Smart testing of functional programs in Isabelle

Lukas Bulwahn

Technische Universität München

Abstract. We present a novel counterexample generator for the interactive theorem prover Isabelle based on a compiler that synthesizes test data generators for functional programming languages (e.g. ML, Haskell) from specifications in Isabelle. In contrast to naive type-based test data generators, the smart generators take the preconditions into account and only generate tests that fulfill the preconditions.
The smart generators are constructed by a compiler that reformulates the preconditions as logic programs and analyzes them with an enriched mode inference. From this inference, the compiler can construct the desired generators in the functional programming language.
Applying these test data generators reduces the number of tests significantly and enables us to find errors in specifications where naive random and exhaustive testing fail.

1 Introduction

Writing programs and specifications is an error-prone business, and testing is common practice to find errors and validate software. Being aware that testing cannot prove the absence of errors, formal methods are applied for safety- and security-critical systems. To ensure the correctness of programs, critical properties are guaranteed by a formal proof. Proof assistants are used to develop a proof with trustworthy sound logical inferences. Once one has completed the formal proof, the proof assistant certifies that the program meets its specification. But in the process of proving, errors could still be revealed and tracking these down by failed proof attempts is a tedious task for the user. Undoubtedly, testing is still fruitful on the way to quickly detect errors in programs and specifications while the user attempts to prove them. Modern interactive theorem provers therefore do not only provide means to prove properties, but also to *disprove* properties in the form of counterexample generators.

Without specifications, it is common practice to write manual test suites to check properties. However, having a formal specification at hand, we can automatically generate test data and check if the program fulfills its specification. Such an automatic specification-based testing technique for functional Haskell programs was introduced by the popular tool QuickCheck [8], which is based on random testing. The tool SmallCheck [19] also tests Haskell programs against its specification, but is based on exhaustive testing.

The interactive theorem prover Isabelle [22] provides a counterexample generator [3], which currently incorporates the two approaches, random and exhaustive testing, similar to QuickCheck and SmallCheck. It works well on specifications that have weak preconditions and properties in a form that is directly executable in the functional language. If the property to be tested includes a precondition, both approaches generate test data that seldom fulfill the precondition, and so most of the execution time for testing is spent generating useless test values and rejecting them.

Our new approach aims to only generate test data that fulfill the precondition. The test data generator for a given precondition is produced by a compiler[1] that analyzes preconditions and synthesizes a purely functional program that serves as generator. For this purpose, the compiler reformulates the preconditions as logic programs by translating formulas in predicate logic with quantifiers and recursive functions to Horn clauses. The compiler then analyzes the Horn clauses with a data flow analysis, which determines which values can be computed from other values and which values must be generated. From this analysis, the compiler constructs the desired generators. This way, a much smaller number of test cases suffices to exhaustively test a program against its specification. Consequently, we can find errors in specifications where random and naive exhaustive testing fail to find a counterexample in a reasonable amount of time.

After discussing related work (§1.1), we show examples that motivate the work on our new counterexample generator (§2). In the main part, we then describe key ideas of this counterexample generator, the preprocessing, the data flow analysis and compilation (§3 to §6). In the end, we evaluate our counterexample generator compared with the existing approaches (§7).

1.1 Related work

The aforementioned Haskell tool QuickCheck has many descendants in interactive theorem provers, e.g., Agda/Alfa, ACL2, ACL2 Sedan, Isabelle and PVS, and in a variety of programming languages. QuickCheck uses test data generators that create random values to test the propositions. Random testing can handle propositions with strong preconditions only very poorly. To circumvent this, the user must manually write a test data generator that only produces values that fulfill the precondition. SmallCheck tests the propositions exhaustively for small values. It also handles propositions with strong preconditions poorly, but in practice handles preconditions better than QuickCheck because it gives preference to small values, and they tend to fulfill the commonly occurring preconditions more often. Lazy SmallCheck [19] uses partially-instantiated values and a refinement algorithm to simulate narrowing in Haskell. This is closely related to the work of Lindblad [13] and EasyCheck [7], based on the narrowing strategy in the functional logic programming language Curry [11]. This approach can cut the search space of possible values to check, if partially instantiated values already violate the precondition. The three approaches, QuickCheck (without manual test data generators), SmallCheck and Lazy SmallCheck, are examples of *black-box testing*, i.e., they do not consider the description of the precondition – they generate (partial) values and test the precondition.

The counterexample generators in Isabelle translate the conjecture and related definitions to an ML program, exploiting Isabelle's code generation infrastructure [10].

[1] Throughout the presentation, we use the term *compilation* with a very specific meaning: to designate our translation of Horn specifications in Isabelle into programs written in a functional programming language.

Employing this translation yields a very efficient evaluation: The ML runtime environment can check millions of test cases within seconds, which is thousands of times faster than evaluating within the prover. Like the existing counterexample generators in Isabelle, the new one also builds upon this translation. Previous work [2] on code generation focused on the *verification* of the transformation of Horn clauses to functional programs, whereas the focus of this work is the extension and application of the transformation for *counterexample generation*. Our new counterexample generator is a glass-box testing approach, i.e., it considers the description of the precondition and compiles a purely functional program that generates values that fulfill the precondition. We reported about this work in an early stage in [5]. Closely related to our work is the glass-box testing by Fischer and Kuchen [9] for functional logic programs, but they take advantage of narrowing and nondeterministic execution in Curry.

Another approach to finding values that fulfill the preconditions is to use a CLP(FD) constraint solver, as done by Carlier et al. [6]. A completely different approach to finding counterexamples is translating the specification to propositional logic and invoking a SAT solver, as performed by the Isabelle tools Refute [21] and Nitpick [4].

2 Motivation

The previously existing counterexample generators in Isabelle, which test with random values or exhaustively with small values, perform well on conjectures without preconditions. For example, for the invalid conjecture about lists[2]

$$\mathit{reverse}\ (\mathit{append}\ xs\ ys) = \mathit{append}\ (\mathit{reverse}\ xs)\ (\mathit{reverse}\ ys),$$

the counterexample generators provide the counterexample $xs = [a_1]$ and $ys = [a_2]$ (for atoms $a_1 \neq a_2$) instantaneously. For conjectures of this kind, random and exhaustive testing are perfectly suited thanks to their lightweight nature.

But random and exhaustive testing generate values without analyzing the conjecture. This can lead to many vacuous test cases, as in this simple example:

$$\mathit{length}\ xs = \mathit{length}\ ys \wedge \mathit{zip}\ xs\ ys = zs \Longrightarrow \mathit{map}\ \mathit{fst}\ zs = xs \wedge \mathit{map}\ \mathit{snd}\ zs = ys$$

The random and exhaustive strategies first generate values for xs, ys, and zs in an unconstrained fashion and then check the premises, namely that xs and ys are of equal length and that zs is the list obtained by zipping xs and ys together. For the vast majority of variable assignments, the premises are not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the premises into account when generating values. For further illustration, we focus on a simpler valid conjecture about distinct lists:

$$\mathit{distinct}\ xs \Longrightarrow \mathit{distinct}\ (\mathit{tl}\ xs)$$

The previously existing counterexample generator, testing exhaustively, produces the following test program in Standard ML to check the validity of this conjecture:

[2] We use common notations from functional programming languages: [] and $x \cdot xs$ denote the two list constructors *Nil* and *Cons x xs*, lists, such as $(x \cdot (y \cdot (z \cdot \mathit{Nil})))$, are conveniently written as $[x, y, z]$. The tail of a list xs is obtained with $\mathit{tl}\ xs$, where $\mathit{tl}\ [] = []$ and $\mathit{tl}\ (x \cdot xs) = xs$. Furthermore, free variables are implicitly universally quantified.

```
**val** $\mathit{generate\text{-}nat}\ \mathit{size}\ \mathit{chk} = \mathit{if}\ \mathit{size} = 0\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{case}\ \mathit{chk}\ 0\ \mathit{of}$
    $\mathit{Some}\ \mathit{xs} \Rightarrow \mathit{Some}\ \mathit{xs}$
  $\mid \mathit{None} \Rightarrow \mathit{generate\text{-}nat}\ (\mathit{size} - 1)\ (\lambda n.\ \mathit{chk}\ (n + 1))$

**val** $\mathit{generate\text{-}list}\ \mathit{size}\ \mathit{chk} = \mathit{if}\ \mathit{size} = 0\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{case}\ \mathit{chk}\ []\ \mathit{of}$
    $\mathit{Some}\ \mathit{xs} \Rightarrow \mathit{Some}\ \mathit{xs}$
  $\mid \mathit{None} \Rightarrow \mathit{generate\text{-}nat}\ (\mathit{size} - 1)\ (\lambda x.\ \mathit{generate\text{-}list}\ (\mathit{size} - 1)$
            $(\lambda \mathit{xs}.\ \mathit{chk}\ (x \cdot \mathit{xs})))$

**val** $\mathit{test}\ \mathit{xs} = \mathit{if}\ \mathit{distinct}\ \mathit{xs}\ \wedge \neg\ \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})\ \mathit{then}\ \mathit{Some}\ \mathit{xs}\ \mathit{else}\ \mathit{None}$

**val** $\mathit{check}\ \mathit{size} = \mathit{generate\text{-}list}\ \mathit{size}\ (\lambda \mathit{xs}.\ \mathit{test}\ \mathit{xs})$
```

The *check* function implements a simple generate-and-test loop. It uses the function *generate-list* that generates all possible lists (of natural numbers) up to a given bound and iteratively calls the *test* function to test the property at hand. It returns the found counterexample as an optional value, i.e., if the property holds for all values up to the given bound, the *check* function returns *None*, otherwise the counterexample is returned as result with *Some*.

Our new approach interleaves generation and checking in a way that avoids generating lists that are not distinct. From the definition of the *distinct* predicate,

$$\mathit{distinct}\ [] = \mathit{True}$$
$$\mathit{distinct}\ (x \cdot \mathit{xs}) = (x \notin \mathit{set}\ \mathit{xs} \wedge \mathit{distinct}\ \mathit{xs}),$$

we can derive how to construct distinct lists: First, the empty list is distinct; secondly, larger distinct lists can be constructed taking a (shorter) distinct list and appending an element which is not in the list already to its front. This insight is reflected in the following test data generator:

```
**val** $\mathit{generate\text{-}distinct}\ \mathit{size}\ \mathit{chk} = \mathit{if}\ \mathit{size} = 0\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{case}\ \mathit{chk}\ []\ \mathit{of}$
    $\mathit{Some}\ \mathit{xs} \Rightarrow \mathit{Some}\ \mathit{xs}$
  $\mid \mathit{None} \Rightarrow \mathit{generate\text{-}distinct}\ (\mathit{size} - 1)\ (\lambda \mathit{xs}.\ \mathit{generate\text{-}nat}\ (\mathit{size} - 1)$
            $(\lambda x.\ \mathit{if}\ x \notin \mathit{set}\ \mathit{xs}\ \mathit{then}\ \mathit{chk}\ (x \cdot \mathit{xs})\ \mathit{else}\ \mathit{None}))$
```

The function *generate-distinct* only generates and tests the given property with distinct lists. It constructs lists by applying the two rules mentioned above. With this generator at hand, we can check the conclusion more efficiently by:

```
**val** $\mathit{test}\ \mathit{xs} = \mathit{if}\ \neg\ \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})\ \mathit{then}\ \mathit{Some}\ \mathit{xs}\ \mathit{else}\ \mathit{None}$
**val** $\mathit{check}\ \mathit{size} = \mathit{generate\text{-}distinct}\ \mathit{size}\ (\lambda \mathit{xs}.\ \mathit{test}\ \mathit{xs})$
```

Using these *smart* test data generators reduces the number of tests, and as our evaluation (§7) shows, this allows us to explore test values of larger sizes where exhaustive testing cannot cope with the explosion of useless test values. More precisely, in our simple example, naive exhaustive testing cannot check all lists of size 15 within one hour, where the smart generator can easily explore all the lists up to this size within 30 seconds.

In the following sections, we describe how we synthesize these test data generators automatically from the precondition's definition.

3 Overview of the tool

In this section, we present the overall structure of our counterexample generator, and motivate the key features and design decisions. The detached presentation of individual components is then discussed in the following three sections.

3.1 Design decisions

QuickCheck and SmallCheck execute the program with concrete values. Testing with concrete values has the clear advantage of being natively supported by the functional programming language, in our case ML, and hence can be executed very fast. But testing with concrete values has the drawback that a large set of test inputs may exhibit indistinguishable executions. E.g., in our example about distinct lists, the lists $[1,1,2]$, $[1,1,3]$, $[1,1,4]$, ... are all non-distinct because of the non-distinct prefix $[1,1]$, and hence testing the conjecture with all these lists succeeds without even checking its conclusion.

An alternative to testing functional programs is executing the program by a *needed narrowing strategy* [1], which executes the program symbolically as far as possible. It avoids symmetric executions, i.e., a set of input values that result in the same execution. It checks the conjecture for a set of values with one symbolic execution, reducing the number of tests. In our example, all the lists $[1,1,2]$, $[1,1,3]$, $[1,1,4]$, ... can be treated immediately with one symbolic execution $1 \cdot (1 \cdot xs)$, where xs is a free variable representing any list of natural numbers. Symbolic executions usually result in a non-deterministic computation, which is implemented with a backtracking mechanism, as known from Prolog. This execution principle requires some overhead, which then causes symbolic testing to be slower than testing with concrete values, if the number of eliminated symmetric executions is too low to compensate for the execution's overhead.

Two circumstances contribute to the fact that symbolic executions frequently do not pay off in practice: First, large parts of the program are purely functional executions; nevertheless one inherits some overhead even in those parts of the execution. Second, if the conclusion is *hyperstrict*, i.e., requires checking with all test values, it incurs the overhead of symbolic executions, but ends up doing all executions necessarily with concrete values anyway.

Our test data generators aim to find a balance between *fast execution with concrete values* and *avoiding symmetric executions*. The test data generators produce concrete values during the execution, so that it can be translated directly into the target functional programming language.

For conjectures without preconditions, we enumerate all possible concrete values. This is quite effective, because usually there are only very few symmetric executions in that case. When preconditions occur in conjectures, test data generators only produce values fulfilling the precondition, and then test the conclusion. We find values fulfilling the precondition by an implementation that queries the precondition's predicate for all possible values up to some bound. The generator will enumerate possible values, similar to a query in Prolog, but returning only ground solutions, i.e., not using logical variables. The query is integrated in a lightweight fashion into the test program by a compilation. It retains purely functional evaluations, detects values that can be computed by inferring data flow in the program (between variables), and combines it with generation of values if data flow cannot be inferred.

In the end, this static analysis and the compilation lead to test data generators for the preconditions. They discard useless test inputs before generating them, and keep the execution mechanism simple to target functional programming languages. If large parts of symmetric executions are avoided by the data flow analysis, these generators can explore the space of test input faster than symbolic and concrete executions.

3.2 Architecture

The counterexample generator performs these steps: As the original specification can be defined using various definitional mechanisms, the specification is preprocessed by a few simple syntactic transformations (§4) to Horn clauses. The core component, which was previously described in [2], consists of a static data flow analysis, the mode analysis (§5) and the code generator (§6). This core component only works on a syntactic subset of the Isabelle language, namely Horn clauses of the following form:

$$Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$$

In a premise $Q_i\ \overline{u}_i$, Q_i must be a predicate defined by Horn clauses and the terms $\overline{u}_i$ must be constructor terms, i.e., only contain variables or datatype constructors. Furthermore, we allow negation of atoms, assuming the Horn clauses to be stratified. If a premise obeys these restrictions, the core compiler infers modes and compiles functional programs for the inferred modes. If a premise has a different form, e.g., the terms contain function symbols, or a predicate is not defined by Horn clauses, the core compiler will treat them as side conditions. For side conditions, the mode analysis does not infer modes, but requires all arguments as inputs. Enriching the mode analysis, we mark unconstrained values to be generated. Once we have inferred modes for the Horn clauses, these are turned into test data generators in ML using non-deterministic executions and type-based generators.

4 Preprocessing

In this section, we sketch how specifications in predicate logic and functions are preprocessed to Horn clauses. A definition in predicate logic is transformed to a system of Horn clauses, based on the fact that a formula of the form $P\ \overline{x} = \exists \overline{y}.\ Q_1\ u_1 \wedge \cdots \wedge Q_n\ u_n$ can be soundly underapproximated by a Horn clause $Q_1\ u_1 \Longrightarrow \cdots \Longrightarrow Q_n\ u_n \Longrightarrow P\ \overline{x}$. Predicate logic formulas in a different form are transformed into the form above by a few logical rewrite rules in predicate logic. We rewrite universal quantifiers to negation and existential quantifiers, put the formula in negation normal form, and distribute existential quantifiers over disjunctions. In the process of creating Horn clauses, it is necessary to introduce new predicates for subformulas, as our Horn clauses do not allow disjunctions within the premises or nested expressions under negations. Furthermore, we take special care of *if*, *case* and *let*-constructions.

Example 1. The *distinct* predicate on lists is defined by the two equations,

$$\mathit{distinct}\ [] = \mathit{True}$$
$$\mathit{distinct}\ (x \cdot xs) = (x \notin \mathit{set}\ xs \wedge \mathit{distinct}\ xs)$$

In the preprocessing step, these are made to fit the syntactic restrictions of the core component, yielding the two Horn clauses:

$$\mathit{distinct}\ []$$
$$x \notin \mathit{set}\ xs \Longrightarrow \mathit{distinct}\ xs \Longrightarrow \mathit{distinct}\ (x \cdot xs)$$

To enable inversion of functions, we preprocess n-ary functions to $(n+1)$-ary predicates defined by Horn clauses, which enables the core compilation to inspect the definition of the function and leads to better synthesized test data generators. This is achieved by *flattening* a nested functional expression to a flat relational expression, i.e., a conjunction of premises in a Horn clause.

Example 2. We present how the *length* function for lists and a precondition containing this function are turned into relational expressions by flattening. The *length* of a list is defined by $\mathit{length}\ [] = 0$, and $\mathit{length}\ (x \cdot xs) = \mathit{Suc}\ (\mathit{length}\ xs)$[3] . We derive a corresponding relation length_P with two Horn clauses:

$$\mathit{length}_P\ []\ 0$$
$$\mathit{length}_P\ xs\ n \Longrightarrow \mathit{length}_P\ (x \cdot xs)\ (\mathit{Suc}\ n)$$

The precondition $\mathit{length}\ xs = \mathit{length}\ ys$ is then transformed into

$$\mathit{length}_P\ xs\ n \wedge \mathit{length}_P\ ys\ n$$

In the new formulation, the constraint of the two lists having the same length is expressed by their shared variable n. This relational description helps our mode analysis to find a more precise data flow.

This well-known technique of flattening is similarly described by Naish [15] and Rouveirol [18]. We also support flattening of higher-order functions, which allows inversion of higher-order functions if the function argument is invertible.

5 Mode analysis

In order to execute a predicate P, its arguments are classified as *input* or *output*, made explicit by means of *modes*. Modes can be inferred using a static analysis on the Horn clauses. Our mode analysis is based on Mellish [14]. There are more sophisticated mode analysis approaches, e.g., by using abstract domains [20] or by translating to a boolean constraint system [17]. But for our purpose, we can apply the simple mode analysis, because if the analysis does not discover a dataflow due to its imprecision, the overall process still leads to a test data generator.

[3] Natural numbers are defined by constructors 0 and *Suc*.

Modes. For a predicate P with k arguments, a *mode* is a particular dataflow assignment which follows the type of the predicate and annotates all arguments as input (i) or output (o), e.g., for $length_P$, $o \Rightarrow i \Rightarrow bool$ denotes the mode where the first argument is output, the last argument is input.
A *mode assignment* for a given clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \bar{t}$ is a list of modes $M, M_1, \ldots M_n$ for the predicates $P, Q_1, \ldots, Q_n$. Let $FV(t)$ denote the set of free variables in a term t. Given a vector of arguments $\bar{t}$ and a mode M, the projection expression $\bar{t}\langle M\rangle$ denotes the list of all arguments in $\bar{t}$ (in the order of their occurrence) which are input in M.

Mode consistency. Given a clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \bar{t}$ a corresponding mode assignment $M, M_1, \ldots M_n$ is *consistent* if the chain of sets of variables $v_0 \subseteq \cdots \subseteq v_n$ defined by **(1)** $v_0 = FV(\bar{t}\langle M\rangle)$ and **(2)** $v_j = v_{j-1} \cup FV(\overline{u}_j)$ obeys the conditions **(3)** $FV(\overline{u}_j\langle M_j\rangle) \subseteq v_{j-1}$ and **(4)** $FV(\bar{t}) \subseteq v_n$. Mode consistency guarantees the possibility of a sequential evaluation of premises in a given order, where v_j represents the known variables after the evaluation of the j-th premise. Without loss of generality, we can examine clauses under mode inference modulo reordering of premises. For side conditions R, condition 3 has to be replaced by $FV(R) \subseteq v_{j-1}$, i.e., all variables in R must be known when evaluating it. This definition yields a check whether a given clause is consistent with a particular mode assignment.

Generator mode analysis. To generate values that satisfy a predicate, we extend the mode analysis in a genuine way: If the mode analysis cannot detect a consistent mode assignment, i.e., the values of some variables are not *constrained* after the evaluation of the premises, we allow the use of *generators*, i.e., the values for these variables are constructed by an *unconstrained* enumeration. In other words, we combine two ways to enumerate values, either driven by the *computation* of a predicate or by *generation* based on its type.

Example 3. Given a unary predicate R with possible modes $i \Rightarrow bool$ and $o \Rightarrow bool$ and the Horn clause $R\ x \Longrightarrow P\ x\ y$, classical mode analysis fails to find a consistent mode assignment for P with mode $o \Rightarrow o \Rightarrow bool$. To generate values for x and y fulfilling P, we combine computation and generation of values as follows: the values for variable x are built using R with $o \Rightarrow bool$; values for y are built by a generator.

This extension gives rise to a number of possible modes, because we actually drop the conditions **(3)** and **(4)** for the mode analysis. Instead, we use a heuristic to find a considerably good dataflow by locally selecting the optimal premise Q_j and mode M_j with respect to the following criteria:

1. minimize missing values, i.e., have $\left|FV(\overline{u}_j\langle M_j\rangle) - v_{j-1}\right|$ to be minimal;
2. use functional predicates with their functional mode;
3. use predicates and modes that do not require generators themselves;
4. minimize number of output positions;
5. prefer recursive premises.

Next, we motivate and illustrate these five criteria. In general, we would like to avoid generation of values and computations that could fail, and to restrain ourselves from enumerating any values that could possibly be computed. Hence, the first priority is to use modes where the number of missing values is minimal. This way, we partly recover conditions **(3)** and **(4)** from the mode analysis.

Example 3 (continued). For mode M_1 for $R\ x$, one has two alternatives: generating values for x and then testing R with mode $i \Rightarrow bool$, or only generating values for x using R with $o \Rightarrow bool$. The first choice generates values and rejects them by testing; the latter only generates fulfilling values and is preferable. The analysis favors $o \Rightarrow bool$ to $i \Rightarrow bool$ due to criterion 1: for $v_0 = \{\}$, $\overline{u}_1 = x$ and $M_1 = i \Rightarrow bool$, $FV(\overline{u}_1 \langle M_1 \rangle) - v_0 = \{x\}$; whereas for $M_1 = o \Rightarrow bool$, $FV(\overline{u}_1 \langle M_1 \rangle) - v_0 = \{\}$. $|FV(\overline{u}_1 \langle M_1 \rangle) - v_0|$ is minimal for $M_1 = o \Rightarrow bool$.

Example 4. Consider a clause $R\ x\ y \Longrightarrow F\ x\ y \Longrightarrow P\ x\ y$ where R is a one-to-many relation and F is functional. R and F both allow modes $i \Rightarrow o \Rightarrow bool$ and $i \Rightarrow i \Rightarrow bool$. For $M = i \Rightarrow o \Rightarrow bool$, $R\ x\ y$ and $F\ x\ y$ can be evaluated in either order. Our criterion 2 induces preference for computing y with the functional computation $F\ x\ y$ and checking $R\ x\ y$, i.e., whether the one value for y can fulfill $R\ x\ y$ or not.

Criterion 3 induces avoiding the generation of values in the predicate to be invoked. Furthermore, we minimize output positions, e.g., we prefer checking a predicate (no output position) before computing some solution (one output position) as we illustrate by the following example:

Example 5. In a clause $R\ x\ y \Longrightarrow Q\ x \Longrightarrow P\ x\ y$ with mode $i \Rightarrow o \Rightarrow bool$ for R and P, and $i \Rightarrow bool$ for Q, we prefer $Q\ x$ before $R\ x\ y$, since computing values for y would be useless if $Q\ x$ fails. This ordering is enforced by criterion 4.

Finally, we prefer recursive premises – this leads to a bottom-up generation of values. Generating larger values for predicates from smaller values for the predicate is commonly preferable because it takes advantage of the structure of the preconditions.

Example 6. In a clause $P\ xs \Longrightarrow C\ xs \Longrightarrow P\ (x \cdot xs)$, $P\ xs$ is favored for generation of xs and $C\ xs$ for checking. Generating values for P, we apply the generator for P recursively and check the condition $C\ xs$ afterwards.

This "aggressive" mode analysis results in moded Horn clauses with annotations for generators of values. In summary, it does not only *discover* an existing dataflow, but helps to *create* a dataflow by filling the gaps with value generators.

6 Generator compilation

In this section, we discuss the translation of the compiler from moded Horn clauses to functional programs. First, we present the building blocks of the compiler, the execution mechanism and the generators. Then, we sketch the compilation scheme by applying it to the introductory examples.

Monads for non-deterministic computations. We use continuations with type $\alpha\ cps$ to enumerate the (potentially infinite) set of values fulfilling the involved predicates – in other words, the constructed continuations will hold the *enumerated solutions*. We define *plus monad* operations describing non-deterministic computations. Depending on our enumeration scheme, we employ three different plus monads: one for unbounded computations, and two others for depth-limited computations within positive and negative contexts, respectively.

A plus monad supports four operations: *empty*, *single*, *plus* and *bind*. It provides executable versions of basic set operations: $\mathit{empty} = \emptyset$, $\mathit{single}\ x = \{x\}$, $\mathit{plus}\ A\ B = A \cup B$ and $\mathit{bind}\ A\ f = \bigcup_{x \in A} f\ x$. Employing these operations in SML results in a Prolog-like execution strategy, with a depth-first search. This strategy is fine for user-initiated evaluations, but for counterexample generation, automatically generated values cause infinite computations escaped from the control of the user. To avoid being stuck in such a computation, we also employ a plus monad with a different carrier that limits the computation by a depth-limit. Evaluating predicates with a depth-limited computation, we must take special care of negation. We implement different behaviors for queries in different contexts: for positive contexts, we compute an underapproximation; for negative contexts, an overapproximation.

For positive contexts, we implement a plus monad with the type $\mathit{int} \to \alpha\ \mathit{cps}$ as carrier. The bind^{+} operation checks the depth-limit and if reached, returns *empty*, which yields a sound underapproximation; otherwise it passes a decreased depth-limit to its argument. It is defined by:

$$\mathit{bind}^{+}\ xq\ f = (\lambda i.\ \mathit{if}\ i = 0\ \mathit{then\ empty\ else\ bind}\ (xq\ (i-1))\ (\lambda a.\ f\ a\ i))$$

In negative contexts, we must explicitly distinguish failure (no solution found) from reaching the depth limit. To signal reaching the depth-limit, we include an explicit element to model an *unknown* value (as a third truth value), and continue the computation with this value. This makes the monad carrier type be $\mathit{int} \to \alpha\ \mathit{option}\ \mathit{cps}$ where the option value *None* stands for unknown. If one computation reaches the depth-limit and another computation fails, then the overall computation fails; in other words *failure absorbs the unknown value* (which is consistent with a three-valued logic interpretation).

Because negative and positive occurrences of predicates are intermixed, in actual enumeration we have to combine the positive and negative monads – the bridge between them is performed by executable *not*-operations that handle the unknown value depending on the context. For instance, when applied to a solution enumeration of a negated premise, *unknown* is mapped to *false* (computation failure); this reflects the intuition that if we were not able to prove a negated premise $\neg Q\ x$ within a given depth-limit for x, then all we can soundly assume is that $Q\ x$ may hold; hence the computation cannot proceed further.

The compilation scheme builds abstractly on the monad structure interface and hence is employed for all three monads. For the rest of the presentation, we write *plus* and *bind* infix as $\sqcup$ and $\gg\!=$.

Type-based generators. If values cannot be computed, we enumerate them up to a given depth. To generate values of a specific type, we make use of type classes in Isabelle. More specifically we require that the involved types τ come equipped with an operation $\mathit{gen}\ \tau$, the generator for type τ that enumerates all values. For inductive datatypes τ with n constructors $C_1\ \tau_1^1 \ldots \tau_1^{m_1}\ | \ldots |\ C_n\ \tau_n^1 \ldots \tau_n^{m_n}$ we construct generators that enumerate values exhaustively up to depth d by the following scheme:

$$\begin{array}{l}
\mathit{gen}\ \tau\ d = \\
\quad \mathit{if}\ d = 0\ \mathit{then\ empty\ else} \\
\quad\quad (\mathit{gen}\ \tau_1^1\ (d-1) \gg\!= (\lambda x^1.\ \mathit{gen}\ \tau_1^2\ (d-1) \gg\!= \ldots \gg\!= (\lambda x^{m_1-1}. \\
\quad\quad\quad \mathit{gen}\ \tau_1^{m_1}\ (d-1) \gg\!= (\lambda x^{m_1}.\ \mathit{single}\ (C_1\ x^1 \ldots x^{m_1}))) \ldots)) \sqcup \ldots \sqcup \\
\quad\quad (\mathit{gen}\ \tau_n^1\ (d-1) \gg\!= (\lambda x^1.\ \mathit{gen}\ \tau_n^2\ (d-1) \gg\!= \ldots \gg\!= (\lambda x^{m_n-1}. \\
\quad\quad\quad \mathit{gen}\ \tau_n^{m_n}\ (d-1) \gg\!= (\lambda x^{m_n}.\ \mathit{single}\ (C_n\ x^1 \ldots x^{m_n}))) \ldots))
\end{array}$$

We already have seen concrete instances of these generators for lists and natural numbers, *generate-list* and *generate-nat* in §2 – although there, the scheme is disguised by the fact that we inlined the plus monad operations.

Compilation of moded clauses. The central idea underlying the compilation of a predicate P is to generate a function P^M for each mode M of P that, given a list of input arguments, enumerates all tuples of output arguments. The functional equation for P^M is the union of the output values generated by the characterizing clauses. Employing the data flow from the mode inference, the expressions for the clauses are essentially constructed as chains of type-based generators and function calls for premises, connected through *bind* and *case* expressions. All functions P^M are executable in ML, because they only employ the monad operations and pattern matching. The function P^M for the mode M with all arguments as output serves as test data generator for predicate P.

Example 7. For the predicate *distinct*, we can infer the mode $o \Rightarrow bool$: The first clause *distinct* [] allows the mode $o \Rightarrow bool$, as the empty list is just a constant value. The second clause allows the mode $o \Rightarrow bool$ by choosing modes for its premises, i.e., *distinct* xs with mode $o \Rightarrow bool$ and $x \notin \mathit{set}\ xs$ with mode $i \Rightarrow i \Rightarrow bool$. This is then compiled to a test data generator $\mathit{distinct}^o$ for lists of type τ:

$$\begin{array}{l}
\mathit{distinct}^o\ \tau = \mathit{single}\ [\,]\ \sqcup \\
(\mathit{distinct}^o \gg\!= (\lambda xs.\ \mathit{gen}\ \tau\ \gg\!= (\lambda x.\mathit{if}\ x \notin \mathit{set}\ xs\ \mathit{then\ single}\ (x \cdot xs)\ \mathit{else\ empty})))
\end{array}$$

Instantiating τ to the natural numbers and unfolding the plus monad operators, the definition of $\mathit{distinct}^o$ yields the test data generator *generate-distinct* from section 2.

Example 8. For the precondition $\mathit{length}\ xs = \mathit{length}\ ys \wedge \mathit{zip}\ xs\ ys = zs$, we obtain the following moded clause:

- $\mathit{length}_P\ xs\ n$ with mode $o \Rightarrow o \Rightarrow bool$,
- $\mathit{length}_P\ ys\ n$ with mode $o \Rightarrow i \Rightarrow bool$,
- $\mathit{zip}_P\ xs\ ys\ zs$ with its functional mode $i \Rightarrow i \Rightarrow o \Rightarrow bool$

In other words, we enumerate lists with their corresponding length, and as we know the length of xs, we only enumerate lists ys of equal length, and finally we obtain zs by executing $\mathit{zip}\ xs\ ys$. The generator for this precondition then is:

$$\begin{array}{l}
{\mathit{length}_P}^{oo} \gg\!= (\lambda(xs,n).\ {\mathit{length}_P}^{oi}\ n \\
\gg\!=(\lambda ys.\ \mathit{single}\ (\mathit{zip}\ xs\ ys) \gg\!= (\lambda zs.\ \mathit{single}\ (xs,ys,zs))))
\end{array}$$

	size									
predicate	5	6	7	8	9	10	11	12	13	14
–	24	89	425	2,373	16,072	125,673	1,112,083	10,976,184	119,481,296	1,421,542,641
distinct	16	39	105	315	1,048	3,829	15,207	65,071	297,840	1,449,755
sorted	15	31	63	127	255	511	1,023	2,047	4,095	8,191

Table 1. Number of test cases for given sizes and preconditions

Unfolding the definitions of the plus monad operators and reducing the syntactic clutter, this leads to

$$\begin{array}{l} \textit{if } d = 0 \textit{ then None} \\ \textit{else length}_P{}^{oo}\ (d-1)\ (\lambda(xs, n).\ \textit{length}_P{}^{oi}\ n\ (d-1)\ (\lambda ys.\ c\ xs\ ys\ (\textit{zip}\ xs\ ys))) \end{array}$$

The arguments c and d make the continuation and the limit on the depth of the computation explicit. The monad operations implicitly pass around the values for c and d.

7 Evaluation

To evaluate our approach, we compared the performance of the new approach against the three other existing testing approaches in Isabelle: random, exhaustive and narrowing-based testing. Random and exhaustive testing employ concrete values, whereas narrowing-based testing employs symbolic values. The narrowing-based testing in Isabelle is a descendant of Lazy SmallCheck, employing the same evaluation mechanism.

First, we compare their performance validating conjectures with simple preconditions. Table 1 shows the number of test cases up to a given size, and the number of test cases (for that size) for which the preconditions *distinct* and *sorted* hold. In other words, we measured the density of the search space if restricted by some precondition, compared to the unrestricted search space. For example, testing the proposition $\textit{distinct}\ xs \Longrightarrow \textit{distinct}\ (\textit{tl}\ xs)$, the table shows how many test cases are generated by the naive exhaustive testing and by the smart test generators. This already gives a rough estimate on the possible improvement avoiding useless tests. Table 2 shows the run time[4] to validate properties with values up to a given size on some representative conjectures from Isabelle's library with the precondition *distinct* (D_1, D_2, D_3) and *sorted* (S_1, S_2, S_3):

- D_1: $\textit{distinct}\ xs \Longrightarrow \textit{distinct}\ (\textit{tl}\ xs)$
- D_2: $\textit{distinct}\ xs \Longrightarrow \textit{distinct}\ (\textit{remove1}\ x\ xs)$
- D_3: $\textit{distinct}\ xs \Longrightarrow \textit{distinct}\ (\textit{zip}\ xs\ ys)$
- S_1: $\textit{sorted}\ xs \Longrightarrow \textit{sorted}\ (\textit{remdups}\ xs)$
- S_2: $\textit{sorted}\ xs \Longrightarrow \textit{sorted}\ (\textit{insort-insert}\ x\ xs)$
- S_3: $\textit{sorted}\ xs \wedge i \leq j \wedge j < \textit{length}\ xs \Longrightarrow \textit{nth}\ xs\ i \leq \textit{nth}\ xs\ j$

The numbers of D_1 indicate the improvement using the smart test generators for *distinct*. In case of D_2, a more representative conjecture of the Isabelle's theory of lists, we observe a similar behaviour. In D_3, the exhaustive testing does not enumerate all pairs of

[4] All tests ran on a Pentium DualCore P9600 2.6GHz with 4GB RAM using Poly/ML 5.4.1 and Ubuntu GNU/Linux 11.04

		size																	
		8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
D_1	E	0	0	0	0.3	3.2	38	509											
	N	0	0.1	0.4	3.5	32	364												
	S	0	0	0	**0**	**0.2**	**0.7**	**3.8**	**22**	**135**	**862**								
D_2	E	0	0	0	0.4	3.8	45	589											
	N	0	0.1	0.5	4.0	37	395												
	S	0	0	0	**0.1**	**0.4**	**2.5**	**16**	**98**	**671**									
D_3	E	0.1	4.3	**155**															
	N	0.9	17	446															
	S	0.1	4.3	157															
S_1	E	0	0	0	0.2	2.7	31	404											
	N	0	0	0	0.1	0.1	0.1	0.2	0.4	0.9	2.0	4.6	10	23	52	115	257	565	1238
	S	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0.1**	**0.2**	**0.3**	**0.8**	**1.7**	**3.6**	**7.8**	**17**	**36**
S_2	E	0	0	0	0.2	2.5	29	381											
	N	0	0.1	0.1	0.1	0.1	0.2	0.4	0.8	1.8	3.9	8.8	20	44	98	218	286	1063	
	S	0	0	0	**0**	**0**	**0**	**0.1**	**0.1**	**0.2**	**0.5**	**1.1**	**2.5**	**5.5**	**12**	**28**	**61**	**135**	**292**
S_3	E	0	0	0	0.2	2.3	27	337											
	N	0	0	0.1	0.1	0.2	0.5	1.3	2.9	6.9	16	38	87	204	467	1064			
	S	0	0	0	**0**	**0**	**0.1**	**0.1**	**0.2**	**0.4**	**0.9**	**2.2**	**5.1**	**12**	**26**	**59**	**136**	**311**	**708**

Table 2. Run time in seconds for given sizes – E, N, S denote exhaustive testing, narrowing, and smart generators, resp.; 0 denotes time < 50 ms, empty cells denote timeout after 1h; bold numbers indicate the lowest run time

lists for *xs* and *ys*, but only generates lists *ys* if the generated list *xs* is distinct. This simple optimisation already reduces the number of tests dramatically, i.e., only 0.025 percent of all tests are rejected by the precondition. Due to this fact, using the smart generator does not add any further significant improvement in the run time behaviour. Hence our smart generators perform practically the same to the exhaustive testing. Symbolic execution with narrowing performs worst due to its overhead in the execution in all three cases. On the very sparse precondition, *sorted xs*, the improvements with smart test data generators are even more apparent. For example, in S_1, naive exhaustive testing times out at size 15 (with a time limit of one hour), where the smart generators can still enumerate lists up to size 20 within a second. Narrowing performs better than exhaustive testing, but is still slower than the smart generators. These numbers show that the test data generators outperform the naive exhaustive testing and the symbolic narrowing-based testing.

Second, to show that this performance improvement also results in a direct gain for our users, we apply the counterexample generators on faulty implementations of typical functional data structures. We injected faults by adding typos into the correct implementations of the delete operation of 2-3 trees, AVL trees, and red-black 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 the tree balanced and ordered, i.e., $\mathit{balanced}\ t \Longrightarrow \mathit{balanced}\ (\mathit{delete}\ k\ t)$, and $\mathit{ordered}\ t \Longrightarrow \mathit{ordered}\ (\mathit{delete}\ k\ t)$. With the 10 versions, this yields 20 tests, on which we apply the different counterexample generators. Random testing (with 2,000 iterations for each size) finds errors in 5, and exhaustive testing in 7 of 20 tests within thirty seconds. The smart generator finds errors in five more cases, uncovering 12 errors in the 20 tests; the narrowing approach performs equally well. In principle, exhaustive testing should find the errors eventually: so, on the five more intrinsic cases where the generators perform well, we increased the time for naive exhaustive testing to finally discover the fault – even after one hour of testing, exhaustive testing was not able to detect them. Also increasing the iterations for random testing to 20,000 iterations, it still discovers only five faults. This shows that using the test data generators in this case is clearly superior to naive exhaustive testing. In the eight cases, where all approaches found no fault, 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.

On AVL trees, we observe a similar behaviour. When checking the two invariants of its delete operation on 10 modified versions, random testing uncovers 5, exhaustive testing 6, the smart generators and narrowing-based approach 11 errors in 20 cases. On red-black trees, the invariant was formulated in a way by the user that our data flow analysis cannot discover a reasonable ground data flow and therefore the synthesized generators perform very poorly. Here, the narrowing-based testing clearly benefits from its usage of symbolic values.

Beyond data structures, we also check a hotel key card system in Isabelle by Nipkow [16] which itself was inspired by a model from Jackson [12]. The faulty system contains a tricky man-in-the-middle attack, which is only uncovered by a trace of length 6. The formalisation uses a restrictive predicate that describes in which order specific events can occurs. Using the smart generators, we can find the attack within a few seconds. Synthesizing a test data generator for these valid traces requires the preprocessing techniques (§4), i.e., we can eliminate existential quantifiers, which render it non-executable for the random and exhaustive testing. Even after manual refinements to obtain an executable reformulation, random and exhaustive testing fail to find the counterexample within ten minutes of testing. The narrowing-based testing can handle the existential quantifiers in principle, but practically it performs badly with the deeply nested existential quantifiers in the specification, rendering it impossible to find the counterexample. After manual rewriting to eliminate the existentials, we also can obtain a counterexample with this approach within a few seconds.

8 Conclusion

This counterexample generator described in this paper is included in the current Isabelle development version and can be invoked by Isabelle's users to validate their specifications before proving them correct. It complements the existing naive exhaustive and narrowing-based testing techniques by combining the strengths of both: it reduces the number of tests, as narrowing-based testing does, and it executes tests very fast, as the naive exhaustive testing does.

Acknowledgements I would like to thank Andrei Popescu, Sascha Boehme, Tobias Nipkow, Alexander Krauss, Thomas Tuerk, Brian Huffman, Jasmin Blanchette and the anonymous referees for comments on earlier versions of this paper.

References

1. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM 47, 776–822 (2000)
2. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: TPHOLs '09. LNCS, vol. 5674. Springer (2009)
3. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: SEFM '04, pp. 230–239. IEEE Computer Society (2004)
4. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: ITP '10. LNCS, vol. 6172, pp. 131–146. Springer (2010)
5. Bulwahn, L.: Smart test data generators via logic programming. In: ICLP '11 (Technical Communications). Leibniz Int. Proc. in Informatics, vol. 11, pp. 139–150. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2011)
6. Carlier, M., Dubois, C., Gotlieb, A.: Constraint Reasoning in FocalTest. In: ICSOFT '10. (2010)
7. Christiansen, J., Fischer, S.: EasyCheck – Test Data for Free. In: FLOPS '08. LNCS, vol. 4989, pp. 322–336. Springer (2008)
8. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP '00, pp. 268 – 279. ACM SIGPLAN (2000)
9. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP '07, pp. 63–74. ACM (2007)
10. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS '10. LNCS, vol. 6009, pp. 103–117. Springer (2010)
11. Hanus, M.: Multi-paradigm declarative languages. In: ICLP '07. LNCS, vol. 4670, pp. 45–75. Springer (2007)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
13. Lindblad, F.: Property directed generation of first-order test data. In: The Eigth Symposium on Trends in Functional Programming. (2007)
14. Mellish, C.S.: The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence (1981)
15. Naish, L.: Adding equations to NU-Prolog. In: 3rd Int. Sym. on Programming Language Implementation and Logic Programming. LNCS, pp. 15–26. Springer (1991)
16. Nipkow, T.: Verifying a Hotel Key Card System. In: ICTAC '06. LNCS, vol. 4281. Springer (2006) Invited paper.
17. Overton, D., Somogyi, Z., Stuckey, P.J.: Constraint-based mode analysis of mercury. In: PPDP '02, pp. 109–120. ACM (2002)
18. Rouveirol, C.: Flattening and Saturation: Two Representation Changes for Generalization. Mach. Learn. 14(2), 219–232 (1994)
19. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In: Haskell '08, pp. 37–48. ACM (2008)
20. Smaus, J.G., Hill, P.M., King, A.: Mode analysis domains for typed logic programs. In: Sel. papers from the 9th Int. Workshop on Logic Programming Synthesis and Transformation, pp. 82–101. Springer (2000)
21. Weber, T.: Bounded model generation for Isabelle/HOL. In: PDPAR '04. Electronic Notes in Theoretical Computer Science, vol. 125(3), pp. 103–116. Elsevier (2005)
22. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: TPHOLs '08. LNCS, vol. 5170, pp. 33–38. Springer (2008)