The New Quickcheck for Isabelle

Random, Exhaustive and Symbolic Testing Under One Roof

Lukas Bulwahn

Institut für Informatik, Technische Universität München, Germany

Abstract. The new Quickcheck is a counterexample generator for Isabelle/HOL that uncovers faulty specifications and invalid conjectures using various testing strategies. The previous Quickcheck only tested conjectures by random testing. The new Quickcheck extends the previous one and integrates two novel testing strategies: exhaustive testing with concrete values; and symbolic testing, evaluating conjectures with a narrowing strategy. Orthogonally to the strategies, we address two general issues: First, we extend the class of executable conjectures and specifications, and second, we present techniques to deal with conditional conjectures, i.e., conjectures with premises. We evaluate the testing strategies and techniques on a number of specifications, functional data structures and a hotel key card system.

1 Introduction

Counterexample generators are very useful advisory tools for users of interactive theorem provers. They make developing and proving specifications an enjoyable experience. Users can identify errors leading to invalid conjectures by immediate counterexamples rather than by time-consuming unsuccessful proof attempts.

Isabelle [13] uncovers invalid conjectures by two means: Refute [17] and Nitpick [3] search for countermodels by reducing a conjecture to boolean satisfiability, whereas Quickcheck *tests* a conjecture by assigning values to the free variables of the conjecture and evaluating it. To evaluate the conjecture efficiently, Quickcheck translates the conjecture and related definitions to an ML or Haskell program, exploiting Isabelle's code generation infrastructure [10]. This allows Quickcheck to test a conjecture with millions of test cases within seconds.

In earlier work [1], Quickcheck was originally modeled after the QuickCheck tool for Haskell [7], which tests user-supplied properties of a Haskell program with randomly generated values. The first contribution of this work is to extend Quickcheck with exhaustive and narrowing-based testing as complements to random testing. Exhaustive testing checks the formula for every possible set of values up to a given bound, and hence finds counterexamples that random testing might miss. Narrowing-based testing evaluates the formula symbolically rather than with a finite set of ground values, and therefore, it can be more precise and more efficient than the other two approaches.

Another contribution is to address previous weaknesses of counterexample generation by testing. Quickcheck is inherently limited to *executable* specifications, and consequently the specification must be transformed into a functional program. We extend the class of executable conjectures in several directions:

- Narrowing-based testing can handle unbounded existential quantifiers over infinite types, enabling refutation for a class of conjectures where all other counterexample generators fail due to their imprecision or lack of support.
- For polymorphic conjectures, Quickcheck finds counterexamples by evaluating the conjecture for all finite models of small sizes.
- Quickcheck now handles *underspecified functions*, and provides a simple user interface to cope with arbitrary type definitions.

A well-known problem of testing with concrete values are *conditional conjectures*, especially those with very restrictive premises. These conjectures are problematic because when testing naively, for the vast majority of variable assignments the premise is not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the premise into account when generating values. We present three solutions for Quickcheck to generate only appropriate variable assignments:

- Derivation of custom test data generators from user declarations
- Automatic synthesis of test data generators that take the condition's definition into account
- Symbolic evaluation

To measure the impact of our improvements, we compare the various testing approaches in Quickcheck on a large set of automatically generated conjectures, on faulty implementations of functional data structures, and on a formalization of a hotel key card system, which was until now beyond the reach of Isabelle's counterexample generators. In a unrefereed paper [2], we previously presented an overview of this work as one part of Isabelle's latest developments.

The paper is structured as follows. We begin with Quickcheck's basic infrastructure (§2 and §3) for testing with concrete values, i.e., random and exhaustive testing. We show how to deal with conditional conjectures (§4) to avoid the vacuous test cases that plague most specification testing tools. Then we discuss the advantages of narrowing-based testing (§5). We highlight aspects (§6) that improve Quickcheck's performance and complete its infrastructure. Our evaluation (§7) sheds further light on the counterexample generators' strength.

2 From conjectures to test programs

Given a conjecture, Quickcheck builds a test program that combines the conjecture's evaluation with the generation of test values. This test program is then passed to Isabelle's code generator, which executes it efficiently within Isabelle's underlying ML runtime system. Turning the conjecture into a test program is a step common to both random and exhaustive testing.

We create a test program for a given conjecture by enclosing its evaluation with test data generators for its free variables. The test program returns the counterexample as an optional value: It either returns *Some* x, where x is a counterexample, or *None*. Both testing approaches define test data generators. A generator creates a finite domain of values and performs a test for a given conjecture to all elements of that domain. Our presentation here focuses on exhaustive testing. The construction for random testing is analogous.

Given a function c that checks the conjecture for a single value, the generator *exhaustive* c yields a function that checks the conjecture for all values up to a given bound. For feedback for the user, a counterexample of type τ is mapped to a fixed type *result* using the function $\mathit{reify} :: \tau \Rightarrow \mathit{result}$. We describe the generators in detail in §3. A simple test program for a conjecture C with a single variable x can be expressed as:

$$\mathit{exhaustive}\ (\lambda x.\ \text{if}\ C\ x\ \text{then}\ \mathit{None}\ \text{else}\ \mathit{Some}\ (\mathit{reify}\ x))$$

Test programs are improved by taking the common structure of conjectures into account, as a list of premises and a conclusion. If a premise does not depend on a free variable, the generation of values for this free variable can be postponed until after checking the premise. Thus, Quickcheck optimizes the test program so that it generates the values for each variable as late as possible.

For example, consider the function *insort*, which inserts an element into a sorted list in such a way that it remains sorted. If *insort* is implemented correctly, the following property should hold:

$$\mathit{sorted}\ \mathit{xs} \Longrightarrow \mathit{sorted}\ (\mathit{insort}\ x\ \mathit{xs})$$

Quickcheck generates values for xs and checks the premise $\mathit{sorted}\ \mathit{xs}$. Now only for values fulfilling the premise, Quickcheck proceeds generating values for x, and checks the conclusion $\mathit{insort}\ x\ \mathit{xs}$. Consequently, Quickcheck produces this optimized test program:

$$\begin{aligned}&\mathit{exhaustive}\ (\lambda \mathit{xs}.\ \text{if}\ \neg\ \mathit{sorted}\ \mathit{xs}\ \text{then}\ \mathit{None}\\ &\qquad\qquad\text{else}\ \mathit{exhaustive}\ (\lambda x.\ \text{if}\ \mathit{sorted}\ (\mathit{insort}\ x\ \mathit{xs})\ \text{then}\ \mathit{None}\\ &\qquad\qquad\qquad\qquad\text{else}\ \mathit{Some}\ (\mathit{reify}\ (x, \mathit{xs}))))\end{aligned}$$

In the presence of (multiple) premises, this interleaving of generation and evaluation already improves its performance dramatically. In §4, we optimize the generation and evaluation of this kind of conjectures even more.

3 Test data generators

Quickcheck automatically synthesizes test data generators for random and exhaustive testing (§3.1 and §3.2). For both testing strategies, Quickcheck supports the definition of generators: Generators of inductive data types (§3.3) are automatically defined, and generators of arbitrary type definitions (§3.4) can be defined with some guidance from the user.

Both testing approaches build on a family of test data generators. These test data generators are type-based, i.e., there is exactly one generator for each type. Generators for a complex type τ are constructed following its type structure, which is nicely described using type classes in Isabelle [18]. For example, given a generator for polymorphic lists α *list* and a generator for the type of natural numbers (type *nat*), the generator for *nat list* is implicitly composed from those two generators by the type class mechanism. Throughout the presentation, we denote an instance of an overloaded constant c with type τ by c_τ.

Generators are put together by *chaining* and *choosing between alternatives*. The generators express a nondeterministic (branching) computation. The generators' operations are closely related to operations on a *plus monad*, a generalization of the ideas for nondeterministic computations in [16].

3.1 Basic random generators

Random generators are provided by the type class *random*, which defines a function *random* of type $nat \Rightarrow seed \Rightarrow \tau \times seed$ for type τ in this class. The generator yields a value of type τ, and is parametrized by the size of values to be generated. The state *seed* is used for the underlying random engine. Random generators are chained together by the *return* and *bind* (written infix as $\gg\!=$) operators on an open state monad:

$$\begin{aligned}
&return :: \alpha \Rightarrow \sigma \Rightarrow \alpha \times \sigma \\
&return\ x\ s = (x, s) \\
&\gg\!= :: (\sigma \Rightarrow \alpha \times \sigma) \Rightarrow (\alpha \Rightarrow \sigma \Rightarrow \beta \times \sigma) \Rightarrow \sigma \Rightarrow \beta \times \sigma \\
&(f \gg\!= g)\ s = g\ x\ s' \ \textit{where}\ (x, s') = f\ s
\end{aligned}$$

With this notation, the random generator for product types is built from generators for its type constructor's arguments, where i denotes the size:

$$random_{\alpha\times\beta}\ i = random_\alpha\ i \gg\!= (\lambda x.\ random_\beta\ i \gg\!= (\lambda y.\ return\ (x, y)))$$

Given a list of generators with associated weights, *select* yields a random generator that chooses one of the generators (randomly using the *seed* value). The weights are used to give a non-uniform probability distribution to the alternatives. The random generator for the sum type $\alpha + \beta$ (with constructors *Inl* and *Inr*) illustrates selecting of alternative generators:

$$\begin{aligned}
random_{\alpha+\beta}\ i = select\ [&(1,\ random_\alpha\ i \gg\!= (\lambda x.\ return\ (Inl\ x))), \\
&(1,\ random_\beta\ i \gg\!= (\lambda x.\ return\ (Inr\ x)))]
\end{aligned}$$

3.2 Basic exhaustive generators

Similar to random generators, exhaustive generators are provided by the type class *exhaustive* with a function *exhaustive* of type $(\tau \Rightarrow result\ option) \Rightarrow nat \Rightarrow result\ option$. The exhaustive generators are expressed with continuations: They take a continuation (which ultimately checks the conjecture), and evaluate it with

all values of type τ up to the given size. Generators are chained by nesting the continuations. For example, for a given continuation c and size i, the generator for product types is defined by

$$\mathit{exhaustive}_{\alpha\times\beta}\ c\ i = \mathit{exhaustive}_{\alpha}\ (\lambda x.\ \mathit{exhaustive}_{\beta}\ (\lambda y.\ c\ (x, y)))\ i)\ i$$

As the weights of alternatives are irrelevant for exhaustive testing, generators can be simply combined with the binary operation $\sqcup$, which chooses the first *Some* value when evaluating from left to right:

$$\sqcup :: \alpha\ \mathit{option} \Rightarrow \alpha\ \mathit{option} \Rightarrow \alpha\ \mathit{option}$$
$$(\mathit{Some}\ x)\ \sqcup\ y = \mathit{Some}\ x$$
$$\mathit{None}\ \sqcup\ y = y$$

The generator for $\alpha + \beta$ joins the two exhaustive generators for types α and β employing the operator $\sqcup$:

$$\mathit{exhaustive}_{\alpha+\beta}\ c\ i =$$
$$\quad \mathit{exhaustive}_{\alpha}\ (\lambda x.\ c\ (\mathit{Inl}\ x))\ i\ \sqcup\ \mathit{exhaustive}_{\beta}\ (\lambda x.\ c\ (\mathit{Inr}\ x))\ i$$

3.3 Generators for inductive datatypes

Most commonly, new types are defined by datatype declarations. For these types, Quickcheck automatically constructs random and exhaustive generators upon the type's definition. The construction of random generators has been described in [1], so we only sketch the construction of exhaustive generators here.

We view a datatype as a recursive type definition of a sum of product types. For example, the datatype $\alpha\ \mathit{list}$ can be seen as least fixed point of the equation $\alpha\ \mathit{list} = \mathit{unit} + \alpha \times (\alpha\ \mathit{list})$. Following the scheme of exhaustive generators for product and sum type, the exhaustive generator for lists is defined recursively:

$$\mathit{exhaustive}_{\alpha\ \mathit{list}}\ c\ i = \text{if}\ (i = 0)\ \text{then}\ \mathit{None}\ \text{else}\ (c\ \mathit{Nil}\ \sqcup$$
$$\quad \mathit{exhaustive}_{\alpha}\ (\lambda x.\ \mathit{exhaustive}_{\alpha\ \mathit{list}}\ (\lambda xs.\ c\ (\mathit{Cons}\ x\ xs))\ (i-1))\ i)$$

Generalizing this example to an arbitrary datatype is almost straightforward, only recursion through functions takes some care.

3.4 Generators for arbitrary type definitions

Beyond inductive datatypes, types can also be defined by other means, e.g., by HOL-style type definitions. For such types, code generation requires special setup by the user. Quickcheck provides a simple interface with which users can specify generators. One simply lists the *constructing functions* for values of this type. Generators are then built using these functions, as if they were datatype constructors for this type. For example, red-black trees are binary search trees with a sophisticated invariant. The type $(\alpha, \beta)\ \mathit{rbt}$ contains all binary search trees with keys of type α and values of type β fulfilling the invariant. Values of this type can be generated with the invariant-preserving operations:

$empty :: (\alpha, \beta)\ rbt$
$insert :: \alpha \Rightarrow \beta \Rightarrow (\alpha, \beta)\ rbt \Rightarrow (\alpha, \beta)\ rbt$

Using these as constructing functions, Quickcheck provides random and exhaustive generators for (α, β) *rbt* that produce values starting with the *empty* tree and executing a sequence of *insert* operations. The random generator chooses the key and value for the *insert* operation randomly from the set of possible values, whereas the exhaustive generator enumerates all possible keys and values (up to a given size) for the *insert* operations.

4 Conditional conjectures

The main weakness of both random and exhaustive testing, already mentioned in the original QuickCheck for Haskell paper, is that they do not cope well with hard-to-satisfy premises. For example, when testing our previous conjecture about *insort*,

$$sorted\ xs \Longrightarrow sorted\ (insort\ x\ xs)$$

the conjecture is evaluated with all lists up to a given bound for *xs*. For all unsorted lists, the premise is not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the condition into account when generating values: In this example, we would like to only generate sorted lists.

Often, these conditional conjectures arise in the verification of functional data structures, e.g., red-black trees. A properly implemented *delete* operation for red-black trees satisfies the following property:

$$is\text{-}rbt\ t \Longrightarrow is\text{-}rbt\ (delete\ k\ t)$$

The predicate *is-rbt* captures the invariant of red-black trees on the type of binary search trees (α, β) *tree*. Again, binary trees generated naively rarely satisfy the premise, and we prefer to only generate trees satisfying the invariant.

4.1 Custom generators

The simplest solution to test conditional conjectures effectively is to employ a custom generator that has been provided by the user. Assuming the user provides a generator for some type restricted by a predicate (cf. §3.4) that matches the condition, Quickcheck automatically *lifts* the conjecture to the restricted type. For example, the conjecture about *delete* is automatically lifted to the type (α, β) *rbt*, where $Rep_{rbt}\ t'$ maps a red-black tree t' of type (α, β) *rbt* to its representative binary tree on type (α, β) *tree*:

$$is\text{-}rbt\ (Rep_{rbt}\ t') \Longrightarrow is\text{-}rbt\ (delete\ k\ (Rep_{rbt}\ t'))$$

Note that t' is now of type (α, β) *rbt*, unlike the original conjecture, where t has the type (α, β) *tree*. As all representatives of type (α, β) *rbt* satisfy the predicate *is-rbt* (by the type's construction), the premise $is\text{-}rbt\ (Rep_{rbt}\ t')$ simplifies to *true*. This way, Quickcheck obtains an unconditional conjecture, which it tests either with the random or exhaustive generator of (α, β) *rbt*.

4.2 Smart generators

A more sophisticated solution to test conditional conjectures effectively, is smart test data generators that take the condition's definition into account. These test data generators construct values in a bottom-up fashion, simultaneously testing the condition and generating appropriate values.

For our conjecture about *insort*, Quickcheck can automatically derive a test data generator that only constructs *sorted* lists. From the definition for *sorted*,

$$\begin{aligned}&\mathit{sorted}\ \mathit{Nil} = \mathit{True}\\ &\mathit{sorted}\ [x] = \mathit{True}\\ &\mathit{sorted}\ (x_1 \cdot (x_2 \cdot xs)) = (x_1 \leq x_2\ \wedge\ \mathit{sorted}\ (x_2 \cdot xs)),\end{aligned}$$

we obtain an exhaustive generator that constructs sorted lists choosing either *Nil*, a singleton list $[x]$, or appending an element to the front of a sorted list if the element is smaller than the list's head:

$$\begin{aligned}&\mathit{exhaustive\text{-}sorted}_{\alpha\ \mathit{list}}\ c\ i = \text{if}\ (i = 0)\ \text{then}\ \mathit{None}\ \text{else}\ ((c\ \mathit{Nil})\ \sqcup\\ &\quad(\mathit{exhaustive}_{\alpha}\ (\lambda x.\ c\ [x])\ (i-1))\ \sqcup\\ &\quad(\mathit{exhaustive\text{-}sorted}_{\alpha\ \mathit{list}}\ (\lambda xs'.\ \text{case}\ xs'\ \text{of}\ \mathit{Nil} \Rightarrow \mathit{None}\\ &\qquad\mid x_2 \cdot xs \Rightarrow \mathit{exhaustive}_{\alpha}\ (\lambda x_1.\ \text{if}\ (x_1 \leq x_2)\ \text{then}\ c\ (x_1 \cdot (x_2 \cdot xs))\\ &\qquad\qquad\qquad\qquad\qquad\qquad\text{else}\ \mathit{None}))\ (i-1))\ (i-1))\end{aligned}$$

Briefly, we synthesize these generators by reformulating the definitions as a set of Horn clauses and computing its data-flow dependencies (cf. [4] for more details). Applying these generators, Quickcheck's performance improves significantly (cf. §7.2).

5 Narrowing-based testing

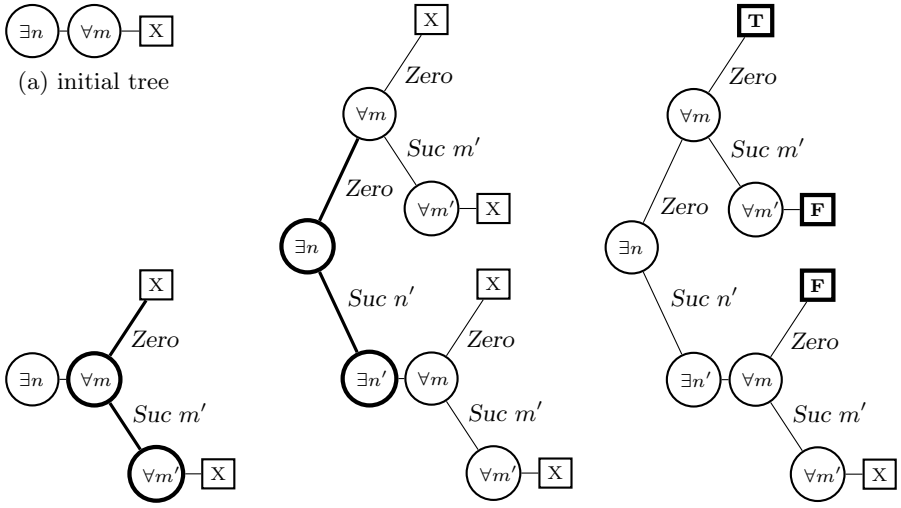
The random and exhaustive strategies suffer from two important limitations: They cannot refute propositions that existentially quantify over infinite types, and they often repeatedly test formulas with values that checks essentially the same executions (e.g., because of symmetries).

Both issues arise from the use of ground values and can be addressed by evaluating the formula symbolically. The technique is called narrowing and is well known from term rewriting. The main idea is to evaluate the conjecture with partially instantiated terms and to progressively refine these terms as needed. The following simple conjecture illustrates the benefit of the narrowing approach.

$$\exists n :: \mathit{nat}.\ \forall m :: \mathit{nat}.\ n = m$$

To disprove it, we must show for every natural number n that $\exists m.\ n \neq m$. Taking a symbolic view, if $n = 0$, we can choose any $m \neq 0$ and if $n > 0$, then 0 can serve as a witness for m.

At its core, the mechanism evaluates boolean expressions where free variables are substituted by partially instantiated terms. These terms are *constructor terms*, i.e., they are built from datatype constructors and distinct variables,



(a) initial tree (b) after 1. refinement (c) after 2. refinement (d) after 3 evaluations

Fig. 1: Refinement tree for the evaluation of $\exists n :: nat.\ \forall m :: nat.\ m = n$

e.g., *Suc* n, *Zero* and *Cons* x_1 (*Cons* x_2 x_3). Exploiting evaluations in Haskell, an expression with partial terms is evaluated to head normal form as far as possible: The execution either returns the (ground) head normal form if it is reduced despite variables in the initial term, or it indicates which variable is critical for the evaluation. For the evaluation of a boolean expression, it yields ground values *true* or *false*, if the expression is true or false for all substitutions of the free variables, resp., or it indicates the critical variable. For example, the execution determines that *Zero* $\neq$ *Suc* n is true for all natural numbers n, but the value of *Suc Zero* $\neq$ *Suc* n depends the value of n.

On top of this evaluation for partial terms, there is a refinement algorithm that refutes formulas in prenex normal form. It uses a *refinement tree* that records the results of the evaluation with partial terms and keeps track of refinements. The tree is used to determine the formula's truth value and successive evaluations with partial terms. Figure 1 shows the refinement tree during the refutation of the conjecture $\exists n :: nat.\ \forall m :: nat.\ m = n$.

Leafs of the tree carry the evaluation's result: initially *unknown* (X), and after the evaluation, the definite results *true* (T) or *false* (F). Inner nodes carry a variable and are classified as universal or existential. Branches are annotated with simple substitutions for its parent's variable, i.e., variables are assigned a single constructor with fresh variables as arguments. A path from the root to a leaf represents an assignment of partial terms by composing the substitutions along the path. For example, the path to the node annotated with T in Fig. 1d assigns n and m to *Zero*.

The truth value of a tree is defined recursively: The leafs' values are given by their annotations; the value of a universal node is the conjunction of the values of its subtrees; dually for existential nodes, it is the disjunction of its subtrees. Conjunction and disjunction are defined as in Kleene's three-valued logic with *unknown* as the third value. Starting with an initial tree with no refinements, the refinement algorithm does the following three steps:

1. Find by depth-first search a leaf, that makes the tree's truth value *unknown*, and evaluate the property with the partial terms associated with this leaf.
2. If the evaluation yields a boolean truth value, the leaf is annotated. If the evaluation calls for a refinement, the refinement tree is altered reflecting a case distinction on the critical variable.
3. If the new tree's truth value is false, we have found a counterexample to the conjecture. If it remains *unknown*, we continue with the first step. If the evaluation requires too many refinement steps, the execution is aborted. In rare cases, the tree's truth value might be true, proving the conjecture.

The illustrated evaluation in Fig. 1 starts with an initial tree that represents the quantifier part of the formula above and one leaf annotated with X (Fig. 1a). The first evaluation of $m = n$ with symbolic values m and n leads the tool to refine m. The top-most constructor of m can either be *Zero* or *Suc* (Fig. 1b). The next evaluation with $m \mapsto \mathit{Zero}$ requires a refinement of n, resulting in the state of Fig. 1c. Now, the evaluation with $n \mapsto \mathit{Zero}, m \mapsto \mathit{Zero}$ yields true, and for $n \mapsto \mathit{Zero}, m \mapsto \mathit{Suc}\ m_1$ with some fresh variable m_1 yields false. As the truth value of the upper branch $n \mapsto \mathit{Zero}$ is false, we continue with the lower branch $n \mapsto \mathit{Suc}\ n'$. The last evaluation for $n \mapsto \mathit{Suc}\ n_1, m \mapsto \mathit{Zero}$ yields false, and thus shows the invalidity of the formula (Fig. 1d). We note that the refutation never evaluated $n \mapsto \mathit{Suc}\ n_1, m \mapsto \mathit{Suc}\ m_1$.

The above example is perhaps too simple to be convincing. A more realistic example is based on the observation that the palindrome $[a, b, b, a]$ can be split into the list $[a, b]$ and its reverse $[b, a]$. Generalizing this to arbitrary lists, we boldly conjecture that

$$\mathit{rev}\ xs = xs \implies \exists ys.\ xs = ys \mathbin{@} \mathit{rev}\ ys$$

The narrowing approach immediately finds the counterexample $xs = [\mathsf{a}_1]$, inferring that there is no witness for ys in the infinite domain of lists: If ys is empty, $ys \mathbin{@} \mathit{rev}\ ys = [] \neq [\mathsf{a}_1]$, and if ys is not empty, $ys \mathbin{@} \mathit{rev}\ ys$ consists of at least two elements and hence cannot be equal to $[\mathsf{a}_1]$.

Narrowing also deals very well with conditional conjectures. In our example with the *delete* operation on red-black trees,

$$\mathit{is\text{-}rbt}\ t \implies \mathit{is\text{-}rbt}\ (\mathit{delete}\ k\ t)$$

the premise *is-rbt* t ensures that the tree t has a black root node, and in fact, after a few refinements, narrowing will only test symbolic values satisfying this property, already pruning away about half of the overall test cases.

6 Completing the infrastructure

So far, we presented the core parts of Quickcheck. In the following subsections, we touch on two further aspects: testing of polymorphic conjectures and underspecified functions.

6.1 Polymorphic conjectures

If the conjecture is polymorphic, we can instantiate the type variables with any concrete type for refuting it. Older versions of Quickcheck instantiated type variables with the type of integers (if possible depending on the type class constraints), and tested the conjecture with increasing integer values. Lately, Quickcheck prefers to use a set of small finite types instead, so that conjectures with quantifiers, e.g., existential conjectures $\exists x :: \alpha.\ P\ x$, can be refuted by a finite number of P tests.

The implementation for refuting quantified formulas over a finite type is based on the type class *enum*. This allows us to obtain implementations for more complex types by composition. E.g., the type $\alpha \times \beta \Rightarrow \gamma$ is finite if α, β and γ are finite types. The type class *enum* provides three operations for every finite type τ: $\mathit{univ} :: \tau\ \mathit{list}$ enumerates the finite universe; $\mathit{all} :: (\tau \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ and $\mathit{ex} :: (\tau \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ check universal and existential properties. The existential and universal quantifiers could be expressed just with $\mathit{univ} :: \tau\ \mathit{list}$, i.e., $\forall x :: \tau.\ P\ x = \mathit{list\text{-}all}\ P\ (\mathit{univ} :: \tau\ \mathit{list})$. Due to the strict evaluation of ML, this would be rather inefficient: The evaluation would first construct a finite (but potentially large) list of values, and then check them sequentially. To avoid the large intermediate list, we implement the quantifiers using continuations, similar to the construction of the exhaustive generators (cf. §3.2). For example, the universal quantifiers for product and sum type are implemented by

$$\mathit{all}_{\alpha\times\beta}\ P = \mathit{all}_\alpha\ (\lambda a :: \alpha.\ \mathit{all}_\beta\ (\lambda b :: \beta.\ P\ (a, b)))$$
$$\mathit{all}_{\alpha+\beta}\ P = (\mathit{all}_\alpha\ (\lambda a :: \alpha.\ P\ (\mathit{Inl}\ a))\ \wedge\ \mathit{all}_\beta\ (\lambda b :: \beta.\ P\ (\mathit{Inr}\ b)))$$

For most types, the implementation is straightforward. Only for the function type, it is a bit more involved. To construct the set of all functions $\alpha \Rightarrow \beta$, we have to create all possible mappings, i.e., all lists of type $\beta\ \mathit{list}$ with the same length as $\mathit{univ} :: \alpha\ \mathit{list}$, and transform those lists into functions.

6.2 Underspecified functions

Even though HOL is a logic of total functions, users can give underspecified function definitions. The results are total functions, but equations only exist for some subset of possible inputs. A prominent example here is the head function on lists. It is specified by $\mathit{hd}\ (\mathit{Cons}\ x\ \mathit{xs}) = x$, but no equation is given for the *Nil* constructor. Some facts only hold on the domain where the function is specified, while others may hold in general, even on values where the function has no specifying equations. For example, the conjecture about *hd* and *append*,

$hd\ (append\ xs\ ys) = (\text{if } xs = Nil \text{ then } hd\ ys \text{ else } hd\ xs)$,

is valid for all lists xs and ys, even if xs and ys are Nil. In this special case, left and right hand side are equal, i.e., they reduce to the same term $hd\ Nil$. In contrast, the conjecture $hd\ (map\ f\ xs) = f\ (hd\ xs)$ is valid only if $xs \neq Nil$. In the presence of underspecified function definitions, Quickcheck cannot distinguish the two cases occurring in the examples above. In other words, it cannot determine if a counterexample in the examples above is genuine or spurious. Therefore, it marks the counterexample as *potentially spurious*. On the two conjectures above, Quickcheck returns the potentially spurious counterexamples $xs = Nil, ys = Nil$ and $xs = Nil, f = \lambda x.\ \mathtt{a}_1$. Nevertheless, these potentially spurious counterexamples are useful in two ways: First, it makes users aware that the choice of how the underspecified function is turned into a total function might be crucial for the validity of this conjecture; second, when users know that the property only holds on values where the function is properly specified, they can validate that the given assumptions suffice to restrict the values to the defined part of the function by observing that no potentially spurious counterexample is found.

To uncover counterexamples with underspecified functions, we slightly change the test programs. The evaluation of underspecified functions in Standard ML yields a `Match` exception if it encounters a call to such a function and no pattern matches the given arguments. The test program catches this exception. If we are interested in possible counterexamples due to underspecification, Quickcheck returns the values that yield the exception as counterexample. Alternatively, if we are only interested in genuine counterexamples, Quickcheck continues to search for other values.

7 Empirical results

We evaluated Quickcheck with its different strategies on a database of theorem mutations, faulty implementations of functional data structures, and a trace-based hotel key card system.[1] The functional data structures and the key card system are well suited for comparing the different techniques to cope with conditional conjectures.

7.1 Evaluation on theorem mutations

To obtain a large set of non-theorems in Isabelle, we derive formulas *mutating* existing theorems by replacing constants and swapping arguments, as in [1, 3]. Table 1 shows the results of running the counterexample generators on 400 mutated theorems of 13 theories with a very liberal time limit of 30 seconds. The chosen set of theories focuses on executable ones, and leaves out those that are obviously not executable. For example, theories with axiomatic definitions or

[1] The test data is available at http://www21.in.tum.de/~bulwahn/cpp2012.tar.gz

Theory	Counterexample generators			
	Random	Exhaustive	Narrowing	Nitpick
Arithmetics				
Divides [fin]	199/318	212/318	221/343	259/400
Divides [int]	224/369	239/369	248/394	
GCD	203/294	203/294	228/336	216/400
MacLaurin [fin]	44/61	44/61	45/77	19/400
MacLaurin [int]	55/79	55/79	56/95	
Set Theory				
Fun [fin]	214/394	215/394	201/396	235/400
Fun [int]	146/254	144/254	161/326	
Relation [fin]	248/395	251/395	248/395	247/400
Relation [int]	139/230	155/230	160/258	
Set [fin]	246/395	246/395	249/395	260/400
Set [int]	205/329	206/329	220/369	
Wellfounded [fin]	229/372	233/372	232/373	249/400
Wellfounded [int]	45/94	47/94	51/122	
Datatypes				
List [fin]	197/319	197/318	215/354	245/400
List [int]	191/312	193/312	212/351	
Map [fin]	257/400	257/400	257/400	258/400
Map [int]	146/221	148/221	160/248	
AFP Theories				
Huffman	244/399	248/399	246/399	251/400
List-Index	256/399	256/399	263/399	271/400
Max-Card-Matching [fin]	152/345	212/345	212/345	214/400
Max-Card-Matching [int]	4/11	4/11	4/11	
Regular-Sets	154/304	152/304	210/368	142/400

Table 1: Results for running counterexample generators on mutated theorems on a Intel Core2 Duo T7700 2.40GHz with a time limit of 30 seconds

coinductive datatypes are not executable with Isabelle's code generation. Conjectures in these theories are only refuted by Nitpick.

The four columns show the absolute number of genuine counterexamples of the different approaches: random testing, exhaustive testing, narrowing-based testing, and Nitpick. In a cell with values A/B, A is the number of genuine counterexamples and B the number of executable mutants of the corresponding counterexample generator. As Nitpick handles arbitrary specifications, it is able to check all 400 mutants. Quickcheck can use finite types or integers to instantiate polymorphic conjectures (cf. §6.1). For theories with polymorphic conjectures, we show both modes separately in the table, indicated with [fin] and [int]. Using finite types for polymorphic conjectures makes almost all conjectures in the set theory domain amenable to Quickcheck, closing the previously existing gap

between Quickcheck and Nitpick in this domain. The narrowing-based testing can execute more conjectures than concrete testing with random and exhaustive testing. We gain most on the Regular-Sets theory, increasing from 304 to 368.

We also compared the tools against each other, and measured the number of counterexamples that can be found uniquely by one tool compared to another. Exhaustive testing slightly outperforms random testing. Narrowing often finds a few more counterexamples than exhaustive testing, but this is mainly due to the larger set of executable formulas. Narrowing and Nitpick complement each other to some extent, as witnessed most prominently by Isabelle's GCD theory. In absolute numbers, narrowing and Nitpick find 228 and 216 counterexamples; hence only differing by 12. However, they succeed on different conjectures—narrowing finds 23 counterexamples where Nitpick fails, Nitpick finds 11 where narrowing fails—meaning that employing them in combination yields 239 counterexamples.

To illustrate the differences in strength between testing with Quickcheck and model finding with Nitpick, we show two interesting examples of our evaluation. On the one hand, consider one of the monotonicity lemmas for integer division:

$$b{\cdot}q{+}r = b'{\cdot}q'{+}r' \wedge 0 \leq b'{\cdot}q'{+}r' \wedge r' < b' \wedge 0 \leq r \wedge 0 < b' \leq b \Longrightarrow q \leq q'$$

For Quickcheck, it is no problem to detect a typo that changes the second premise to $0 \leq b' \cdot b' + r'$. It produces the counterexample $b = -2$; $q = 3$; $r = 1$; $b' = -2$; $q' = 1$; $r' = 3$ instantaneously, while Nitpick replies after seven minutes with a similar counterexample.

On the other hand, in the Isabelle theory of maximal matchings in graphs (Max-Card-Matching), a certain invalid conjecture is refuted by constructing a graph with 4 vertices and a matching with two edges. Owing to the power of its SAT solver, Nitpick finds this matching within a few seconds. Exhaustive testing tries to enumerate all graphs and searches for matchings quite naively. Thus, Quickcheck needs roughly a minute to find a counterexample. Random testing does not find the counterexample, even with 100,000 iterations for each size and testing a few minutes—a matching for a valid graph is too unlikely to obtain by randomly chosen values. Narrowing prunes the search space before evaluating the conjecture with all possible concrete values, and finds a counterexample in about thirty seconds.

These two examples demonstrate the strength of both tools: Quickcheck is strong on arithmetics, while Nitpick handles well boolean constraints over finite domains.

7.2 Functional data structures

Beyond the mutations of lemmas, we evaluated the different testing approaches on faulty implementations of typical functional data structures. We injected faults by adding typos into the correct implementations of the delete operation of AVL trees, red-black trees, and 2-3 trees. By adding typos, we create 10 different (possibly incorrect) versions of the delete operation for each data structure. On 2-3 trees, we check two invariants of the delete operation, keeping

	R_{2K}	R_{20K}	Exh.	Cu.G.	Sm.G.	Nar.	Nit.
AVL trees	5	7	7	9	9	11	4
Red-black trees	10	18	21	22	19	26	11
2-3 trees	5	5	7	11	12	12	0

Table 2: Number of counterexamples on faulty implementations of functional data structures (time limit: 30 s for AVL and red-black trees; 120 s for 2-3 trees)

the tree balanced and ordered, i.e., *balanced* $t \Longrightarrow$ *balanced* (*delete* k t), and *ordered* $t \Longrightarrow$ *ordered* (*delete* k t). We check two similar properties for AVL trees, and three similar properties for red-black trees. With the 10 versions, this yields 20 tests each for 2-3 and AVL trees, and 30 tests for red-black trees, on which we apply various counterexample generators. In this setting, we compare the techniques to deal with *conditional conjectures*. Random testing is applied with 2,000 and 20,000 iterations for each size (abbrev. R_{2K}, R_{20K}). Furthermore, we used exhaustive testing (Exh.), custom generators (Cu.G., §4.1), smart generators (Sm.G., §4.2), narrowing (Nar.) and Nitpick (Nit.).

Table 2 summarizes the results. Overall, narrowing, smart, and custom generators beat exhaustive testing, which itself performs better than random testing and Nitpick. Nitpick struggles with large functional programs and is limited to shallow errors in the smaller implementations of AVL and red-black trees. Increasing the number of iterations for random testing helps, but in our experience, it does not find any error that was not also found by testing exhaustively. For the 2-3 trees, the smart generators and narrowing find errors in 5 more cases than exhaustive testing. However, in principle, exhaustive testing should find the errors eventually. Thus, in these more intrinsic cases, we increased the time for the naive exhaustive testing to finally discover the fault. However, even after one hour of testing, exhaustive testing was not able to detect a single one of them. This shows that using the test data generators and narrowing-based testing in this setting is clearly superior to naive exhaustive testing. The smart generators and narrowing find 12 errors in 20 conjectures. In the eight cases where they did not find anything within the time limit, even testing more thoroughly for an hour did not reveal any further errors. Most probably, the property still holds, as the randomly injected faults do not necessarily affect the invariant.

7.3 Trace-based hotel key card system

As a further case study, we checked a hotel key card system by Nipkow [12]. The faulty system contains a tricky man-in-the-middle attack, which is only uncovered by a trace of length 6. The formalization uses a restrictive predicate that describes in which order specific events occur. Due to the occurrence of existential quantifiers, the original specification is not executable for random and exhaustive testing. Even after refinements to obtain an executable reformulation, the naive random and exhaustive testing fail to find the counterexample

within ten minutes of testing, as the search space is too large. Smart generators include some processing that detects if the values of existential quantifiers are bound in the formula. Therefore, we do not have to reformulate the specification when we use smart generators. Employing these smart generators, we can find the attack within a few seconds. Narrowing can handle the existential quantifiers in principle, but in practice it performs badly with the deeply nested existential quantifiers in the specification. This renders it impossible to find the counterexample with narrowing. After eliminating the existential quantifiers manually, we also obtain a counterexample with narrowing within a few seconds.

On this trace-based version of the hotel key card system, Nitpick fails to find the counterexample with a time limit of ten minutes. However, Nitpick finds the counterexample on an equivalent *state-based* formalization of the hotel key card system (cf. [3], §6.2). This indicates that Quickcheck and Nitpick excel on formalizations with different specification styles: Nitpick on relational descriptions, Quickcheck on realistic functional programs and trace-based descriptions.

8 Related work

The success story of Haskell's QuickCheck [7] has led to many descendants in interactive theorem provers. Besides Isabelle, PVS [14], Agda [8], ACL2 [9] and ACL2 Sedan [5] include a random testing tool like the original QuickCheck.

The tool in ACL2 Sedan simplifies the conjecture using a synergistic combination of random testing and theorem proving: The application of selected theorem proving methods before testing can ease testing of conditional conjectures.

Our exhaustive testing is inspired by Haskell's SmallCheck [15], but is targeting ML with its strict evaluation. The implementation of Haskell's SmallCheck takes advantage of its laziness, simplifying the definition of generators, while Isabelle's tool takes the strictness of ML into account and uses continuations.

Tools using narrowing for testing functional programs symbolically are the Agsy tool [11] for Agda, EasyCheck [6] for the programming language Curry, and LazySmallCheck [15] for Haskell. Like LazySmallCheck, our implementation exploits Haskell's lazy evaluation and *imprecise exceptions*. The refinement algorithm of [15] only allows universal properties whereas our refinement algorithm can also deal with existential quantifiers.

9 Conclusion

As we have seen, the methods to uncover invalid conjectures, testing and model finding, implemented by the counterexample generators Quickcheck and Nitpick in Isabelle, have their justification. Quickcheck with its new testing strategies and our effort to extend its applicability allows to check many conjectures effectively that were previously beyond the scope of testing. Isabelle's users benefit from having all these strategies at their disposal, because they complement each other very well. Unmentioned so far, Quickcheck's performance also profits from the fact that code generation in Isabelle is becoming more common and widely used.

Isabelle's library provides many additions to set up code generation for numerous purposes. To validate specifications with simple examples before proving, users invest some time to make their specifications executable. Quickcheck returns this investment the first time users encounter an invalid conjecture, so they can correct an error immediately instead of wasting hours on an impossible proof.

Acknowledgment. I thank Jasmin Blanchette, Brian Huffman, Peter Lammich, Tobias Nipkow, Lars Noschinski, Andrei Popescu, Thomas Tuerk, Dmitriy Traytel, Tjark Weber and the anonymous referees for suggesting several textual improvements. I acknowledge funding from DFG doctorate program 1480 (PUMA).

References

1. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) SEFM 2004. pp. 230–239. IEEE C.S. (2004)
2. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer (2011)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer (2010)
4. Bulwahn, L.: Smart Testing of Functional Programs in Isabelle. In: LPAR 2012. LNCS, vol. 7180, pp. 153–167. Springer (2012)
5. Chamarthi, H.R., Dillinger, P., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving (2011), avail. at http://arxiv.org/pdf/1105.4394
6. Christiansen, J., Fischer, S.: EasyCheck – Test Data for Free. In: FLOPS '08. LNCS, vol. 4989, pp. 322–336. Springer (2008)
7. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP '00. pp. 268–279. ACM (2000)
8. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: TPHOLs 2003. LNCS, vol. 2758, pp. 188–203. Springer (2003)
9. Eastlund, C.: Doublecheck your theorems. In: 8th Int. Workshop on the ACL2 Theorem Prover and its Applications (2009)
10. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010)
11. Lindblad, F.: Property directed generation of first-order test data. In: Morazán, M. (ed.) TFP 2007. pp. 105–123. Intellect (2008)
12. Nipkow, T.: Verifying a Hotel Key Card System. In: ICTAC '06. LNCS, vol. 4281. Springer (2006), invited paper
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
14. Owre, S.: Random testing in PVS. In: AFM '06 (2006)
15. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In: Haskell Symp. '08. pp. 37–48 (2008)
16. Wadler, P.: How to replace failure by a list of successes. In: Functional programming languages and computer architecture. LNCS, vol. 201, pp. 113–128. Springer (1985)
17. Weber, T.: SAT-based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Institut für Informatik, Technische Universität München, Germany (2008)
18. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: Gunter, E.L., Felty, A. (eds.) TPHOLs '97. LNCS, vol. 1275, pp. 307–322 (1997)