Smart test data generators via logic programming*

Lukas Bulwahn

Fakultät für Informatik
Technische Universität München
bulwahn@in.tum.de

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. Standard ML, OCaml) 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 by an enriched mode inference. From this inference, the compiler can construct the desired generators in the functional programming language. These test data generators are applied to find errors in specifications, as we show in a case study of a hotel key card system.

1998 ACM Subject Classification D.2.5 Testing and Debugging

Keywords and phrases specification-based testing, functional programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.139

1 Introduction

Writing programs and specifications is an error-prone business and testing is common practice to find bugs and to validate software. Being aware that testing cannot prove the absence of bugs, 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 absence of bugs is certified. But in the process of proving, bugs could still be revealed and tracking down such bugs by failed proof attempts is a tedious task for the user. When reaching for the "holy grail", the formal proof, testing is still fruitful on the way to save time detecting bugs in programs and specifications. Modern interactive theorem provers therefore do not only provide means to prove properties, but also to *disprove* properties by 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 tool QuickCheck [8]. The interactive theorem prover Isabelle [27] provides a counterexample generator [2] based on random testing, similar to QuickCheck. It works fine on specifications that have weak preconditions and properties stated in a form to be directly executable in the functional language. If the properties to be tested only hold under very specific preconditions, test data with a random distribution seldom fulfill the preconditions, and most execution time for testing is spent generating useless test values and rejecting them.

* This work was partially supported by DFG doctorate program 1480 (PUMA).

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 139–150

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our new approach aims to only generate test data that fulfill the preconditions. The test data generator for a given precondition is produced by a compiler[1] that analyzes precondition and synthesizes a purely functional program that serves as generator. For this purpose, the compiler reformulates the preconditions as logic programs. In this process, we adopt various techniques from logic programming. Formulas in predicate logic with quantifiers and recursive functions are translated to logic programs. The compiler then analyzes the logic program by an enriched mode inference. From this inference, the compiler can construct the desired generators. This way, a much smaller number of test cases suffices to find errors in specifications.

We introduce the Isabelle system and illustrate how Isabelle's users interactively explore proofs (Section 2). We present a concrete case study for the counterexample generator (Section 3). In the main part, we then describe key ideas of this counterexample generator, the preprocessing, enriched mode inference and compilation (Sections 4 to 7). In the end, we discuss related work (Section 8) and conclude.

2 Interactive Theorem Prover Isabelle

The Isabelle system is a generic framework for interactive theorem proving. Isabelle's logic is a higher-order logic with Hindley-Milner polymorphism. It provides the usual boolean operations, functional programming constructs, e.g., *let*, *if* and *case* expressions, and mechanisms to define recursive datatypes, recursive functions, and inductive predicates by Horn clauses. The code generation facility creates programs in functional programming languages, e.g., ML, Haskell or OCaml, from their specifications in the theorem prover.

Users of interactive theorem prover intend to construct a machine-checked proof in cooperation with the theorem prover. After stating a proposition, the system invokes automated proof methods and counterexample generators. If the user's proposition is proved automatically, its validity is certified by the system. If a counterexample is found, the user must refine the proposition. If neither happens, the user continues to explore the proof by stating further propositions. In a horizontal exploration, the user believes its truth, but does not prove it formally. Instead, he sketches further lemmas and proof steps based on the skipped proposition.

If the counterexample generators were not be in place, the horizontal exploration would have a major pitfall: With no counterexample generator, the statement is unchecked, but the user continues to develop the proof upon this wrong assertion. Realizing the flaw at a later stage requires much effort restructuring the proof. Counterexample generators are vital to spare the user this frustration and time-consuming work of unsuccessful proof attempts.

3 Case Study: Hotel Key Card System

Most hotels employ some kind of digital key card system. We describe a hotel key card system where every room is secured by a digital lock. Every guest of the hotel receives a card at the reception. The locks at the rooms can read the cards from the guest, and open the door if it is the card of the owner. The key card system is *decentralized*, i.e., the locks cannot communicate with each other or the reception. Nevertheless, only guests that check

[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.

in for a room should be allowed to enter, and previous guests should not have further access to the room once they checked out.

Safety is achieved by the following system: Upon check-in, a new guest gets a card at the reception which carries two keys, the old key of the previous guest of the room and his own new key. The locks only store their current key, i.e., the key of the latest guest the lock has been aware of. When a new guest enters the room, the lock checks if the old card key matches its current key, and if so discards the old key and stores the new key as its new current key. Once the lock has been recoded, it allows access only to the card with the current key until the next guest enters the room. This recoding ensures that the previous guest cannot enter the room after the new guest has been in his room.

Nipkow [17] gives a formalization of this hotel key card system in Isabelle which itself was inspired by a model from Jackson [13]. The safety property of the hotel key card system is: Once the owner of the room, i.e., the guest who was the last to check in, entered his room, no previous guest can enter the room (even if they have kept or copied their cards). Unfortunately, this property does not hold. But fortunately, the counterexample generator finds the tricky man-in-the-middle attack.

In Isabelle, the hotel key card system is formalized as follows: We consider three events, a guest g checking in for a room r where he gets a card with keys (k, k') from the reception, a guest g entering a room r with a card with keys (k, k'), or a guest g leaving a room r. We denote these events as *Checkin* g r (k, k'), *Enter* g r (k, k'), and *Exit* g r. A trace, represented as a list of events, describes the temporal order of events taking place. Without going into details here, the set of possible and valid traces in a hotel and safety is given by the following functional description in Isabelle:

$$
\begin{aligned}
&\mathit{hotel}\ [] = \mathit{True}\\
&\mathit{hotel}\ (e \cdot \mathit{evs}) = (\mathit{hotel}\ \mathit{evs} \wedge (\mathit{case}\ e\ \mathit{of}\\
&\quad \mathit{Checkin}\ g\ r\ (k, k') \Rightarrow k = \mathit{currentkey}\ \mathit{evs}\ r \wedge k' \notin \mathit{issued}\ \mathit{evs}\ |\\
&\quad \mathit{Enter}\ g\ r\ (k, k') \Rightarrow (k, k') \in \mathit{cards}\ \mathit{evs}\ g \wedge (\mathit{roomkey}\ \mathit{evs}\ r \in \{k, k'\})\ |\\
&\quad \mathit{Exit}\ g\ r\ \ \Rightarrow g \in \mathit{isin}\ \mathit{evs}\ r))\\
&\mathit{safe}\ \mathit{evs}\ r =\\
&\quad \exists \mathit{evs}_1\ \mathit{evs}_2\ \mathit{evs}_3\ g\ c\ c'.\ \mathit{evs} = \mathit{evs}_3\ @\ (\mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2\ @\ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1) \wedge\\
&\quad \mathit{noCheckin}\ (\mathit{evs}_3\ @\ \mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2) \wedge \mathit{isin}\ (\mathit{evs}_2\ @\ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1) = \emptyset
\end{aligned}
$$

where $\mathit{noCheckin}\ \mathit{evs}\ r\ = \neg(\exists g\ c.\ \mathit{Checkin}\ g\ r\ c \in \mathit{evs})$ and @ denotes appending two lists.

The safety property is formally $\mathit{hotel}\ \mathit{evs} \wedge \mathit{safe}\ \mathit{evs}\ r \wedge g \in \mathit{isin}\ \mathit{evs}\ r \Longrightarrow \mathit{owner}\ \mathit{evs}\ r = g$. When checking the validity of this property, the existing counterexample generator for Isabelle [2], based on the ideas of QuickCheck, faces two problems:

Firstly, naive black-box testing would generate traces where most traces do not fulfill the necessary conditions to be a valid hotel key card trace, i.e., *hotel evs* evaluates to false for most traces *evs*. The common approach is to write a manual generator for the predicate *hotel*. But the functional description already contains all necessary information on how to construct values that fulfill the predicate *hotel*, which is *not* exploited by black-box testing.

Secondly, a predicate, such as *safe* cannot be (naively) executed because it contains unbounded existential quantifiers (over an infinite type) for evs_1, evs_2, evs_3. However, having a closer look at the description of *safe*, we see that the values for evs_1, evs_2 and evs_3 are actually restricted to be parts of the trace *evs*, which could be computed given *evs*.

The system we describe tackles the two problems, generating test data that fulfills the precondition, and detecting for which quantifier the values are bound within the computation, using an enriched mode analysis. For the hotel key card example, the new approach allows us to find the man-in-the-middle attack within a few seconds, whereas the black-box testing does not find the counterexample even after ten minutes of testing.

4 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.

Design decisions. The Isabelle system is implemented in ML and runs in the interactive shell of the ML compiler. ML source code generated from Isabelle specifications can be passed to the underlying ML compiler, and executed in the same process.

As Isabelle runs on top of the ML compiler, it is a natural choice to test specifications in the underlying programming language ML as it adds no further system requirements and dependencies. ML as a strict functional language does not support natively the common execution mechanisms from logic programming, such as non-determinism and logic variables in terms. If we want to use these mechanisms, we must *embed them in the functional language*. But the execution in an embedding is considerably slower than native execution. We aim to find a balance between *deep embedding* and *fast execution*.

We only allow values to be *ground* during the execution, which is the native setting for purely functional languages. In other words, we do not allow logic variables in terms or partially instantiated terms. Allowing such terms would require that even purely functional specifications be embedded deeply as well, causing a tremendous overhead for purely functional execution. Restricting ourselves to ground terms has the advantage that functional specifications can be translated directly into the target functional programming language. The test data generators only construct proper ground values in the functional programming language. Only parts of the specification, i.e., predicates occurring in preconditions, are compiled to test data generators with an embedding of nondeterministic execution whereas the testing of the conclusion can be done via the fast direct execution mechanism of the functional language. E.g., consider the safety property of the hotel key card system, the set of values for evs, r, and g for the preconditions, *hotel* evs, *safe* evs and *isin* $evs\ r$, are computed with the deep embedding, the conclusion, *owner* $evs\ r = g$, is executed as functional program directly.

The decision above burdens the compilation with a static analysis – the mode analysis – to determine a possible dataflow of ground values in the description of the precondition. However, the advantage of smarter test data generators is worth this burden. The test data generators commonly return a *set* of values – we implement this behavior using lazy sequences in ML.

In summary, our system compiles predicate preconditions to smart test data generators using ground terms, a dataflow analysis and nondeterministic execution. The conclusion is tested via direct functional execution.

Architecture. The counterexample generator performs the these steps: As the original specification can be defined using various definitional mechanisms, the specification is preprocessed by a few simple syntactic transformations (Section 5) to Horn clauses. The core component, which was previously described in [1], consists of the mode analysis (Section 6) and the code generator (Section 7). 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 lazy sequences and type-based generators.

5 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 different form are transformed in 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 sub formulas, as our Horn clauses do not allow disjunctions within the premises and nested expressions under negations. Furthermore, we take special care of *if*, *case* and *let*-constructions.

▶ **Example 1.** The predicate *hotel* is processed to a system of Horn clauses with predicates *hotel*, $hotel_{aux}$ and $hotel_{aux2}$; the latter two are introduced during the preprocessing:

$$
\begin{array}{l}
hotel\ [] \\
hotel\ evs \Longrightarrow hotel_{aux}\ e\ evs \Longrightarrow hotel\ (e \cdot evs) \\
k = currentkey\ evs\ r \Longrightarrow k' \notin issued\ evs \Longrightarrow hotel_{aux}\ (Checkin\ g\ r\ (k, k'))\ evs \\
(k, k') \in cards\ evs\ g \Longrightarrow hotel_{aux2}\ evs\ r\ k\ k' \Longrightarrow hotel_{aux}\ (Enter\ g\ r\ (k, k'))\ evs \\
g \in isin\ evs\ r \Longrightarrow hotel_{aux}\ (Exit\ g\ r)\ evs \\
roomkey\ evs\ r\ k \Longrightarrow hotel_{aux2}\ evs\ r\ k\ k' \\
roomkey\ evs\ r\ k' \Longrightarrow hotel_{aux2}\ evs\ r\ k\ k'
\end{array}
$$

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.** Consider the formula $evs = evs_3\ @\ (Enter\ g\ r\ c \cdot evs_2\ @\ Checkin\ g\ r\ c' \cdot evs_1)$ used in the predicate *safe*. This formula is flattened to two premises,

$$append_P\ evs_2\ (Checkin\ g\ r\ c' \cdot evs_1)\ r_1 \text{ and } append_P\ evs_3\ (Enter\ g\ r\ c \cdot r_1)\ evs,$$

and $append_P$ is defined by two Horn clauses derived from its functional definition:

$$append_P\ []\ ys\ ys \text{ and } append_P\ xs\ ys\ zs \Longrightarrow append_P\ (x \cdot xs)\ ys\ (x \cdot zs)$$

This well-known technique is similarly described by Naish [16] and Rouveirol [21]. We also support flattening of higher-order functions, which allows inversion of higher-order functions if the function argument is invertible.

6 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 [15]. There are more sophisticated mode analysis approaches, e.g., by Smaus et al. [23] using abstract domains and Overton et al. [18] by translating to a boolean constraint system. 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 an test data generator.

Modes. For a predicate P with k arguments, we call *mode* a particular dataflow assignment by a which follows the type of the predicate and annotates all arguments as input (i) or output (o), e.g., for $append_P$, $o \Rightarrow o \Rightarrow i \Rightarrow bool$ denotes the mode where the first two arguments are output, the last argument is input.
A *mode assignment* for a given clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$ 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 $\overline{t}$ and a mode M, the projection expression $\overline{t}\langle M\rangle$ denotes the list of all arguments in $\overline{t}$ (in the order of their occurrence) which are input in M.

Mode consistency. Given a clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$ 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)** $v_0 = FV(\overline{t}\langle M\rangle)$ and **(2)** $v_j = v_{j-1} \cup FV(\overline{u}_j)$ obey the conditions **(3)** $FV(\overline{u}_j\langle M_j\rangle) \subseteq v_{j-1}$ and **(4)** $FV(\overline{t}) \subseteq v_n$. 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 computations 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 number of missing values, i.e., have $|FV(\overline{u}_j\langle M_j\rangle) - v_{j-1}|$ is 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, i.e., a one-to-one relation. 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., the one value for y can either 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 *creating* a dataflow by filling the gaps with value generators.

7 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 and show its application in the hotel key card example.

Monads for non-deterministic computations. We use lazy sequences to enumerate the (potentially infinite) set of values fulfilling the involved predicates – in other words, the lazy sequences will hold the *enumerated solutions*. As customary [25], they are implemented using the ML datatype *'a lazy*. On lazy sequences, 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*. They provide executable versions of basic set operations: $\mathit{empty} = \emptyset$, $\mathit{single}\ x = \{x\}$, $\mathit{plus}\ A\ B = A \cup B$ and $\mathit{bind}\ A\ f = \bigcup_{x \in A} f\ x$. Using lazy sequences 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 $\mathit{int} \to \mathit{'a\ lazy}$ 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 $\mathit{bind}^+\ xq\ f = (\lambda i.\ \mathit{if}\ i = 0\ \mathit{then}\ \mathit{empty}\ \mathit{else}\ \mathit{bind}\ (xq\ (i-1))\ (\lambda a.\ f\ a\ i))$.

In negative contexts, we must distinguish more explicitly 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 $\mathit{int} \to \mathit{'a\ option\ lazy}$ 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). This is witnessed by the behavior of the bind^- operator: $\mathit{bind}^-\ \mathit{single\text{-}none}\ (\lambda x.\ \mathit{empty}^-) = \mathit{empty}^-$ where *single-none* is the singleton sequence with the unknown value.

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 which the involved types τ come equipped with an operation $\mathit{gen}\ \tau$, the generator for type τ that enumerates all values as lazy sequence. For recursively-defined datatypes τ with n constructors $C_1\ \tau_1^1 \ldots \tau_1^{m_1} \mid \ldots \mid C_n\ \tau_n^1 \ldots \tau_n^{m_n}$ we construct generators that enumerate values exhaustively up to depth d by the following scheme:

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

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.** The ML function $hotel^o$ with mode $o \Rightarrow bool$ is the test data generator for the predicate *hotel*:

$$
\begin{aligned}
&\textbf{val } hotel^o = (single\ () \gg\!= (\lambda().\ single\ [])) \\
&\quad \sqcup\ (single\ () \gg\!= (\lambda().\ hotel^o \gg\!= (\lambda evs.\ {hotel_{aux}}^{oi}\ evs \gg\!= (\lambda e.\ single\ (e \cdot evs))))) \\
&\textbf{fun } {hotel_{aux}}^{oi}\ evs = (single\ evs \gg\!= (\lambda evs.\ {currentkey_P}^{ioo}\ evs \gg\!= (\lambda(r, k_1).\ gen_{key} \\
&\qquad \gg\!= (\lambda k_2.\ not\ ({issued}^{ii}\ evs\ k_2) \gg\!= (\lambda().\ gen_{guest} \gg\!= (\lambda g.\ single\ (Checkin\ g\ r\ (k_1, k_2)))))))) \\
&\quad \sqcup\ (single\ evs \gg\!= (\lambda evs.\ {cards}^{ioo}\ evs \\
&\qquad \gg\!= (\lambda(g, (k_1, k_2)).\ {hotel_{aux2}}^{ioii}\ evs\ k_1\ k_2 \gg\!= (\lambda r.\ single\ (Enter\ g\ r\ (k_1, k_2)))))) \\
&\quad \sqcup\ (single\ evs \gg\!= (\lambda evs.\ {isin}^{ioo}\ evs \gg\!= (\lambda(r, g).\ single\ (Exit\ g\ r)))) \\
&\textbf{fun } {hotel_{aux2}}^{ioii}\ evs\ k_1\ k_2 = \\
&\quad (single\ (evs, (k_1, k_2)) \gg\!= (\lambda(evs, (k_1, \_)).\ {roomkey_P}^{ioi}\ evs\ k_1 \gg\!= (\lambda r.\ single\ r))) \\
&\quad \sqcup\ (single\ (evs, (k_1, k_2)) \gg\!= (\lambda(evs, (\_, k_2)).\ {roomkey_P}^{ioi}\ evs\ k_2 \gg\!= (\lambda r.\ single\ r)))
\end{aligned}
$$

The generator $hotel^o$ constructs hotel traces in a bottom-up fashion. ${hotel_{aux}}^{oi}$ adds a new event as prefix to (shorter) hotel traces. ${hotel_{aux}}^{oi}$ can either prefix a trace by *Checkin*, *Enter*, or *Exit* events; the conditions for these events, i.e., restriction on the values of these constructors, are fulfilled by either computing values using further generating functions or are generated unrestrictedly based on their type. An instance of computation is the call ${isin}^{ioo}\ evs$ to construct *Exit* events; an instance of generation is gen_{guest} to select a guest for *Checkin* events. Applying the counterexample generator to the safety property (cf. §3) results in the following counterexample trace:

$$
\begin{aligned}
&Enter\ g_1\ r_0\ (k_1, k_2) \cdot Enter\ g_1\ r_0\ (k_0, k_1) \cdot Checkin\ g_0\ r_0\ (k_2, k_3) \\
&\cdot Checkin\ g_1\ r_0\ (k_1, k_2) \cdot Checkin\ g_0\ r_0\ (k_0, k_1)
\end{aligned}
$$

This resembles the following situation in a hotel with one room r_0: **(1)** Joe (Guest g_0) checks in and gets a card (k_0, k_1). **(2)** Eve (Guest g_1) checks in and gets a card (k_1, k_2). **(3)** Joe checks in again and gets a card (k_2, k_3). **(4)** At this point, Joe has two cards for the room: He tries the newest card (k_2, k_3), but it does not open the door, so he gives it a try with the card from his last stay (k_0, k_1) which unlocks the door. Feeling safe in his room, he puts his wallet on the nightstand and goes to bed. **(5)** At night, Eve enters the room with card (k_1, k_2) and takes Joe's wallet. A subtle error in the key card system causes this jeopardy and can be resolved if Joe would have followed a reasonable safety policy, i.e., to only use his recent card. After understanding the counterexample and formulating this safety policy, one can prove the safety of the key card system.

8 Related work

The idea of specification-based testing was pioneered by the Haskell tool QuickCheck and has many descendants in interactive theorem provers, e.g., Agda/Alfa [9], ACL2 [10], Isabelle [2] and PVS [19], 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 [22] 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 [22] uses partially-instantiated values and a refinement algorithm to simulate narrowing in Haskell. This is closely related to the work of Lindblad [14] and EasyCheck [7], based on the narrowing strategy in the functional logic programming language Curry [12]. 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, have in common *black-box testing*, i.e., not considering the description of the precondition - they generate (partial) values and test the precondition.

Previous work [1] 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 approach 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. Closely related to our work is the glass-box testing by Fischer and Kuchen [11] for functional logic programs, but they take advantage of narrowing and non-determinism features of Curry.

Another approach to finding values that fulfill the preconditions is to use a CLP(FD) constraint solver, as done by Carlier et al. [4]. Testing specifications using αProlog [6] is described by Cheney and Momigliano [5]. A completely different approach to finding counterexamples is translating the specification to propositional logic and invoking a SAT solver, as practiced by the Isabelle tools Refute [26] and Nitpick [3].

9 Conclusion

We described a counterexample generator that improves upon existing solutions by translating specifications into logic programs and which in turn are processed to functional programs, applying an enriched mode analysis. This counterexample generator is included in the next Isabelle release and can be invoked by Isabelle's users to validate their specifications before proving them correct.

Thus, we adopt mode analysis, a common technique from *logic programming*, and apply it in the context of *functional programming* for synthesizing test data generators. We employ the analysis in a compilation with an embedding of depth-limited non-deterministic computations in the functional language. Using these generators for preconditions allows us to find counterexamples in Isabelle specifications where type-based exhaustive and random testing have failed.

In the future, we would like to investigate counterexample generation via testing in (functional) logic programming languages, e.g., Curry, Mercury [24], αProlog [6] and XSB [20].

Acknowledgements

I would like to thank Andrei Popescua, Sascha Boehme, Tobias Nipkow, Alexander Krauss and the anonymous referees for comments on earlier versions of this paper.

References

1 Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning inductive into equational specifications. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 131–146. Springer, 2009.

2 Stefan Berghofer and Tobias Nipkow. Random Testing in Isabelle/HOL. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

3 Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.

4 Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. Constraint Reasoning in FocalTest. In *5th International Conference on Software and Data Technologies (ICSOFT 2010)*, 2010.

5 James Cheney and Alberto Momigliano. Mechanized metatheory model-checking. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 75–86. ACM, 2007.

6 James Cheney and Christian Urban. Alpha-Prolog: A Logic Programming Language with Names, Binding and Alpha-Equivalence. In *Proceedings of the International Conference on Logic Programming (ICLP 2004)*, volume 3132 of *LNCS*, pages 269–283, 2004.

7 Jan Christiansen and Sebastian Fischer. EasyCheck – Test Data for Free. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, volume 4989 of *LNCS*, pages 322–336. Springer, 2008.

8 Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 268 – 279. ACM SIGPLAN, 2000.

9 Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *Theorem Proving in Higher Order Logics*, pages 188–203, 2003.

10 Carl Eastlund. Doublecheck your theorems. In *Eighth International Workshop On The ACL2 Theorem Prover and Its Applications*, 2009.

11 Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 63–74. ACM, 2007.

12 M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 45–75. Springer, 2007.

13 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

14 Fredrik Lindblad. Property directed generation of first-order test data. In *The Eigth Symposium on Trends in Functional Programming*, 2007.

15 C. S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence, 1981.

16 Lee Naish. Adding equations to NU-Prolog. In *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, LNCS, pages 15–26. Springer, 1991.

17 Tobias Nipkow. Verifying a Hotel Key Card System. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *LNCS*. Springer, 2006. Invited paper.

18 David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '02, pages 109–120. ACM, 2002.

19 Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods*, 2006.

20 Prasad Rao, Konstantinos Sagonas, Terrance Swift, David Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *LNCS*, pages 430–440. Springer, 1997.

21 Céline Rouveirol. Flattening and Saturation: Two Representation Changes for Generalization. *Mach. Learn.*, 14(2):219–232, 1994.

22 Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2008.

23 Jan-Georg Smaus, Patricia M. Hill, and Andy King. Mode analysis domains for typed logic programs. In *Selected papers from the 9th International Workshop on Logic Programming Synthesis and Transformation*, pages 82–101, London, UK, 2000. Springer.

24 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1-3):17–64, December 1996.

25 Philip Wadler. How to replace failure by a list of successes. In *Proceedings of a conference on Functional programming languages and computer architecture*, pages 113–128. Springer, 1985.

26 Tjark Weber. Bounded model generation for Isabelle/HOL. In Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle, Silvio Ranise, and Cesare Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 103–116. Elsevier, 2005.

27 Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Ait Mohamed, Munoz, and Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.