{@} xs) \wedge (ys = zs)$
$(ys \mathbin{@} xs = zs \mathbin{@} xs) \longrightarrow (ys = zs)$
$(ys \mathbin{@} xs = zs \mathbin{@} xs) \vee (ys = zs)$

Of these 3 mutants, the second one is true. Since the signature contains no other functions of type $'a\ list \Rightarrow {'a\ list} \Rightarrow {'a\ list}$, the only possible mutation is to replace $=$ by other boolean operators.

We have applied a combination of these two mutation strategies to all theorems from the theory of lists, which is part of Isabelle/HOL, and used the testing framework to detect false mutants. The result of this experiment is shown in Fig. 4. The $x$ axis of the diagram denotes the number of iterations (i.e. test runs), while the $y$ axis denotes the number of accepted mutants. Each curve in the diagram visualizes the dependency of the number of accepted mutants on the number of iterations for a specific maximum size of the generated test data. The curves converge to the number of mutants for which no counterexample of the given size exists. The number of accepted mutants stabilizes after approximately 100 iterations. Assuming a number of 100 iterations, the influence of the test data size on the number of accepted mutants is as follows: With a maximum test data size of 1, a lot of false mutants remain undetected, since increasing the test data size to 2 reduces the number of accepted mutants by about 300. When increasing the size to 4, another 40 mutants are excluded. A further increase in the size of the test data does not seem to lead to a substantial improvement. For most practical applications a maximum data size of 10 seems to be reasonable. In order to obtain counterexamples that are as small as possible, our testing tool gradually increases the size limit for the test data, until either a counterexample is found or the size limit exceeds a specified value.

## 8. Conclusion

We have described what appears to be the first automatic testing framework for a higher-order theorem prover covering both recursive functions and inductive predicates. The difficulties of estimating or measuring the effectiveness of testing in practice are well known. Currently we can only cite the positive experience with Haskell's *QuickCheck* [3] and our personal experience. The latter has been very favourable in the early stages of a development when one has not yet built up a clear mental model and is likely to try and prove many non-theorems.

Our next aim is to realize an idea of Larry Paulson's (personal communication) and make testing an invisible part of any interactive proof attempt: modern hardware has enough spare cycles to devote some of them to finding counterexamples (by any means possible). This should be particularly helpful for novices. We believe that testing conjectures will occupy a much more prominent position in the theorem proving area in the future and could even lead to a shift from proving to refuting.

## References


[1] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer-Verlag, 2002.

[2] P. E. Black, V. Okun, and Y. Yesha. Mutation Operators for Specifications. In *15th Automated Software Engineering Conference (ASE2000), Grenoble, France (September 2000)*, pages 81–88. IEEE Computer Society, 2000.

[3] 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 (ICFP '00)*, volume 9 of *SIGPLAN Notices*, pages 268–279. ACM, September 2000.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[5] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003), Rome, Italy*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer-Verlag, 2003.

[6] M. Hanus. The integration of functions into logic programming: From theory to practice. *J. Logic Programming*, 19&20:583–628, 1994.

[7] D. Jackson. Automating first-order relational logic. In *Proc. 8th ACM SIGSOFT Int. Symp. Foundations of software engineering*, pages 130–139. ACM Press, 2000.

[8] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[9] A. Kimming. Red-black trees of SmlNJ. Studienarbeit, Universität Freiburg, January 2004.

[10] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[11] J. Slaney. FINDER, finite domain enumerator. In A. Bundy, editor, *Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Computer Science*, pages 798–801. Springer-Verlag, 1994.

[12] K. Slind and J. Hurd. Applications of Polytypism in Theorem Proving. In D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003), Rome, Italy*, volume 2758 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2003.

[13] T. Weber. Bounded model generation for Isabelle/HOL. In *2nd Int. Joint Conf. Automated Reasoning (IJCAR 2004), Workshop on Disproving – Non-Theorems, Non-Validity, Non-Provability*, 2004.

[14] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI95)*, pages 298–303, 1995.