Technische Universität München
Institut für Informatik

Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming

Lukas Bulwahn

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Andrey Rybalchenko
Prüfer der Dissertation:	
1.	Univ.-Prof. Tobias Nipkow, Ph.D.
2.	Prof. Colin Runciman The University of York, UK

Die Dissertation wurde am 08.10.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.02.2013 angenommen.

Abstract

This thesis presents a counterexample generator for the interactive theorem prover Isabelle/HOL that uncovers faulty specifications and invalid conjectures using various testing methods.

The primary contributions are two novel testing strategies: exhaustive testing with concrete values; and symbolic testing, evaluating conjectures with a narrowing strategy. Orthogonally to the strategies, this work addresses two general issues: First we extend the class of executable conjectures and specifications. One main aspect are techniques to improve the capabilities of the code generator to extend it to Isabelle's logic programs and to turn non-executable specifications into executable ones by automatic transformations. Second, we present techniques to deal with conditional conjectures, i.e., conjectures with restrictive premises. This thesis includes a novel approach to synthesize test data generators based on an extended mode analysis that creates data derived from the premise's definition. When the tool applies these techniques, testing requires a much smaller number of test cases to find errors in specifications.

The testing strategies and techniques are evaluated on numerous existing specifications developed in Isabelle/HOL, covering such areas as semantics of programming languages, efficient functional data structures, cryptographic protocols and graph theory.

Zusammenfassung

Diese Dissertation beschreibt einen Gegenbeispielgenerator für den interaktiven Theorembeweiser Isabelle/HOL, der fehlerhafte Spezifikationen und ungültige Hypothesen durch verschiedene Testmethoden aufdeckt.

Der primäre Beitrag dieser Arbeit sind zwei neue Teststrategien: erschöpfendes Testen mit konkreten Werten und symbolisches Testen, bei dem Hypothesen mit einer Narrowing-Strategie evaluiert werden. Orthogonal zu den Strategien adressiert diese Arbeit noch weitere Aspekte: Zum einen wird die Klasse der ausführbaren Hypothesen und Spezifikationen erweitert. Ein Hauptaspekt sind Techniken, um Fähigkeiten der Codegenerierung zu erweitern, und um unausführbare Spezifikationen in ausführbare durch automatische Transformationen umzuwandeln. Zum anderen beschreiben wir Techniken, um mit bedingten Hypothesen—Hypothesen mit restriktiven Prämissen—umzugehen. Die Arbeit beinhaltet einen neuen Ansatz, um Testdatengeneratoren zu synthetisieren, die Daten erzeugen, die aus der Definition der Bedingung hergeleitet werden. Der Ansatz basiert dabei auf einer erweiterten Modusanalyse. Durch Anwendung dieser Techniken werden wesentlich weniger Testfälle benötigt, um Fehler in Spezifikationen zu finden.

Wir evaluieren die Teststrategien und Techniken auf zahlreichen existierenden Spezifikationen in Isabelle/HOL aus Bereichen wie Semantik von Programmiersprachen, effizienten funktionalen Datenstrukturen, kryptographischen Protokollen und Graphentheorie.

Acknowledgment

First of all, I am indebted to my supervisor Tobias Nipkow who guided me through my research and inspired me to work on counterexample generation. I express my gratitude to Tobias, Larry Paulson and Makarius Wenzel for their continuous work on Isabelle and providing a software system that served as an excellent platform for my research.

I am delighted that Colin Runciman accepted the invitation to referee this thesis. I gratefully thank the head of our doctorate program Helmut Seidl to create a great working environment for research of formal methods, and allowing me to receive a broad education in this field.

I want to thank all present and former members of the Isabelle group at the Technische Universität München for making our research group such a friendly place of mutual support: Jasmin Christian Blanchette, Stefan Berghofer, Sascha Böhme, Florian Haftmann, Johannes Hölzl, Brian Huffman, Cezary Kaliszyk, Alexander Krauss, Ondřej Kunčar, Peter Lammich, Lars Noschinski, Andrei Popescu, Dmitriy Traytel, Thomas Türk, Christian Urban and Makarius Wenzel.

I owe a debt of gratitude to Jasmin and Ondřej for reading my thesis in their spare time and suggesting several textual improvements in this thesis.

Jasmin and Andrei deserve special thanks for being my office roommates and helping me throughout the day with many questions. Jasmin motivated and inspired me to develop the tool with focus on Isabelle's users and to address many technical limitations that users encounter when using the developed tool. Andrei gave me constant advice and pushed me to complete this thesis on every occasion.

Andreas Lochbihler was the first using many new developments and provided valuable feedback. I am thankful for his effort and his explanations, which have driven numerous improvements in our development. Thomas Genet uses the tool extensively in his teaching courses and pointed out some bugs in the tool.

I had the pleasure to have great companions in the doctorate program: Chih-Hong (Patrick) Cheng, Jan Hoffmann, Andreas Gaiser, Ruslan Ledesma Garza, Ashutosh Gupta, Johannes Hölzl, Christian Kern, Máté Kovács, Markus Latte, Bogdan Mihaila, Andreas Reuß, Dulma Rodriguez and Markus Weissmann.

I thank Katharina Spiess for inviting me to join the staff of the Marktoberdorf summer school in 2011. The participation in the summer school gave me enlightenment in other research topics and served as a great platform for communicating with other PhD students in the field of formal methods.

My research was financed by the German Research Council under the doctorate program Program and Model Analysis (DFG-GRK 1480).

Contents

Chapter 1

Introduction

The thesis describes Quickcheck, a counterexample generator for Isabelle/HOL that uncovers faulty specifications and invalid conjectures using various testing methods.

1.1 Motivation

Writing programs and specifications is an error-prone business, and testing is common practice to find errors and validate software. As computer scientists are aware that testing alone 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. Interactive theorem provers are used to develop a proof with trustworthy 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.

Common user experience with interactive theorem provers suggests that most conjectures initially stated in an interactive theorem prover do not hold. Typically, errors in conjectures and specifications are due to typos or missing assumptions, but sometimes they are owing to fundamental flaws in the specifications.

Modern interactive theorem provers therefore provide not only means to prove properties, but also to *disprove* properties with counterexample generators. The interactive theorem prover Isabelle [119] provides counterexample generators that uncover invalid conjectures by two means:

- Refute [117, 118] searches for finite countermodels by reducing a conjecture to boolean satisfiability directly. Its successor Nitpick [21] reduces to boolean satisfiability in two steps: It first reduces the conjecture in higher-order logic to first-order relational logic, and then employs the tool Kodkod [112], harnessing its optimized reduction from first-order relational logic to boolean satisfiability.

- Quickcheck searches for counterexamples by *testing* the conjecture. 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. Quickcheck *tests* a conjecture by assigning values to the free variables of the conjecture and evaluating it. To evaluate the conjecture efficiently, Quickcheck translates the conjecture and related definitions to an ML or Haskell program, exploiting Isabelle's code generation infrastructure [54]. This allows Quickcheck to test a conjecture with millions of test cases within seconds.

Although the counterexample generators are built with the same motivation in mind, their abilities are disjoint: Refute and Nitpick can explore large abstract relational specifications, whereas Quickcheck's strategies are lost rather fast in the large search space. When the specifications and conjectures are closer to functional programs, Refute and Nitpick are usually limited to programs with a few lines of code, but Quickcheck can be successfully applied to find errors in large functional programs and software systems.

Novices profit a lot from counterexample generators. Counterexample generators give them the necessary feedback to learn writing correct specifications. They also provide means to make proving a trial-and-error experience until novices reach a better understanding how to work with an interactive theorem prover.

Nonetheless, even long-standing users benefit from the counterexample generators. They regularly do not know all details of the formalized theories by heart. As formalizations deliberately move away from common intuitions and descriptions in textbooks to simplify definitions or proofs in the formal system, even experts can err and counterexample generators provide an elegant way to check their intuitions against the formalizations. Especially when experts build formalizations on top of existing theories from others, they learn the peculiarities of a formalization with small concrete examples provided by the counterexample generators.

1.2 Contributions

Our primary contribution is the continued integration of testing methods in Isabelle under the hood of the existing tool Quickcheck.

In earlier work [14], Quickcheck was originally modeled after the QuickCheck tool for Haskell [33], which tests user-supplied properties of a Haskell program with randomly generated values. Our first contribution is to extend Quickcheck with exhaustive testing to complement random testing. Exhaustive testing checks the formula for every possible set of values up to a given bound, and hence finds small counterexamples that random testing might miss.

The two testing approaches above are limited to evaluations with ground values. Our second contribution is to extend Quickcheck with a symbolic, narrowing-based testing approach. The narrowing-based testing approach evaluates the formula symbolically rather than evaluating with a finite set of ground values. Therefore, it can be more precise and more efficient than the other approaches.

A well-known problem of testing are *conditional conjectures*, especially those

with very restrictive premises. These conjectures are problematic because when testing naively, for the vast majority of variable assignments the premise is not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the premise into account when generating values. Our third contribution is to present three solutions for Quickcheck to generate only appropriate variable assignments:

- Custom generators: Derivation of custom test data generators from user declarations
- Smart testing: Automatic synthesis of test data generators that take the premise's definition into account
- Symbolic evaluation: Search space pruning by refining variable assignments symbolically

Last, the new implementation of Quickcheck supports many specific features of Isabelle, notably parametrized conjectures, polymorphic conjectures and underspecified functions.

In total, Quickcheck incorporates four testing approaches: random, exhaustive, smart and symbolic testing. To illustrate the different testing approaches, we consider how they check the following conjecture about *take n xs*, which computes the prefix of length *n* of the list *xs*.

$$\forall n\ m\ xs.\ n \leq m\ \wedge\ m \leq \mathit{length}\ xs \Longrightarrow \mathit{take}\ n\ (\mathit{take}\ m\ xs) = \mathit{take}\ n\ xs$$

Figure 1.1 shows the first test cases of the different testing approaches in Isabelle's Quickcheck. A test case consists of an assignment of the universal quantified variables *n*, *m*, *xs*. The elements of list *xs* are distinct atoms a_1, a_2 and a_3.

The testing approaches differ in the choice and order of test cases. We present the advantages and disadvantages of the testing approaches by viewing the test cases of each approach for this conjecture. In this example, we want to check the validity of the conjecture and for possible errors in the function *take*, but we know that the other functions, such as *length*, are implemented correctly. We also note that test cases that do not fulfill the premises of the conjecture trivially make the conjecture true without checking the conclusion and the function under test and therefore are considered *superfluous*.

With random testing, three test cases are superfluous. The first two test cases do not fulfill the premise $n \leq m$, the fourth test case violates the premise $m \leq$ *length xs*. Only the third test case checks the conclusion and could reveal the conjecture to be invalid. Although generating the test cases is simple, some test cases obviously violated the premises.

Exhaustive testing suffers from this drawback in a similar way. For example, its second and third test case violate the premise as well. Furthermore when we test exhaustively, we test with many small values before we test with larger ones; indeed, in the first four test cases there is not a single test case that tests the conjecture with a list of length 2.

The third approach, smart testing, employs a special generator for the less-or-equal relation and lists of fixed length. It also determines a reasonable order for the generation. First, we generate numbers *n* and *m* with $n \leq m$, and then

Random Testing:

1. $n = 2,\ \ m = 3,\ \ xs = [\mathsf{a}_1, \mathsf{a}_1]$
2. $n = 4,\ \ m = 1,\ \ xs = []$
3. $n = 1,\ \ m = 2,\ \ xs = [\mathsf{a}_3, \mathsf{a}_1, \mathsf{a}_2]$
4. $n = 2,\ \ m = 4,\ \ xs = [\mathsf{a}_2, \mathsf{a}_1]$

Exhaustive Testing:

1. $n = 0,\ \ m = 0,\ \ xs = []$
2. $n = 1,\ \ m = 0,\ \ xs = []$
3. $n = 0,\ \ m = 1,\ \ xs = []$
4. $n = 0,\ \ m = 0,\ \ xs = [\mathsf{a}_1]$

Smart Testing:

1. $n = 0,\ \ m = 0,\ \ xs = []$
2. $n = 0,\ \ m = 0,\ \ xs = [\mathsf{a}_1]$
3. $n = 0,\ \ m = 1,\ \ xs = [\mathsf{a}_1]$
4. $n = 1,\ \ m = 1,\ \ xs = [\mathsf{a}_1]$

Symbolic Testing:

1. $n = 0,\ \ m = M,\ \ xs = XS$
2. $n = Suc\ N,\ \ m = 0,\ \ xs = XS$
3. $n = Suc\ N,\ \ m = Suc\ M,\ \ xs = []$
4. $n = Suc\ 0,\ \ m = Suc\ M,\ \ xs = X \cdot XS$

Figure 1.1: Test cases of Quickcheck's testing approaches for the conjecture $\forall n\ m\ xs.\ n \leq m\ \wedge\ m \leq length\ xs \Longrightarrow take\ n\ (take\ m\ xs) = take\ n\ xs$

we generate a list xs that has a length greater than m. In this approach, we test exhaustively, but avoid the superfluous test cases.

Finally, we describe the test cases of the symbolic testing. In the presentation, variables with capital letters N, M, X and XS denote symbolic values. The first symbolic test case chooses 0 for n, but keeps the value of xs and m purely symbolic. If n is 0, the conjecture's evaluation exhibits the same behavior for any xs and m, and hence can be covered with this one symbolic test. As we explored the case that n is 0, we continue with test cases where n is not 0, i.e, n is instantiated to $Suc\ N$ for some fresh symbolic value N. The second test case checks the conjecture with a symbolic value with $n = Suc\ N$ and $m = 0$. This test violates the premise $n \leq m$. All further test cases of this form can be discarded. In the next test case, we choose $n \geq 1$ and $m \geq 1$ and check with the instantiation $xs = []$. This time, the second premise $m \leq length\ xs$ is violated. Hence, testing continues with $xs = X \cdot XS$ and instantiates $N = 0$. Remarkably, a single symbolic evaluation can cover many symmetric evaluations with concrete values. In the presence of premises, symbolic evaluation can discard many further instantiations and symbolic tests, and thus can prune the search space.

After the presentation of this example, one might be tempted to draw premature conclusions. For example, one could think that smart testing must be more efficient than the naive approaches, as it takes premises into account, or that symbolic testing outperforms testing with ground values because it can possibly cover many concrete tests with one symbolic test. However, only a thorough investigation with examples and case studies will show if these hypotheses are true. This is our last contribution in this thesis: shedding some light on the performance of the different testing approaches.

Traditionally testing methods can be divided into white-box and black-box

testing, according to the point of view that one takes to choose test cases:

- Black-box testing treats the object under test, e.g., software, specifications or conjectures, as a black box, without taking the knowledge about the internal structure of the object into account. Random and exhaustive testing choose the values independent of the property under test and lie in the category of black-box testing.
- White-box testing in contrast takes an internal perspective on the object. Our smart testing technique follows this idea of white-box testing, as the test cases are generated taking the definition of premises into account.

The symbolic testing generates test values without taking the internal structure into account. However, the choice of test cases is driven by the result of previous evaluations and these results can depend on the internal structure. Hence, it seems inappropriate to classify the symbolic testing in Quickcheck as black-box or white-box testing.

1.3 Publications

Most of the contributions described here have been presented at international conferences. This thesis was accompanied by the following papers:

1. L. B. Smart test data generators via logic programming. In: John Gallagher and Michael Gelfond, editors, *Technical Communications of the 27th International Conference on Logic Programming (ICLP 2011)*, pages 139–150, volume 11 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011.
2. J. C. Blanchette, L. B., and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems (FroCoS 2011)*, volume 6989 of *LNAI*, pages 12–27. Springer, 2011.
3. L. B. Smart testing of functional programs in Isabelle. In N. Bjørner and A. Voronkov, editors, *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-18)*, volume 7180 of *LNCS*, pages 153–167. Springer, 2012.
4. L. B. The New Quickcheck for Isabelle: random, exhaustive and symbolic testing under one roof. Accepted at *The Second International Conference on Certified Programs and Proofs (CPP 2012)*.

Some of our effort to improve code generation in Isabelle, which was done as part of the Ph.D., is described in this joint work with Andreas Lochbihler:

4. A. Lochbihler and L. B. Animating the formalised semantics of a Java-like language. In: M. van Eekelen and H. Geuvers and J. Schmalz and F. Wiedijk, editors, *Interactive Theorem Proving (ITP 2011)*, pages 216–232, volume 6898 of *LNCS*, Springer, 2011.

1.4 Structure of This Thesis

The thesis is structured as follows:

- Chapter 2 briefly introduces higher-order logic, Isabelle's definitional principles and code generation.
- Chapter 3 mainly describes random and exhausting testing, but also a collection of techniques integrated in Quickcheck.
- Chapter 4 presents two techniques to handle conditional conjectures.
- Chapter 5 describes symbolic testing by narrowing.
- Chapter 6 presents the evaluation of the testing approaches on various case studies.
- Chapter 7 summarizes our results and gives directions for future work.

As the syntax in this thesis largely adheres to standard Isabelle notation, an expert Isabelle user can skip chapter 2 and directly start with the subsequent chapters. Chapter 4 relies on some parts of chapter 3. Chapter 5 can be read independently of the previous chapters, but it requires to follow some references to chapter 3 and chapter 4 for the introduction of some examples. In any case, if parts in subsequent chapters relate to previous descriptions in other chapters, we point to the subsection for further reading. Related work is considered at the end of each chapter 3 to 5. To follow the evaluation of chapter 6, readers need not to know the technical details of the testing approaches. However, further explanations for their behavior is only understood with some knowledge of the previous chapters.

Chapter 2

Background

This chapter introduces interactive theorem proving, higher-order logic and the definitional mechanisms in the interactive theorem prover Isabelle. After this general introduction, we focus on Isabelle's code generation, which provides the basic infrastructure for the Quickcheck tool.

2.1 Interactive Theorem Proving

Interactive theorem proving acknowledges that humans and computers capitalize their abilities together best through interaction. Humans capture ideas with their intuition and are gifted with creativity, whereas machines can reliably check the correctness of human deductions and can aid the human on various computational tasks with proof automation and decision procedures.

Interactive theorem provers or proof assistants are systems that provide means for this type of interaction. Typical systems of this kind are ACL2 [73], Coq [18], HOL4 [51], HOL Light [55], Mizar [86], PVS [96] and Isabelle [93, 98, 119], which serves as platform for this work.

2.1.1 Isabelle/HOL

Isabelle is a *generic* interactive theorem prover, and can be instantiated with different logics. Isabelle's most widely used logic is *Isabelle/HOL*. It provides classical higher-order logic with rank-1 polymorphism and axiomatic type classes. Isabelle/HOL's types and terms are based on the simply typed λ-calculus [32]. Types are constructed by type variables α and type constructors κ with fixed arities:

$$\tau \ ::= \ \alpha \mid (\tau_1, \ldots, \tau_n)\ \kappa$$

For type constructors with arity 1, we omit the parentheses and write $\tau\ \kappa$. We use the Greek letters α, β, γ for type variables and τ for types. Isabelle/HOL is equipped with two special types, the type of boolean values *bool* and the type of functions $\Rightarrow$, commonly used infix as in $\alpha \Rightarrow \beta$.

Terms are typed variables, typed constants, applications or typed λ-abstractions:

$$t ::= x^\tau \mid c^\tau \mid t_1\ t_2 \mid \lambda\ x^\tau.\ t$$

A sequence of terms is denoted by $\bar{t}$. Terms are generally viewed under the equivalence of α-renaming, β-reduction, and η-reduction.

The type system of Isabelle/HOL is the Hindley-Milner type system extended with type classes [116], similar to the original Haskell type system [60]. Throughout the presentation, we assume that all terms are well-typed. Whenever the type of a term is not clear by the given context, we annotate a term t by $t :: \tau$ to denote that it is of type τ.

Isabelle/HOL supplies meta-operators such as an universal quantifier and an implication with its meta-logic, and copies of those two operators with higher-order logic (object operators). For our presentation, the distinction between the meta-operators and the object operators is not relevant and we employ only the object operators $\forall :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ and $\Longrightarrow :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$.[1] Mimicking mathematical syntax, $\forall(\lambda x.\ P\ x)$ is written as $\forall x.\ P\ x$.

2.1.2 Definitional Principles

Isabelle/HOL provides two basic means to extend a theory development consistently: simple definitions for constants and Gordon's HOL type definitions. Grounding on these two theory extensions, Isabelle/HOL provides many other means to introduce new constants and type constructors.

Simple Definitions

With **definition** $c :: \tau$ **where** $c\ \bar{x} = t$, Isabelle defines the constant c by introducing the axiom $c = \lambda\bar{x}.\ t$. It ensures that the definition is conservative by some syntactic checks. Isabelle checks that the constant c has not been declared before, and that the closed term t refers only to existing constants and free variables in $\bar{x}$. Furthermore, the variables $\bar{x}$ must be distinct and all type variables in t must occur in τ.

Type Definitions

Given a non-empty subset of an existing type, a new type can be defined by creating an isomorphic copy of the given subset. For example, to obtain a type of three elements, we define a type *three* as a copy of a subset of the natural numbers:

typedef $\mathit{three} = \{0, 1, 2\}$

The **typedef** command axiomatizes the type and provides a bijection between the new type and the set $\{0, 1, 2\}$ with $\mathit{Rep}_{\mathit{three}} :: \mathit{three} \Rightarrow \mathit{nat}$ and $\mathit{Abs}_{\mathit{three}} :: \mathit{nat} \Rightarrow \mathit{three}$. With these bijections, we refer to the values of type *three* by $\mathit{Abs}_{\mathit{three}}\ 0$, $\mathit{Abs}_{\mathit{three}}\ 1$ and $\mathit{Abs}_{\mathit{three}}\ 2$. The command **lift-definition** and the *transfer* proof method [62] support to declare definitions and theorems on the new type. The command **lift-definition** spares users to employ the bijections for their definitions. For example, we define constants *zero*, *one* and the function *shift* with

[1] Readers that have been heavily exposed to Isabelle's notation should be aware that the symbol $\Longrightarrow$ denotes the object logic implication in this thesis and not the meta-implication as in Isabelle. Hence, unlike in Isabelle, the scope of $\forall$ ranges over this implication symbol.

lift-definition *zero* :: *three* **is** 0
lift-definition *one* :: *three* **is** 1
lift-definition *shift* :: *three* ⇒ *three* **is** λx. if $x < 2$ then $x + 1$ else 0

Internally, the command defines *zero*, *one* and *shift* as Abs_{three} 0, Abs_{three} 1 and λy. Abs_{three} $((\lambda x.$ if $x < 2$ then $x + 1$ else $0)$ $(Rep_{three}$ $y))$, respectively. Given the theorem $(\lambda x.$ if $x < 2$ then $x + 1$ else $0)$ $0 = 1$ on the existing type, the *transfer* proof method allows us to derive *shift zero* = *one*.

Although the two theory extensions, constant and type definitions, suffice to work with the system, it is convenient to have further derived mechanisms.

Inductive Datatypes

Inductive datatypes [15] define a new type by providing constructors $C_1, \ldots, C_n$ with recursive types $\tau_1, \ldots, \tau_n$. They are defined with the command

datatype $\overline{\alpha}$ $\kappa = C_1$ τ_1 | $\cdots$ | C_n τ_n

The types $\tau_1, \ldots, \tau_n$ are restricted to involve recursive occurrences of the type $\overline{\alpha}$ κ only on the right-hand side of function types and only nested under previously defined datatypes. We heavily employ some basic datatypes, such as products, sums, the *option* type, the *list* type and the type of natural numbers. These datatypes are defined with

datatype $\alpha \times \beta = Pair\ \alpha\ \beta$
datatype $\alpha + \beta = Inl\ \alpha \mid Inr\ \beta$
datatype α *option* $= None \mid Some\ \alpha$
datatype α *list* $= Nil \mid Cons\ \alpha\ (\alpha\ list)$
datatype *nat* $= 0 \mid Suc\ nat$

All datatypes are equipped with associated case expressions. For example, the case expressions for the product and sum types are defined such that

(case *Pair x y* of *Pair a b* ⇒ *f a b*) = *f x y*,
(case *Inl x* of *Inl l* ⇒ *f l* | *Inr r* ⇒ *g r*) = *f x* and
(case *Inr y* of *Inl l* ⇒ *f l* | *Inr r* ⇒ *g r*) = *g y* holds.

A pair with values a and b is written more readably as (a, b). The selectors on pairs *fst* and *snd* follow the law *fst* $(a, b) = a$ and *snd* $(a, b) = b$. We often employ $(\lambda(a, b).\ f\ a\ b)$ as notation for (case (x, y) of $(a, b) \Rightarrow f\ a\ b$).

For lists, we use these common notations: [] and $x \cdot xs$ denote the two list constructors *Nil* and *Cons x xs*. Longer lists such as $(x \cdot (y \cdot (z \cdot Nil)))$ are conveniently written as $[x, y, z]$. The selectors of *Cons*, head and tail, are denoted by *hd* and *tl*. The *hd* function is only specified for non-empty lists, i.e., *hd* $(x \cdot xs) = x$. The tail function is also defined for non-empty lists with the equations *tl* [] = [] and *tl* $(x \cdot xs) = xs$. For natural numbers, we use numerals as abbreviations for nested *Suc* terms, e.g., $1 = Suc\ 0$ and $2 = Suc\ 1$. Isabelle/HOL also allows mutually recursive datatypes, but we do not use them in the presentation of the thesis.

Inductive Predicates

An inductive predicate P is the the least predicate closed under a given set of introduction rules. It is defined by

inductive $P :: \tau \Rightarrow bool$
where
$\quad Q_{1,1}\ \overline{u}_{1,1} \Longrightarrow \cdots \Longrightarrow Q_{1,m_1}\ \overline{u}_{1,m_1} \Longrightarrow P\ \overline{t}_1$
$\mid Q_{n,1}\ \overline{u}_{n,1} \Longrightarrow \cdots \Longrightarrow Q_{n,m_n}\ \overline{u}_{n,m_n} \Longrightarrow P\ \overline{t}_n$

Isabelle ensures that the introduction rules are *monotonic* to guarantee the existence and uniqueness of the least fix point. For example, the inductive predicate *listrel* lifts a given relation on elements r point-wise onto lists:

inductive $listrel :: (\alpha \Rightarrow \alpha \Rightarrow bool) \Rightarrow \alpha\ list \Rightarrow \alpha\ list \Rightarrow bool$
where
$\quad listrel\ r\ []\ []$
$\mid r\ x\ y \Longrightarrow listrel\ r\ xs\ ys \Longrightarrow listrel\ r\ (x \cdot xs)(y \cdot ys)$

Isabelle also allows inductive predicates to be mutually recursive. For example, consider the predicates *even* and *odd*:

inductive $even :: nat \Rightarrow bool$ **and** $odd :: nat \Rightarrow bool$
where
$\quad even\ 0$
$\mid odd\ n \Longrightarrow even\ (Suc\ n)$
$\mid even\ n \Longrightarrow odd\ (Suc\ n)$

Foundationally, they are defined via a conjoined least fixed point and suitable projections. Besides the introduction rules, Isabelle provides an elimination and induction rule for every inductive predicate.

Recursive Functions

Recursive functions can be defined by recursive equations. If the recursive equations are not restricted, the definitions can easily lead to inconsistencies, e.g., the "definition" $f\ x = f\ x + 1$ is inconsistent with $n \neq n + 1$. Therefore, recursive function definitions are only allowed if the recursive calls induce some well-founded ordering.

Two simple examples of recursively defined functions on lists are *append* and *rev*. The *append* function is written infix as ++.

fun $append :: \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$
where
$\quad append\ Nil\ ys = ys$
$\mid append\ (Cons\ x\ xs)\ ys = Cons\ x\ (append\ xs\ ys)$

fun $rev :: \alpha\ list \Rightarrow \alpha\ list$
where
$\quad rev\ Nil = Nil$
$\mid rev\ (Cons\ x\ xs) = rev\ xs +\!\!+ [x]$

To find the well-founded ordering in these cases is simple, as both functions are *primitive recursive* on the first argument.

2.1.3 Type Classes

As type classes play a prominent role in the implementation of the counterexample generators, we illustrate the usage of type classes in Isabelle with a simple example. We consider type classes with a programming language's point of view: Type classes describe collections of types that provide operations of certain names and types. Classes are defined with the command **class** naming their associated operations. The membership of type τ in the class c is given by providing operations in the context of an **instantiation** $\tau :: c$.

Whereas in a programming language like Haskell, the semantic properties of the operations are only implicit, an interactive theorem prover allows making semantic properties explicit by augmenting the classes with *class axioms*. Therefore, class instances not only declare operations with certain names, but also provide proofs that the operations respect the stated class axioms. For example, the class *order* states that types of this class provide a partial order $\leq$:

class *order* =
  fixes $\leq$ $:: \tau \Rightarrow \tau \Rightarrow bool$
  assumes $x \leq x$
  and $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
  and $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$

As stated by the three class axioms, the relation should be reflexive, antisymmetric and transitive. Choosing the ordering on natural numbers, we declare the type *nat* a member of the type class *order*:

instantiation *nat* :: *order*
begin
  fun $\leq_{nat}$ $:: nat \Rightarrow nat \Rightarrow bool$
  where
    $0 \leq_{nat} m \longleftrightarrow True$
    $Suc\ n \leq_{nat} Suc\ m \longleftrightarrow n \leq_{nat} m$
  instance $\langle proof \rangle$
end

The boundary of a context is indicated by keywords **begin** and **end**. The notation $\langle proof \rangle$ is placeholder for the proof of the class axioms for $\leq_{nat}$. The actual proof is of no interest here. Isabelle also allows *parametric instances*. For example, the order on α *list* can be defined extending the order on α to a point-wise order on α *list*:

instantiation *list* :: (*order*) *order*
begin
  definition $\leq_{list}$ $:: \alpha\ list \Rightarrow \alpha\ list \Rightarrow bool$
  where $xs \leq_{list} ys \longleftrightarrow listrel\ (\lambda x\ y.\ x \leq y)\ xs\ ys$

instance ⟨*proof*⟩

end

Similarly one can provide instantiations for other type constructors, like the product and sum type.

Note that there is at most one instantiation for each type constructor. Consequently, once the class *order* on lists is instantiated to point-wise ordering, it permits us to redefine the ordering to be the lexicographic ordering on lists. Type classes enable us to define constants by primitive recursion over types, e.g., the order on lists is defined recursing on the order of the list's type arguments. We use type classes for definitions with primitive recursion over types heavily in this thesis. Throughout the presentation, we denote the instance of a class operation c with type τ by c_τ.

2.2 Code Generation

With all the definitional principles and type classes at hand, HOL provides everything to serve as an adequate functional programming language and specification language, as suggested by Nipkow's slogan [93, ch. 1.1]:

HOL = Functional Programming + Logic

However, one should be aware of subtle differences between the HOL logic and a functional programming language: HOL does not mandate any fixed evaluation order. Furthermore, as HOL allows us to define non-computable functions, the definitions in HOL may not even have a counterpart in a functional programming language.

Isabelle's code generator [53, 54] views a subset of HOL as a functional programming language and turns a set of equational theorems (*code equations*) into a functional program with the same equational rewrite system. As it builds on equational logic, the translation guarantees partial correctness and allows the user to refine programs and data. The code generator supports the target languages Standard ML, OCaml, Haskell and Scala.

For example, Isabelle's code generator produces the following Standard ML code for the *append* function:

```
structure List : sig
  datatype 'a list = Nil | Cons of 'a * 'a list
  val append : 'a list -> 'a list -> 'a list
end = struct

datatype 'a list = Nil | Cons of 'a * 'a list;

fun append Nil ys = ys
  | append (Cons (x, xs)) ys = Cons (x, append xs ys);

end;
```

In a first approximation, code generation might be considered a simple syntactic transformation of the function's definition. In the next section, we see that the generated code can tremendously differ from the definition in HOL.

Type classes are eliminated by expressing them with suitable notions of the target language. In Haskell, they are expressed by the built-in type classes. For target languages that do not support type classes, like Standard ML, they can be expressed by a dictionary construction.

2.2.1 Program Refinement

Program refinement is a technique that enables us to separate code generation issues from the rest of the formalization. As any executable equational theorem suffices for code generation, users can *locally* derive new equations to be used for code generation. Hence, existing definitions and proofs remain unaffected, while still providing an efficient implementation.

The Standard ML code for the *rev* function above can be generated using these two theorems as code equations:

$$rev\ Nil = Nil$$
$$rev\ (Cons\ x\ xs) = rev\ xs\ @@\ [x]$$

However, the code equations are exchangeable. We can also employ other derived equations for code generation. This mechanism is called *program refinement*. We show a simple example here. We implement list reversal with an optimized tail-recursive function. At first, we simply define an additional constant *prepend-rev* that prepends the reverse of a list to another list:

definition $prepend\text{-}rev :: \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$
where
$prepend\text{-}rev\ xs\ ys = (rev\ xs)\ @@\ ys$

Intuitively, this constant captures the intermediate states in the execution of the list reversal. From this definition, we can derive the following two equations for *prepend-rev*:

lemma [code]:
$prepend\text{-}rev\ Nil\ ys = ys$
$prepend\text{-}rev\ (Cons\ x\ xs)\ ys = prepend\text{-}rev\ xs\ (Cons\ x\ ys)$

The annotation [code] registers these equations as *code equations* for the constant. Instead of the definitional equations, the code generator now use the alternative ones. The *rev* function is also expressed by the alternative equation

lemma [code]:
$rev\ xs = prepend\text{-}rev\ xs\ Nil$

Putting everything together, we obtain a Standard ML program that uses the tail-recursive *prepend-rev* function for list reversal:

```
structure List : sig
  datatype 'a list = Nil | Cons of 'a * 'a list
  val prepend_rev : 'a list -> 'a list -> 'a list
  val rev : 'a list -> 'a list
end = struct

datatype 'a list = Nil | Cons of 'a * 'a list;

fun prepend_rev (Cons (x, xs)) ys =
      prepend_rev xs (Cons (x, ys))
  | prepend_rev Nil ys = ys;

fun rev xs = prepend_rev xs Nil;

end;
```

2.2.2 Data Refinement

Data refinement enables the user to replace constructors of a datatype by other constants and derive equations that pattern-match on these new (pseudo-)constructors. The new constructors neither need to be injective and pairwise disjoint, nor exhaust the type. Again, this affects only code generation, but not the logical properties of the type.

We illustrate data refinement with a simple example. In a strict language, such as Standard ML, functions are executed eagerly. In this example, we set up the code generator to represent lists in such a way that list functions are executed lazily using a standard idiom [99]. The refinement step does not affect the definition of lists in HOL, but for code generation, we view lists as a datatype with alternative constructors.

In a first step, we define the constant *LCons*, the constructor *Cons* with a semantically vacuous unit closure for its tail:

definition $LCons :: \alpha \Rightarrow (unit \Rightarrow \alpha\ list) \Rightarrow \alpha\ list$
where
$LCons\ x\ lxs = Cons\ x\ (lxs\ ())$

The purpose of the unit closure is that list functions are executed lazily in the eager language, as the closure enforces that the inner evaluation is delayed until the unit value is applied to the closure. With the following command, code generation views the constants *Nil* and *LCons* as constructors for the *list* type:

code-datatype *Nil LCons*

Finally, the simple program refinement

lemma [code]:
$append\ Nil\ ys = ys$
$append\ (LCons\ x\ lxs)\ ys = LCons\ x\ (\lambda u.\ append\ (lxs\ ())\ ys)$

for the *append* function allows us to execute the function lazily. We inspect lazy lists with the *retrieve* function, which returns the list's element with a given index and evaluates the list only as far as necessary:

fun *retrieve* :: α *list* $\Rightarrow$ *nat* $\Rightarrow$ α *option*
where
  retrieve Nil i = *None*
| *retrieve* (*Cons x xs*) 0 = *Some x*
| *retrieve* (*Cons x xs*) (*Suc n*) = *retrieve xs n*

lemma [code]:
  retrieve Nil i = *None*
  retrieve (*LCons x xs*) 0 = *Some x*
  retrieve (*LCons x lxs*) (*Suc n*) = *retrieve* (*lxs* ()) *n*

Provided with this setup, the code generator produces the Standard ML program with lazy lists:

```
structure List : sig
  datatype nat = Zero | Suc of nat
  datatype 'a list = Nil | LCons of 'a * (unit -> 'a list)
  val append : 'a list -> 'a list -> 'a list
  val retrieve : 'a list -> nat -> 'a option
end = struct

datatype nat = Zero | Suc of nat;

datatype 'a list = Nil | LCons of 'a * (unit -> 'a list);

fun append (LCons (x, lxs)) ys =
      LCons (x, (fn _ => append (lxs ()) ys))
  | append Nil ys = ys;

fun retrieve (LCons (x, lxs)) (Suc n) = retrieve (lxs ()) n
  | retrieve (LCons (x, lxs)) Zero = SOME x
  | retrieve Nil i = NONE;

end;
```

As the resulting source code in the target language largely reflects the code equations, we omit the presentation of generated code in the rest of this thesis.

2.2.3 Execution of Inductive Specifications

The code generator is limited to purely equational specifications by its design. However, Isabelle also provides means to define constants by inductive predicates (§2.1.2). For the execution of inductive predicates, we must turn an inductive specification into an executable equational one. In the following, we present two techniques to do this.

Unfolding Equations

We have contributed a tool that provides a simple equational description for every inductive predicate: the one-step unfolding of the least-fixed point equation. For example, the predicates *even* and *odd* are equipped with the unfolding equations

$$even\ n = (n = 0\ \vee\ (\exists m.\ n = Suc\ m\ \wedge\ odd\ m))$$
$$odd\ n = (\exists m.\ n = Suc\ m\ \wedge\ even\ m)$$

As these equations contain unbounded existential quantifiers, they are not executable directly. However, if we instantiate the right-hand sides adequately and simplify the equations, the resulting equations do not contain any existential quantifiers and can be used for code generation. In our example, instantiating *even* and *odd* with patterns *even* 0, *even* (*Suc* n), *odd* 0 and *odd* (*Suc* 0), we automatically obtain simple executable equations:

$$even\ 0 = True \qquad odd\ 0 = False$$
$$even\ (Suc\ n) = odd\ n \qquad odd\ (Suc\ n) = even\ n$$

This simple method yields an executable equation if all existential quantifiers in the equations are eliminated. The existential elimination succeeds only in special cases, e.g., if the variables of the right-hand sides are a subset of the left-hand sides. If this condition is not met, we must employ a more sophisticated method, which is integrated in the *predicate compiler*.

Predicate Compiler

Another contribution is the *predicate compiler* [16], which translates specifications of inductive predicates (the introduction rules) into executable equational theorems for Isabelle's code generator. The translation is based on the notion of *modes*. A mode partitions the arguments into input and output. For a given predicate, the predicate compiler infers the set of possible modes such that all terms are ground during execution. The code equations implement a Prolog-style depth-first execution strategy. Lazy sequences are used to express the nondeterministic behavior of the execution in the functional language.

For example, the predicate $append_P$ of type $\alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list \Rightarrow bool$ corresponding to the function *append* is defined by the two rules, $append_P\ []\ ys\ ys$ and $append_P\ xs\ ys\ zs \Longrightarrow append_P\ (x \cdot xs)\ ys\ (x \cdot zs)$. This predicate supports several modes:

- From the first two arguments *xs* *ys*, we can compute the third argument, essentially evaluating $xs +\!\!+ ys$. This corresponds to the mode $i \Rightarrow i \Rightarrow o \Rightarrow bool$ where i denotes input and o output.
- Inversely, we can enumerate the set of the first two lists given the third list *zs*, i.e., compute $\{(xs, ys).\ xs +\!\!+ ys = zs\}$: $o \Rightarrow o \Rightarrow i \Rightarrow bool$.
- Given all three arguments, we can check whether the first two lists appended equal the third: $i \Rightarrow i \Rightarrow i \Rightarrow bool$.

- Or we can allow other modes that combine computing and checking, e.g., modes $i \Rightarrow o \Rightarrow i \Rightarrow bool$ and $o \Rightarrow i \Rightarrow i \Rightarrow bool$.

Since its initial description [16], we improved the compiler in various aspects.

First, we have enriched modes to handle the ubiquitous product types in a finer-grained manner. For example, given a relation R of type $\alpha \times \beta \Rightarrow bool$ it previously was restricted to two possible modes: The argument was considered as either input or output. Our new implementation also allows modes, where some components of a tuple are input and others output, which enables us to enumerate $\{y.\ R\ (x, y)\}$ for some relation R.

Second, we also improved the compilation scheme. The previous one sequentially checked which of the introduction rules were applicable. Hence, the input values were repeatedly matched against the patterns of the terms in the conclusion of each introduction rule. This compilation scheme is fairly simple, but often rather inefficient: Given an inductive predicate with n introduction rules, the input arguments underwent pattern-matching n times. For large specifications, such as a full-fledged Java semantics [78] with 88 rules, this naive compilation made execution virtually impossible due to the large number of rules. To obtain an efficient code expression, we modified the compilation scheme to partition the rules by patterns of the input values first and then only compose the matching rules. This resembles similar techniques in Prolog compilers, such as clause indexing and switch detection. We achieved dramatic performance improvements with this modification in various applications.

Third, the predicate compiler was originally limited to the restricted syntactic form of introduction rules. We added some preprocessing that transforms definitions in predicate logic to a set of introduction rules. Thus the predicate compiler becomes applicable to predicates specified by other means than inductive definitions. Furthermore, (recursive) functions can be automatically preprocessed to (inductively-defined) relations by *flattening* nested function terms into a set of premises. We describe this improvement in more detail in sections 4.2.2 and 4.2.3.

Fourth, the predicate compiler now offers program refinement similar to the code generator.

Last, mode annotations restrict the generation of code equations to modes of interest. This is useful because the set of modes is exponential in the number of arguments of a predicate. Therefore, the space and time consumption of the underlying mode inference algorithm grows exponentially in that number. For larger applications, the plain construction of this set of modes demands all available resource of memory. To sidestep this limitation, modes can be declared and hence they are not inferred, but only checked to be consistent.

2.2.4 Contributions to Isabelle's Code Generation

The main purpose of this chapter was to provide a gentle introduction to the Isabelle system and its code generation facilities, upon which Quickcheck builds. One of the key points for the added value of Quickcheck in Isabelle is the code generator's ability to generate executable code for many specifications.

This ability is thanks to numerous improvements to the code generation setup in Isabelle/HOL in the scope of this thesis. We just scratch the surface of some improvements of different character here:

- *Correcting the type erasure in the serialization.*

  We lifted a previously existing limitation of the code generator, concerning the type erasure in the *serialization* phase for Haskell code [53, §3.4]. Serialization is the last phase where the generated code in an intermediate language is turned into concrete source code. During the translation process, the expressions in the intermediate language are annotated with explicit types. However when printing the concrete source code, the types are usually omitted, as they can be commonly inferred by the compiler afterward and the generated code resembles more closely to a human-written style. Unfortunately, in the presence of type classes in Haskell source code, omitted types cannot be not re-inferred in general. Here is a contrived example to show the problematic case.

  class $c =$
  fixes $f :: \alpha$

  instance $nat :: c$ $[\ldots]$

  definition $g :: nat$
  where $g = (\text{let } x = (f :: nat) \text{ in } 1)$

  If the definition of g is used for code generation, the annotation for f in the definition must be preserved in the generated code, because the type is required to disambiguate the instance of f, but it cannot be inferred from g's type alone. Previously, the serializer to Haskell code had a simple incomplete heuristics to add types in some special cases [53, §3.4.2], but it missed cases such as the one above. Quickcheck's intensive application of the code generator forced us to replace the incomplete heuristics by a solution to handle all cases correctly. The new solution applies type inference on the whole expression after the types are erased. It then compares the expression with inferred types against the original expression with the known types and adds a minimal number of type annotations to the Haskell code to ensure that the Haskell compiler is never confronted with ambiguous expressions.

- *Adding a code preprocessor for set comprehensions*

  Sets are refined by lists for code generation. This representation requires sets to be constructed by some basic operations, e.g., union, intersection or Cartesian product, but disallows general set comprehensions $\{x.\ P\ x\}$ for some arbitrary boolean predicate P.

  However, many set comprehensions can be expressed by the implemented basic operations. For example, $\{x.\ (x \in A\ \wedge\ x \in B)\ \vee\ x \in C\}$ is equivalent to $(A \cap B) \cup C$. A less constructed example is the definition of the concatenation of two formal languages A and B:

  $$conc\ A\ B = \{xs +\!\!+ ys \mid xs\ ys.\ xs \in A\ \wedge\ ys \in B\}$$

This definition can also be expressed with the executable operations as

$$conc\ A\ B = (\lambda(xs, ys).\ xs +\!\!+ ys)\ `\ (A \times B)$$
where $f\ `\ A = \{f\ x \mid x \in A\}$ and $A \times B = \{(x, y).\ x \in A\ \wedge\ y\ \in\ B\}$

A proof procedure in the code preprocessor automatically rewrites set comprehensions to expressions built from the basic operations. This makes many set comprehensions executable.

- *Refinement of existing specifications towards executability.*

  A further step was to improve code generation setup of specific concepts to enable their evaluation or to improve the evaluation's performance. We just mention two of these refinement here: For refutation of conjectures about multisets, we provided setup that allows us to execute the common set operations on multisets, e.g., union, intersection and set difference. We also inspected existing setup for code generation for possible performance improvements. For example, the predicate *identity-on*, which expresses the identity relation over a predicate P, is defined in a way that makes the evaluation of this constant very slow:

  definition *identity-on* $:: (\alpha \Rightarrow bool) \Rightarrow \alpha \times \alpha \Rightarrow bool$
  where *identity-on* $P = (\lambda y.\ \exists x.\ P\ x\ \wedge\ y = (x, x))$

  By simply stating the alternative equivalent definition,

  $$identity\text{-}on\ P = (\lambda(y_1, y_2).\ y_1 = y_2\ \wedge\ P\ y_1),$$

  and employing it for the evaluation increased the evaluation's performance of *identity-on* dramatically.

Chapter 3

Random and Exhaustive Testing

This chapter describes Quickcheck's main infrastructure and two of four testing approaches: random and exhaustive testing.

Figure 3.1 depicts Quickcheck's main components and artifacts during an invocation. Quickcheck takes the user's conjecture as input and returns a falsifying assignment of the free variables as counterexample if the conjecture is found to be invalid. Most workload of this task is delegated to Isabelle's code generator and the ML interpreter. Quickcheck's main responsibility is to transform the conjecture into a *test program* that attempts to refute the conjecture. Isabelle's code generator is responsible to produce the ML source code of this program; the ML interpreter evaluates this program and directly returns the counterexample.

Delegating the task to the code generator and the interpreter makes Quickcheck lightweight and simple to implement. For example, as Quickcheck shares Isabelle's code generation with other applications, it adopts all the specific configuration of the code generator. Furthermore, it does not need to take care of many technical issues: As it only processes the conjecture, it never needs to compute the call graph and retrieve the code equations of functions that are used in the conjecture. At the same time, Quickcheck's reliance on the code generator impairs its ability to do specific program transformations.

3.1 From Conjectures to Test Programs

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

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

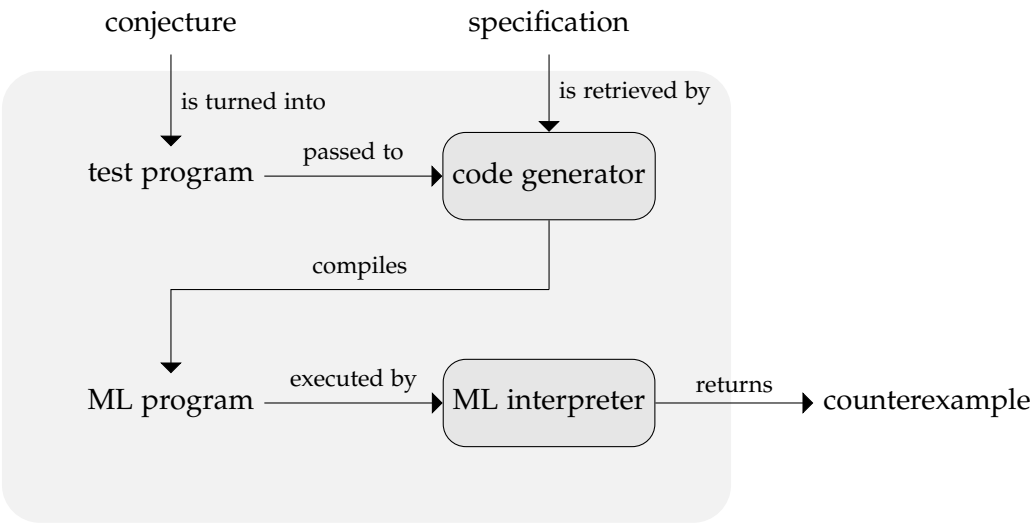


Figure 3.1: Main components and constructed artifacts of Isabelle's Quickcheck

Given a function c that checks the conjecture for a single value, the generator *exhaustive c* yields a function that checks the conjecture for all values up to a given bound. For further user interaction, a counterexample of type τ is mapped to a fixed type *result* using the function $\mathit{reify} :: \tau \Rightarrow \mathit{result}$. We describe the generators and the reification in detail in §3.2 and §3.3.4. A simple test program for a conjecture C with a single variable x can be expressed as

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

Test programs are improved by taking the common structure of conjectures into account, as a list of premises and a conclusion. If a premise does not depend on a free variable, the generation of values for this free variable can be postponed until after checking the premise. Thus, Quickcheck optimizes the test program so that it generates the values for each variable as late as possible. This improvement is in particular important for exhaustive testing, as it turns the brute force enumeration into a *backtracking* one, in which a large number of candidates are avoided by a single test.

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

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

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

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

In the presence of (multiple) premises, this interleaving of generation and evaluation already improves its performance dramatically. In chapter 4, we optimize the generation and evaluation of this kind of conjecture further.

3.2 Test Data Generators

Quickcheck defines test data generators for random and exhaustive testing (§3.2.2 and §3.2.3). For both strategies, Quickcheck supports the definition of generators: Generators of inductive datatypes (§3.2.4) are defined automatically, and generators of arbitrary type definitions (§3.2.5) require some guidance from the user.

Both approaches build on a family of test data generators. These test data generators are type-based, i.e., there is exactly one generator for each type. Generators for a complex type τ are constructed following its structure, which is nicely described using type classes in Isabelle [120]. For example, given a generator for polymorphic lists α *list* and a generator for the type of natural numbers (type *nat*), the generator for *nat list* is implicitly composed from those two generators by the type class mechanism (cf. §2.1.3).

Generators are combined by *chaining* and *choosing between alternatives*. The generators express a nondeterministic (branching) computation. The generators' operations are closely related to operations on a *plus monad*, a generalization of the ideas for nondeterministic computations in [114].

3.2.1 Nondeterministic Computations

Nondeterministic computations provide a general basis for the specific purpose of test data generation. We give an overview how nondeterministic computations are expressed in a functional language. This summarizes some descriptions distributed over a number of papers [45, 56, 109, 114].

Monads [115] are used to express computational effects in a functional programming language. A type constructor M forms a monad if it is equipped with the two operations, $return :: \alpha \Rightarrow \alpha\ M$ and $bind$, $(\gg\!=) :: \alpha\ M \Rightarrow (\alpha \Rightarrow \beta\ M) \Rightarrow \beta\ M$, and satisfies the *monad laws*:

$$return\ a \gg\!= f = f\ a$$
$$m \gg\!= return = m$$
$$(m \gg\!= k_1) \gg\!= k_2 = m \gg\!= (\lambda a.\ k_1\ a \gg\!= k_2)$$

The monad laws express that *return* is left and right unit and *bind* is associative. Monads are widely used to express computations with state, exceptions or output. As we focus on nondeterministic computations, we also employ two further operations for failure and choice: *mzero* is the computation with no successful results, i.e., a failing computation; and *mplus* combines two alternative computations, i.e., we allow branching computations:

$$mzero :: \alpha\ M$$
$$mplus :: \alpha\ M \Rightarrow \alpha\ M \Rightarrow \alpha\ M$$

The two functions *mzero* and *mplus* (with $\oplus$ as infix notation) form a monoid:

$$mzero \oplus m = m$$
$$m \oplus mzero = m$$
$$m_1 \oplus (m_2 \oplus m_3) = (m_1 \oplus m_2) \oplus m_3$$

Together with *return* and *bind*, the four operations form a *plus monad* if the laws

$$zero \gg\!= f = zero, \text{ and either}$$
$$(m_1 \oplus m_2) \gg\!= f = (m_1 \gg\!= f) \oplus (m_2 \gg\!= f) \text{ (left distribution) or}$$
$$(return\ a) \oplus m = return\ a \text{ (left catch) hold.}^1$$

Nondeterministic computations are expressed employing these four primitives. For example, the function *any* chooses nondeterministically one among a list of alternative values for some plus monad M:

$$any :: \alpha\ list \Rightarrow \alpha\ M$$
$$any\ [] = mzero$$
$$any\ (x \cdot xs) = return\ x \oplus any\ xs$$

A simple implementation of a plus monad is the *list monad*. In this monad, alternative values of the computations are expressed by *a list of successes*:

$$return\ x = [x]$$
$$xs \gg\!= f = [y.\ y \leftarrow f\ x,\ \ x \leftarrow xs] = concat\ (map\ f\ xs)$$
$$mzero = []$$
$$mplus\ xs\ ys = xs +\!\!+ ys$$

where $concat :: \alpha\ list\ list \Rightarrow \alpha\ list$ is defined by
$$concat\ [] = []$$
$$concat\ (xs \cdot xss) = xs +\!\!+ (concat\ xss)$$

Although the list monad suffices to express nondeterministic behavior, its behavior (in a strict language) is very inefficient because the list of all possible alternatives is stored in the memory. In our application, testing a conjecture exhaustively with a million test cases is expressed as a computation with a million alternatives. In the case of such a computation, modeling the computation with the list monad would quickly take up all physical memory. To obtain an appropriate and efficient model for nondeterministic computations, we make use of two other plus monads, the *option monad* and the *continuation monad*. The option monad only returns the first successful computation:

$$return\ x = Some\ x$$
$$(Some\ x) \gg\!= f = f\ x$$
$$None \gg\!= f = None$$
$$mzero = None$$
$$mplus\ (Some\ x)\ y = Some\ x$$
$$mplus\ None\ y = y$$

[1] For our purposes, we assume that if one of the two laws holds, we call it *a plus monad*. In a more refined view of plus monads [121], a structure is called a *plus monad* if left distribution holds and an *or monad* if left catch holds.

For more efficient backtracking, we write our nondeterministic programs with continuation-passing style. A function in continuation-passing style does not return its result to its caller, but takes as an additional argument the continuation, which takes the computed result as argument and continues the computation. A continuation monad can be combined with different computations. To describe a continuation monad in its full generality requires rank-2 polymorphism. As Isabelle's logic HOL is limited to rank-1 polymorphism, we present continuation monads in a restricted setting, but expressible in HOL. We assume τ to be some fixed type. A continuation can be described as datatype *cont* with one constructor $Cont :: ((\alpha \Rightarrow \tau) \Rightarrow \tau) \Rightarrow \alpha\ cont$ and its destructor $run :: \alpha\ cont \Rightarrow (\alpha \Rightarrow \tau) \Rightarrow \tau$, such that $run\ (Cont\ c) = c$. The monad operations are then defined by

$$return\ x = Cont\ (\lambda f.\ f\ x)$$
$$m \gg\!= f = Cont\ (\lambda c.\ run\ m\ (\lambda x.\ run\ (f\ x)\ c))$$

If we are given two operations, *failure* and *choice*, on type τ, we can define a *continuation plus monad*:

$$mzero = Cont\ (\lambda f.\ failure)$$
$$mplus\ m_1\ m_2 = Cont\ (\lambda f.\ choice\ (run\ m_1\ f)\ (run\ m_2\ f)$$

The functions *failure* and *choice* resemble *mzero* and *mplus* but do not require τ to be a monad. This forms the basis for expressing nondeterministic computations.

The test programs have a special characteristic compared with nondeterministic computations, which motivates the chosen plus monads employed in our exhaustive generators (§3.2.3).

3.2.2 Basic Random Generators

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

$$return :: \alpha \Rightarrow \sigma \Rightarrow \alpha \times \sigma$$
$$return\ x\ s = (x, s)$$
$$\gg\!= :: (\sigma \Rightarrow \alpha \times \sigma) \Rightarrow (\alpha \Rightarrow \sigma \Rightarrow \beta \times \sigma) \Rightarrow \sigma \Rightarrow \beta \times \sigma$$
$$(f \gg\!= g)\ s = g\ x\ s'\ where\ (x, s') = f\ s$$

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

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

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

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

Given a random seed, the test program with random generators produces one random value, tests the conjecture with this value and returns the evaluation's result and the next random seed. Quickcheck calls this test program for a fixed number of times and if it finds a counterexample, it returns this counterexample to the user. Otherwise, it continues checking the conjecture with increasing size and a fixed number of tests, until the user-provided limit for the size is reached.

3.2.3 Basic Exhaustive Generators

Similar to random generators, exhaustive generators are provided by the type class *exhaustive* with a function *exhaustive* of type $(\tau \Rightarrow result\ option) \Rightarrow nat \Rightarrow result\ option$. In contrast to random generators, which only yield one value, the exhaustive generators produce many values with a nondeterministic computation. To make this computation efficient, the exhaustive generators are expressed with continuations: They take a continuation (which ultimately checks the conjecture), and evaluate it with all values of type τ up to the given size.

Generators are chained by nesting the continuations. For example, for a given continuation c and size i, the generator for product types is defined by

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

Since only the order of alternatives, but not their weights, is relevant for exhaustive testing, generators can be simply combined with the binary operation $\sqcup$, which chooses the first *Some* value when evaluating from left to right:

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

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

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

If we evaluate the definition of the $\sqcup$ operation in a strict language with call-by-value strategy, the second argument y is evaluated even if it is not required. However, Quickcheck expresses the $\sqcup$ operation with the case expression

$$x\ \sqcup\ y = (\text{case}\ x\ \text{of}\ Some\ x' \Rightarrow Some\ x' \mid None \Rightarrow y)$$

and inlines this definition, and hence creates a test program that evaluates the alternatives only if required.

Relating to the descriptions in §3.2.1, the exhaustive generators can be described as a continuation plus monad combined with the option monad, but for the definition of the generators, we omitted wrapping the continuation constructor *Cont*. For the nondeterministic computation of the test program, we view a

property's evaluation that returns true and hence does not yield a counterexample, as a failing computation. Chaining of two generators and choosing between two generators essentially implement the operations *bind* and *mplus*, but the operations for exhaustive generators take the size as further argument.

3.2.4 Generators for Inductive Datatypes

Commonly, new types are defined by datatype declarations. For these types, Quickcheck automatically constructs random and exhaustive generators upon the type's definition. The construction of random generators has been described in [14], so we only sketch the construction of exhaustive generators here.

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

$$\begin{aligned} &exhaustive_{\alpha\ list}\ c\ i = \text{if } i = 0 \text{ then } None \text{ else } (c\ Nil\ \sqcup \\ &\quad exhaustive_{\alpha}\ (\lambda x.\ exhaustive_{\alpha\ list}\ (\lambda xs.\ c\ (Cons\ x\ xs))\ (i-1))\ i) \end{aligned}$$

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

Completeness of the Generators

As the test data generators are defined in the logical framework, we can prove the completeness of the test data generators, i.e., we can prove that generators test a given property for all possible values up to a given bound.

To express this property, we first must make the notion of size more explicit. We define a type class *size* equipped with a function $size :: \tau \Rightarrow nat$, which captures the maximal depth of constructors of the value. For example, the size of lists defined as

$$\begin{aligned} &size_{\alpha\ list}\ [] = 1 \\ &size_{\alpha\ list}\ (x \cdot xs) = max\ (size_{\alpha}\ x)\ (size_{\alpha\ list}\ xs) \end{aligned}$$

We now would like to show the following property:

$$\begin{aligned} &(\exists v.\ size_{\tau}\ v \leq n \wedge is\text{-}some\ (f\ v)) \longleftrightarrow is\text{-}some\ (exhaustive_{\tau}\ f\ n) \\ &\text{where } is\text{-}some\ (Some\ x) = True,\ is\text{-}some\ (None) = False \end{aligned}$$

The function *exhaustive* checks the function f for all values whose size is less than or equal to n. In other words, the function *exhaustive* covers the domain of small values *completely*.

We would like to prove this property for all datatypes τ in Isabelle once and for all. However, this is not possible, because the logic permits to express this property in this full generality. Although the construction of the exhaustive generators follows a fixed scheme, this scheme is only defined on the metalevel. In the logical system, Quickcheck only provides the concrete instances for every datatype when the datatype is declared. Nevertheless, we can show the property for every declared datatype with a *proof procedure* that proves the property after the datatype's definition. For this purpose, we capture the property by the type class *complete*:

class *complete* = *exhaustive* + *size* +
assumes $(\exists v.\ size_\tau\ v \leq n \wedge is\text{-}some\ (f\ v)) \longleftrightarrow is\text{-}some\ (exhaustive_\tau\ f\ n)$

By showing that the type constructor *list* is an instance of this class, we prove the completeness of the test data generator for polymorphic lists. The instance proof for lists yields no surprises: It is proved by induction on n and deploys the type class axiom of *complete* and the *size* definitions.

A general proof procedure for arbitrary datatypes would essentially follow the lines of the instance proof for the *list* type constructor, although it might be technically challenging to implement. As this is only of minor benefit for the counterexample generator, we did not pursue this any further. The formalization of the presented example can be inspected at src/HOL/Quickcheck_Examples/Completeness.thy in the Isabelle repository.

3.2.5 Generators for Arbitrary Type Definitions

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

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

Accepting these *non-free constructors* of (α,β) *rbt* as constructing functions, Quickcheck provides random and exhaustive generators for (α,β) *rbt* that produce values starting with the *empty* tree and executing a sequence of *insert* operations. The random generator chooses the key and value for the *insert* operation randomly from the set of possible values, whereas the exhaustive generator enumerates all possible keys and values (up to a given size) for the *insert* operations. Since Quickcheck does not take the equivalence classes of the non-free constructors into account, the generator might produce many identical values. For example, the equation $insert\ k_1\ v_1\ (insert\ k_2\ v_2\ empty) = insert\ k_2\ v_2\ (insert\ k_1\ v_1\ empty)$ holds if $k_1 \neq k_2$. Ignoring this equivalence, the generator produces some trees at least twice: once when engaging the sequence of two *insert* operations with (k_1, v_1) and (k_2, v_2), and for a second time, engaging the sequence of the two *insert* operations swapped. Furthermore, there is no guarantee that the generator covers all values of the underlying representation type up to the given size.

Generators for Functions

For random testing, the generator for functions employs a special construction. As only a fraction of the domain of the generated function is queried, the function's return values are generated only when the function is called. To keep the

function's return values consistent, previously generated return values for functions are stored. At the beginning of each test case, Quickcheck creates a reference value that stores an empty table and a random seed. Every time, the generated function is called, Quickcheck either returns the value that has been stored in the table for some previous function call with the same argument or generates a fresh random value using the stored random seed and stores it in the table. This implementation in Standard ML is then combined with the test program generated from Isabelle.

For exhaustive testing, function values are generated by constructing constant functions and applying a sequence of function updates:

$$\begin{aligned}
&\mathit{exhaustive}'\ (c :: (\alpha \Rightarrow \beta) \Rightarrow \mathit{result\ option})\ i\ j = (\text{if } i = 0 \text{ then } \mathit{None} \\
&\quad \text{else } \mathit{exhaustive}_\beta\ (\lambda b.\ c\ (\lambda x.\ b))\ j)\ \sqcup \\
&\qquad (\mathit{exhaustive}'\ (\lambda f.\ \mathit{exhaustive}_\alpha\ (\lambda a.\ \mathit{exhaustive}_\beta \\
&\qquad\quad (\lambda b.\ c\ (f(a := b)))\ j)\ j)\ (i - 1)\ j) \\
&\mathit{exhaustive}_{\alpha \Rightarrow \beta}\ c\ i = \mathit{exhaustive}'\ c\ i\ i \\
&\text{where } f(a := b) = (\lambda x.\ \text{if } x = a \text{ then } b \text{ else } f\ x)
\end{aligned}$$

In contrast to the function generator for random testing, the function generator for exhaustive testing eliminates the need to tie some ML implementation with the generated code.

3.3 Extensions of the Infrastructure

So far, we presented the core parts of Quickcheck. In this section, we touch on further aspects: testing of parametrized and polymorphic conjectures and conjunctures with type classes, reification of results, and underspecified functions.

3.3.1 Parametrized Conjectures

Locales [9] in Isabelle allow us to prove theorems abstractly, relative to a set of *fixed parameters and assumptions*. Interpretation of locales transfers theorems from their abstract context to other (concrete) contexts by instantiating the parameters and proving the assumptions.

For example, the locale *antisym* captures that a relation R is antisymmetric:

$$\textsf{locale}\ \mathit{antisym} = \textsf{fixes}\ R\ \textsf{assumes}\ R\ x\ y \Longrightarrow R\ y\ x \Longrightarrow x = y$$

Two concrete examples of the *antisym* locale are the polymorphic equality relation on any type α and the order relation on natural numbers. When the user provides a proof that these relations are antisymmetric, the system registers the two interpretations of the locale *antisym*, so that all facts derived in the locale context are available for the two concrete examples.

Quickcheck has two strategies to refute a conjecture in a locale context:

- *Instantiate to known interpretations.* Quickcheck instantiates the conjecture to all interpretations that have been declared by the user, and then tests the resulting conjectures sequentially.

- *Expand the parameters and axioms.* Quickcheck tries to refute the abstract conjecture by adding all locale assumptions to the abstract conjecture. As a consequence, the test programs first search for variable assignments that fulfill the locale assumptions, and then check the conjecture.

As an example, we conjecture that every antisymmetric relation is transitive, i.e., $\forall x\ y.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$ holds in *antisym*.

Following the first approach, Quickcheck instantiates the conjecture to the two interpretations, and checks the two conjectures $\forall x\ y.\ x = y \Longrightarrow y = x \Longrightarrow x = z$, and $\forall x :: nat\ y :: nat.\ x \leq y \Longrightarrow y \leq x \Longrightarrow x = z$. Following the second approach, Quickcheck also tests the expanded conjecture:

$$\forall R.\ (\forall x\ y.\ R\ x\ y \Longrightarrow R\ y\ x \Longrightarrow x = y) \Longrightarrow \forall x\ y.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$$

Coincidentally, the conjecture holds on the two interpretations; however for the expanded conjecture, Quickcheck finds the counterexample, $R = (\lambda x\ y.\ (x = \mathsf{a}_2 \wedge y = \mathsf{a}_3) \vee (x = \mathsf{a}_3 \wedge y = \mathsf{a}_1))$, $x = \mathsf{a}_2$, $y = \mathsf{a}_3$, $z = \mathsf{a}_1$.

The two options complement each other: The first option only uses the declared interpretations, and can easily miss simple counterexamples, as we have seen in the example. The second option is more general and can be applied even if no interpretations in the development were declared. However, searching for models by testing can be cumbersome, as the testing methods are inappropriate to find functions given a set of constraints. Therefore, the second option only succeeds if small models exist. In contrast, choosing declared interpretations renders the search for models unnecessary and reduces the search space significantly, making Quickcheck also applicable for large models.

In Isabelle, it is still technically very complicated to obtain an executable specification for interpretations. This spoils the otherwise overall impression of Quickcheck's enhancements with parametrized conjectures. Unfortunately, Quickcheck is inherently limited by this weakness of the Isabelle system.

3.3.2 Conjectures with Type Classes

Polymorphic conjectures can also be restricted by sort constraints, i.e., the property is only stated for types with specific classes. For example, one might believe that any order is linear and state the conjecture

$$(a :: \alpha :: order) \leq b\ \wedge\ b \leq c \Longrightarrow a \leq c$$

Although type classes and locales serve a very similar purpose, they are encoded differently in the system. In particular, the code generator translates type classes, but is unaware of the locale mechanism. This circumstance allows Quickcheck to follow only an approach, similar to the first approach of parametrized conjectures. Quickcheck instantiates the conjecture with all types that fulfill the sort constraints and have been registered in the development.

The second approach for parametrized conjectures suggests to enumerate all small types and the type classes' operations that fulfill the class constraints. However, enumerating types is not possible naively as this enumeration is not expressible in the logic. If Quickcheck could choose an alternative translation of types and

type classes, it could express the enumeration of types in the test program. However, as Quickcheck's translation relies heavily on the code generator's translation, we cannot modify the translation for types and type classes.

To strengthen Quickcheck on conjectures with type classes nonetheless, Isabelle provides a special library of small types for some of the typical classes arising in formalizations. This improves the situation with type classes to some extent, but handling conjectures with type classes still remains a weak point of Quickcheck.

3.3.3 Polymorphic Conjectures

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

The implementation for refuting quantified formulas over a finite type is based on the type class *enum*. Type classes allow us to obtain implementations for more complex types by composition. For example, the type $\alpha \times \beta \Rightarrow \gamma$ is finite if α, β and γ are finite types. The type class *enum* provides three operations for every finite type τ: The operation $\mathit{univ} :: \tau\ \mathit{list}$ enumerates the finite universe; the operations $\mathit{all} :: (\tau \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ and $\mathit{ex} :: (\tau \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$ check universal and existential properties. The existential and universal quantifiers could be expressed just with $\mathit{univ} :: \tau\ \mathit{list}$, i.e., $\forall x :: \tau.\ P\ x = \mathit{list\text{-}all}\ P\ (\mathit{univ} :: \tau\ \mathit{list})$. Due to the strict evaluation of ML, this would be rather inefficient: The evaluation would first construct a finite (but potentially large) list of values, and then check them sequentially. To avoid the large intermediate list, we implement the quantifiers using continuations, similar to the construction of the exhaustive generators (cf. §3.2.3). For example, the universal quantifiers for product and sum type are implemented by

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

For most types, the implementation is straightforward. For the function type, it is a bit more involved. To construct the set of all functions $\alpha \Rightarrow \beta$, we must create all possible mappings, i.e., all lists of type $\beta\ \mathit{list}$ with the same length as $\mathit{univ} :: \alpha\ \mathit{list}$, and transform those lists into functions. The function *fun-of dom range* defines a function that maps the values in the list *dom* to their corresponding value in the list *range*. This can be defined with Isabelle's standard operations by

definition $\mathit{fun\text{-}of} :: \alpha\ \mathit{list} \Rightarrow \beta\ \mathit{list} \Rightarrow \alpha \Rightarrow \beta$
where $\mathit{fun\text{-}of\ dom\ range} = \mathit{the}\ (\mathit{map\text{-}of}\ (\mathit{zip\ dom\ range}))$

The function *nlists n xs* defines the set of all lists with length *n* and values from *xs*. We check statements $\forall xs \in \mathit{nlists\ n\ enum}.\ P\ xs$ and $\exists xs \in \mathit{nlists\ n\ enum}.\ P\ xs$ efficiently by iterating over all possible values employing the basic *all* and *ex* combinators using the following equations:

definition *ex-nlists* :: $((\beta :: enum)\ list \Rightarrow bool) \Rightarrow nat \Rightarrow bool$
where $ex\text{-}nlists\ P\ n = (\exists xs \in set\ (nlists\ n\ univ_\beta).\ P\ xs)$

lemma [code]: $ex\text{-}nlists\ P\ n =$
$(\text{if } n = 0 \text{ then } P\ [] \text{ else } ex_\beta\ (\lambda x.\ ex\text{-}nlists\ (\lambda xs.\ P\ (x \cdot xs))\ (n - 1)))$

definition *all-nlists* :: $((\beta :: enum)\ list \Rightarrow bool) \Rightarrow nat \Rightarrow bool$
where $all\text{-}nlists\ P\ n = (\forall xs \in set\ (nlists\ n\ univ_\beta).\ P\ xs)$

lemma [code]: $all\text{-}nlists\ P\ n =$
$(\text{if } n = 0 \text{ then } P\ [] \text{ else } all_\beta\ (\lambda x.\ all\text{-}nlists\ (\lambda xs.\ P\ (x \cdot xs))\ (n - 1)))$

Combining *fun-of* and *all-nlists* and *ex-nlists*, we finally obtain quantifiers for finite functions:

$all_{\alpha \Rightarrow \beta}\ P = all\text{-}nlists\ (\lambda bs :: \beta\ list.\ P\ (fun\text{-}of\ univ_\alpha\ bs))\ (card\ \alpha)$

$ex_{\alpha \Rightarrow \beta}\ P = ex\text{-}nlists\ (\lambda bs :: \beta\ list.\ P\ (fun\text{-}of\ univ_\alpha\ bs))\ (card\ \alpha)$

The test data generators for functions $\alpha \Rightarrow \beta$ with infinite types α and β were presented in §3.2.5. However, if the types α and β are finite, Quickcheck uses another exhaustive test data generator for functions $\alpha \Rightarrow \beta$, which follows the enumeration scheme of $univ_{\alpha \Rightarrow \beta}$ and $all_{\alpha \Rightarrow \beta}$. Similarly to $univ_{\alpha \Rightarrow \beta}$ and $all_{\alpha \Rightarrow \beta}$, it provides a *complete enumeration* of all functions over a finite type. In contrast to $all_{\alpha \Rightarrow \beta}$, the test data generator also returns the counterexample as an optional value. It is implemented with the type class *check-all*:

class *check-all* = *enum* +
fixes $check\text{-}all :: (\alpha \Rightarrow result\ option) \Rightarrow result\ option$

To enumerate all finite functions, we use the functions *fun-of* and *checkall-nlists*:

$checkall\text{-}nlists :: ((\alpha :: check\text{-}all)\ list \Rightarrow result\ option) \Rightarrow nat \Rightarrow result\ option$
$checkall\text{-}nlists\ f\ n = \text{if } n = 0 \text{ then } f\ []$
$\quad \text{else } check\text{-}all\ (\lambda x.\ checkall\text{-}nlists\ (\lambda xs.\ f\ (x \cdot xs))(n - 1)))$

$check\text{-}all_{\alpha \Rightarrow \beta}\ f =$
$\quad checkall\text{-}nlists\ (\lambda ys :: \beta\ list.\ f\ (fun\text{-}of\ univ_{\alpha\ list}\ ys))\ (card\ \alpha)$

3.3.4 Reification

To present the counterexample to the user with proper Isabelle term syntax, the counterexample must be displayed with Isabelle's pretty printer. To employ the pretty printer, the execution's result must be transformed into an Isabelle term. One option for the implementation would be that the test program prints the counterexample as a string and Quickcheck employs a parser to construct the term. However, as the ML environment, in which the test program runs, is tightly integrated with Isabelle, it is not necessary to make the detour by printing and parsing. Instead, the test program itself is able to return an Isabelle term, which is passed directly to the pretty printer. To generate an ML test program that returns an Isabelle term, we must encode Isabelle's internal type and term representations within the logic. The necessary datatypes to encode monomorphic types and ground terms are defined by

datatype *type* = *Type string* (*type list*)
datatype *term* = *App term term* | *Const string type*

The two type classes *typerep* and *term-of*

class *typerep* = **fixes** *typerep* :: α *it* $\Rightarrow$ *type*
class *term-of* = *typerep* + **fixes** *term-of* :: $\alpha \Rightarrow$ *term*

provide functions to obtain the type and term representation of a value, where the phantom type α *it* with the single value T is used to embed types as terms. Instances of *typerep* are automatically derived for all types, and instances of *term-of* for all inductive datatypes. The construction of those functions is straightforward. For example, for the boolean and product type, these functions are

$$\begin{aligned}
&\mathit{typerep}_{\mathit{bool}}\ (T :: \mathit{bool}\ \mathit{it}) = \mathit{Type}\ \text{“}\mathit{bool}\text{”}\ [] \\
&\mathit{typerep}_{\alpha\times\beta}\ (T :: (\alpha \times \beta)\ \mathit{it}) = \mathit{Type}\ \text{“}\mathit{prod}\text{”}\ [\mathit{typerep}\ (T :: \alpha\ \mathit{it}), \mathit{typerep}\ (T :: \beta\ \mathit{it})] \\
&\mathit{term\text{-}of}_{\mathit{bool}}\ \mathit{False} = \mathit{Const}\ \text{“}\mathit{False}\text{”}\ (\mathit{Type}\ \text{“}\mathit{bool}\text{”}\ []) \\
&\mathit{term\text{-}of}_{\mathit{bool}}\ \mathit{True} = \mathit{Const}\ \text{“}\mathit{True}\text{”}\ (\mathit{Type}\ \text{“}\mathit{bool}\text{”}\ []) \\
&\mathit{term\text{-}of}_{\alpha\times\beta}\ (a, b) = \mathit{App}\ (\mathit{App} \\
&\quad (\mathit{Const}\ \text{“}\mathit{Pair}\text{”}\ (\mathit{typerep}\ (T :: (\alpha \Rightarrow \beta \Rightarrow \alpha \times \beta)\ \mathit{it})))\ (\mathit{term\text{-}of}\ a))\ (\mathit{term\text{-}of}\ b)
\end{aligned}$$

The *result* type and the *reify* function, introduced in §3.1, are simply abbreviations for this presentation: *result* abbreviates *term list*, and given a conjecture with free variables $x_1, \ldots, x_n$, *reify* abbreviates [*term-of* x_1, ..., *term-of* x_n].

3.3.5 Underspecified Functions

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

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

is valid for all lists *xs* and *ys*, even if *xs* and *ys* are *Nil*. In this special case, left-hand and right-hand side are equal because they reduce to the same term *hd* []. In contrast, the conjecture $\mathit{hd}\ (\mathit{map}\ f\ \mathit{xs}) = f\ (\mathit{hd}\ \mathit{xs})$ is valid only if $\mathit{xs} \neq []$, because for $\mathit{xs} = []$, as the left-hand and right-hand sides can evaluate to different values: For the constant function $f = (\lambda x.\ c)$ with $c \neq \mathit{hd}\ []$, the right-hand side reduces to $f\ (\mathit{hd}\ []) = c$, which is unequal to the value of the left-hand side *hd* []. Hence, this conjecture is invalid.

To uncover counterexamples with underspecified functions, we slightly change the test programs. The evaluation of underspecified functions in Standard ML yields a *match exception* if it encounters a call to such a function and no pattern matches the given arguments. The test program catches this exception. If we are interested in possible counterexamples due to underspecification, we return the

values that yield the exception as counterexample. Alternatively, if we are only interested in genuine counterexamples, we continue to search for other values. In the presence of underspecified function definitions, Quickcheck cannot determine if a counterexample is genuine or spurious if it was found by the evaluation where exception values occurred. Therefore, it marks the counterexample as *potentially spurious*. On the two conjectures above, Quickcheck returns the potentially spurious counterexamples $xs = [], ys = []$ and $xs = [], f = \lambda x.\ \mathsf{a}_1$. Nevertheless, these potentially spurious counterexamples are useful in two ways: First, it makes users aware that the choice[2] how the underspecified function is turned into a total function might be crucial for the validity of this conjecture; second, when users know that the property only holds on values where the function is fully specified, they can validate that the given assumptions suffice to restrict the values to the defined part of the function by observing that no potentially spurious counterexample is found.

In the implementation, we extend the test program as follows: The match exceptions are caught with a special constant *catch-match* with a general type $\alpha \Rightarrow \alpha \Rightarrow \alpha$. During the execution, the special constant *catch-match* returns its first argument if no match exception occurs during the evaluation of its first argument; otherwise it returns its second argument. As exception values are not modeled in HOL, we provide no definition for this constant, but map *catch-match v e* to the pattern $(v$ handle Match $\Rightarrow e)$ in the generated source code. Furthermore, to indicate if the counterexample is genuine or potentially spurious due to a match exception, we extend the return type for counterexamples by a boolean flag to $(bool \times result)\ option$. Hence, the scheme for a generated test program for a conjecture C is

(if $C\ x$ then *None* else *Some* (*True*, *reify* x))
handle Match $\Rightarrow$ if *genuineonly* then *None*
else *Some* (*False*, *reify* x)

Quickcheck can search for potential and genuine counterexamples with the execution of the same test program. The test program queries the *genuineonly* flag to enable the two execution modes, allowing or disallowing potentially spurious counterexamples. As a result, Quickcheck compiles the test program for both modes with only one invocation of the code generator. Consequently, searching for potential counterexamples does not require any additional effort during Quickcheck's invocation.

Although successful in practice, both execution modes are somewhat dissatisfactory: The one mode is unsound but complete, because it can return spurious counterexamples, but does not miss any counterexample. The other mode is sound but incomplete, because all counterexamples are genuine, but it could miss some counterexamples.

A sound and complete code generation would of course be desirable. This can be achieved by the following construction, exploiting the specific behavior of

[2]The implementations of the definition mechanisms use different means to make the function's definition total.

the current function package: First of all, we observe that the underlying function definition, which is hidden from the user, is crucial to actually find genuine counterexamples. The current function definition packages totalize partial functions by using some default value, typically the constant *undefined*. The constant *undefined* is a polymorphic constant in HOL without any axioms. Now, to obtain a sound counterexample generator, we must find a counterexample for some possible value for *undefined* in the domain of the type. To obtain a complete counterexample generator, we have to try all possible values for *undefined* to ensure that we checked all possible models.

One way to achieve this behavior is by the following steps: We create copies of the functions, which are extended by a further argument that passes around the chosen value for *undefined*. These functions are total by adding equations for the partial patterns that simply return the value of the new argument as result. For example, to the partial function $hd :: \alpha\ list \Rightarrow \alpha$, we obtain a function $hd' :: \alpha \Rightarrow \alpha\ list \Rightarrow \alpha$ with the two equations, $hd'\ undef\ (x \cdot xs) = x$, $hd'\ undef\ [] = undef$. Then, we rewrite the conjecture under test. For example, the two conjectures,

$$hd\ (append\ xs\ ys) = (\text{if } xs = [] \text{ then } hd\ ys \text{ else } hd\ xs) \text{ and}$$
$$hd\ (map\ f\ xs) = f\ (hd\ xs),$$

are transformed to

$$hd'\ undef\ (append\ xs\ ys) = (\text{if } xs = [] \text{ then } hd'\ undef\ ys \text{ else } hd'\ undef\ xs)$$
$$\text{and } hd'\ undef\ (map\ f\ xs) = f\ (hd'\ undef\ xs).$$

After this processing, Quickcheck would try all possible values for the free variable *undef* as part of its testing. As expected, on the first transformed conjecture, Quickcheck finds no counterexample, as the previous potentially spurious counterexample, $xs = []$ and $ys = []$, was in fact spurious. On the second conjecture, we obtain the genuine counterexample, $xs = [], f = \lambda x.\ a_1, undef = a_2$.

Unfortunately, the transformations described above require large modifications of the specification under test, which is technically difficult to achieve.

An alternative is to generate a fixed value for the constant *undefined* at its first occurrence in the evaluation, and then memorize it. This could be implemented without modifying the complete specification under test, but only modifying the code equation for the constant *undefined*. Nonetheless, the alternative also requires us to modify the code generation's setup globally. Therefore the implementation of Quickcheck does not perform those transformation, but instead seeks for the more minimal invasive method to extend the test programs with exception handling.

An Executable Definite Description Operator for Finite Types

Russell's definite description operator $\iota :: (\alpha \Rightarrow bool) \Rightarrow \alpha$ is axiomatized by

$$\iota\ (\lambda x.\ x = a) = a,$$

which specifies the value of $\iota\ P$ for single-valued predicates $P :: \alpha \Rightarrow bool$, i.e., predicates that only hold for exactly one value of type α. If the predicate P is not single-valued, the value of $\iota\ P$ is unspecified. Like any underspecified function,

the definite description operator poses a challenge for code generation [52, §5], because its axiomatization is not an unconditional equation. Hence, we must derive such an equation from the axiomatization via some refinement. To preserve partial correctness of the code generator's evaluation, the expression $\iota\ P$ is only allowed to be evaluated to some definite value if the predicate P is single-valued. For other predicates, the value of $\iota\ P$ is underspecified, and the implementation must abort or diverge.

We provide an executable implementation for ι if α is a finite type. We evaluate $\iota\ P$ as follows: We enumerate all values of the finite type and check the predicate P. If there is exactly one value for which P holds, we return it; otherwise, we throw an exception. This is implemented by the code equation

$$\iota\ P = (\textit{case filter } P \textit{ univ of } [x] \Rightarrow x \mid \_ \Rightarrow \textit{The'}\ P)$$

where the constant $\textit{The'}\ P$ is defined as $\iota\ P$, but the evaluation of $\textit{The'}\ P$ raises an exception when evaluated.

If Quickcheck is applied to a conjecture with $\iota\ P$, we obtain a genuine counter-example for single-valued predicates P, otherwise if P is not single-valued and the exception is raised, we obtain only a potential one.

3.4 Simple Treatments

Quickcheck incorporates two basic but useful treatments: Conjectures are *massaged* in a preprocessing step and test programs are optimized if the conjectures contain premises with equalities.

3.4.1 Quantifier Massaging

So far in our presentation, we assumed that conjectures were universally quantified statements of the form $A_1 \Longrightarrow ... \Longrightarrow A_n \Longrightarrow C$. We extend the range of formulas by rewriting them to obtain formulas of a suitable form. We apply various rewrite rules

- to move universal quantifiers to the front of the formula,

$$\begin{array}{ll} (\forall x.\ P\ x) \wedge Q \rightsquigarrow \forall x.\ P\ x \wedge Q & P \wedge (\forall x.\ Q\ x) \rightsquigarrow \forall x.\ P \wedge Q\ x \\ (\forall x.\ P\ x) \vee Q \rightsquigarrow \forall x.\ P\ x \vee Q & P \vee (\forall x.\ Q\ x) \rightsquigarrow \forall x.\ P \vee Q\ x \\ (\exists x.\ P\ x) \Longrightarrow Q \rightsquigarrow \forall x.\ P\ x \Longrightarrow Q & P \Longrightarrow (\forall x.\ Q\ x) \rightsquigarrow \forall x.\ P \Longrightarrow Q\ x \end{array}$$

- to move quantifiers through negations,

$$\neg(\exists x.\ P\ x) \rightsquigarrow \forall x. \neg P\ x \quad \neg(\forall x.\ P\ x) \rightsquigarrow \exists x. \neg P\ x$$

- to split conjunctive premises into multiple premises,

$$(P \wedge Q \Longrightarrow R) \rightsquigarrow (P \Longrightarrow Q \Longrightarrow R)$$

- and to check equalities on functions by extensionality.

$$(f :: \alpha \Rightarrow \beta) = g \rightsquigarrow \forall x.\ f\ x = g\ x$$

3.4.2 Equality Optimization

For most premises, Quickcheck generates values and then tests the premise. This behavior is typical for black-box testing tools. However, equational premises, i.e., premises of the form $t = u$, can be optimized. Consider the (invalid) conjecture about the concatenation of two maps, i.e., functions with type $\alpha \Rightarrow \beta\ option$:

$$m_1\ k = Some\ v \Longrightarrow (m_1 \oplus m_2)\ k = Some\ v,$$
$$\text{where } m_1 \oplus m_2 = (\lambda x.\ \text{case } m_2\ x \text{ of } None \Rightarrow m_1\ x \mid Some\ y \Rightarrow Some\ y)$$

Testing the conjecture naively would require us to generate values for all variables freely. However, the premise $m_1\ k = Some\ v$ indicates how to compute the value for v: If $m_1\ k$ is a defined value, i.e., a value with constructor *Some*, then v can be obtained by destructing the value $m_1\ k$. Taking this idea into account, a special treatment of equational premises leads to an improved test program:

$$\begin{array}{l} exhaustive\ (\lambda(m_1, k).\\ \quad \text{case } m_1\ k \text{ of } None \Rightarrow None\\ \qquad\qquad \mid Some\ v \Rightarrow \text{if } (m_1 \oplus m_2)\ k = Some\ v \text{ then } None\\ \qquad\qquad\qquad\qquad \text{else } Some\ (reify\ (m_1, k, v, m_2))) \end{array}$$

In general, we analyze the left-hand and right-hand side of the equation and generate a case-expression to obtain the values of fresh variables on either side of the equation. If variables occur multiple times, the test program is extended with an if-expression with some further equality checks.

3.5 Datatype Refinements

Some conjectures cannot be refuted by Quickcheck because they are not executable due to an inappropriate representation for the values in the conjecture. For example, to the invalid conjecture[3]

$$(x, y) \in (R \cup S)^* \Longrightarrow (x, y) \in R^* \cup S^*,$$

Quickcheck finds the counterexample $x = \mathsf{a}_1$, $y = \mathsf{a}_3$, $R = \{(\mathsf{a}_1, \mathsf{a}_2)\}, S = \{(\mathsf{a}_2, \mathsf{a}_3)\}$ for a type with three elements a_1, a_2, a_3. After instantiating the polymorphic conjecture to relations on natural numbers, it remains invalid: A similar counterexample, $x = 1$, $y = 3$, $R = \{(1,2)\}$, $S = \{(2,3)\}$, would show the invalidity for relations on natural numbers quickly. Unfortunately, the conjecture on natural numbers is not refutable with Quickcheck, because it is not executable employing the default representation for sets. In the target language, sets are represented by lists applying a simple datatype refinement. Hence, only finite and cofinite[4] sets are representable, but the identity relation, which immediately appears in the execution of the reflexive transitive closure, is not finite or cofinite.

Similarly, the conjecture about function composition

$$f \circ g = f \circ h \Longrightarrow g = h$$

[3] R^* denotes the reflexive transitive closure of the relation R.

[4] A cofinite set is a set whose complement is finite.

is refutable with Quickcheck only if the domain type of the functions g and h is finite. As code generation maps functions in HOL directly to functions in the target language Standard ML, it is impossible to check equality of two functions with an infinite domain type.

To make these conjectures executable, we require alternative datatype refinements. In the following subsections, we present two useful refinements for relations and functions and discuss how to automatically obtain suitable representations.

3.5.1 Finitely Representable Relations

To refute the conjecture about the distributivity of union and reflexive transitive closure on relations over natural numbers, we must choose a suitable representation to execute the operations on relations. For example, it must allow us to express reflexive transitive relations over an infinite type. We choose to represent a relation with two finite sets: The first set describes all entries on the diagonal, the second set describes the non-diagonal entries of the relation. This representation for relations is by no means an universal representation, but it is practical for a number of conjectures about relations and relational dataflow dependencies in imperative programs. In this subsection, the example illustrates how Quickcheck can take advantage of data refinements.

The datatype refinement requires *lifting* constants and *transferring* theorems (cf. §2.1.2, [62]). First, we define the type α *rel* for relations on type α, i.e., sets of $\alpha \times \alpha$, and lift the necessary set operations, i.e., membership, union, and the reflexive transitive closure:

typedef α *rel* $=$ *UNIV* :: $((\alpha \times \alpha)$ *set*$)$ *set*
lift-definition *mem* :: $\alpha \times \alpha \Rightarrow \alpha$ *rel* $\Rightarrow$ *bool* **is** *Set.member*
lift-definition *union* :: α *rel* $\Rightarrow \alpha$ *rel* $\Rightarrow \alpha$ *rel* **is** *Set.union*
lift-definition *rtrancl* :: α *rel* $\Rightarrow \alpha$ *rel* **is** *Transitive-Closure.rtrancl*

Now, we choose the representation for relations. The constant *Rel* serves as constructor for our finitely representable relations:

lift-definition *Rel* :: α *set* $\Rightarrow (\alpha \times \alpha)$ *set* $\Rightarrow \alpha$ *rel* **is** $\lambda X\ R.\ (\textit{Id-on}\ X) \cup R$
where *Id-on* $X = \{(x, x) \mid x \in X\}$
code-datatype *Rel*

With this representation, we can obtain values for the type α *rel* that we cannot express on the type $(\alpha \times \alpha)$ *set* during the evaluation. For example, the identity relation is represented with *Rel Univ* $\{\}$ on α *rel*. For the new representation, we now simply derive new code equations for the lifted operations:

$$\textit{mem}\ (x, y)\ (\textit{Rel}\ X\ R) = ((x = y \wedge x \in X) \vee (x, y) \in R)$$
$$\textit{union}\ (\textit{Rel}\ X\ R)\ (\textit{Rel}\ Y\ S) = \textit{Rel}\ (X \cup Y)(R \cup S)$$
$$\textit{rtrancl}\ (\textit{Rel}\ X\ R) = \textit{Rel}\ \textit{UNIV}\ (R^+)$$

Checking relation membership reduces to an equality check and membership tests on the diagonal and non-diagonal part of the relation. The code equations for the union and the reflexive transitive closure reflect the insight that the *Rel* operator and union are distributive, and that the reflexive transitive closure can be expressed as composition of reflexive and transitive closure.

As a last step, we define generators for abstract type *α rel* employing the generator for sets and pairs. After this setup, Quickcheck finds the aforementioned counterexample, when transferring the original conjecture to

$$mem\ (x, y)\ (rtrancl\ (union\ R\ S)) \Longrightarrow mem\ (x, y)\ (union\ (rtrancl\ R)\ (rtrancl\ S))$$

3.5.2 Finite Functions

For an executable equality on functions, we must refine function values to values with a data representation, for which we can implement function equality. In general, equality of two HOL functions is undecidable, but if we restrict ourselves to refute the conjecture on a reasonable subset of the function space, the counterexample on this subset also serves as counterexample on the full function space. One such subset of the function space is Lochbihler's formalization of finite functions [79]. By restricting functions to be constant except for finitely many points, we can represent them by association lists and an explicit default value. On these finite functions, we can check the equality of two finite functions with a simple implementation that compares the two association lists and default values.

The implementation is almost straightforward. It must take one specialty into account: The behavior depends on the universe of the function's domain being finite or infinite, which can be nicely expressed with type classes. As the initial formalization is Lochbihler's contribution, we do not go into details here, but refer the reader to [79].

The initial formalization required the user to modify an existing specification extensively, or decide at the beginning of his formalization to use finite functions. Our contribution simplified the formalization employing the new lifting and transfer mechanism [62], following similar lines to the formalization of finitely representable relations. This engineering effort integrates finite functions with the system smoothly, and now enables us to transfer conjectures on functions to conjectures on finite functions before trying to refute them.

3.5.3 Automatic Data Refinements

For the two examples, we came up with the representations by human insight, and informally argued why they are suitable representations. However in general, a suitable representation varies from one conjecture to another. For most examples, representing sets by finite lists is preferred to represent them as membership functions. However, the set representation with finite lists permits to represent the set $\{xs.\ length\ xs = 2\}$, but this can be easily expressed as a membership function directly. Even worse, although sets (*α set*) and predicates ($\alpha \Rightarrow bool$) are isomorphic, choosing one or the other influences whether the conjecture is within the executable fragment. Hence, users must be aware of the code representation, which

is equipped with the various types.

The presented refinements provide only a partial solution: The chosen relation representation cannot handle relations like $\{(x, y).\ x = c\}$ and $\{(x, y).\ y = c\}$. At the moment, the user still has to come up with a representation for code generation, for the specific class of functions, sets or relations he is interested in.

In larger specifications, choosing one representation uniformly for all set and function types is often impossible, but one requires different representation for the various types occurring in the specification. At the moment, this is only possible by providing copies of the types with different representations, and manually instructing the system with a mapping of types to its representations.

A next step is to automatically discover a suitable representation for code generation. In principle, a tool can also automatically discover that all occurring functions can be finitely represented and obtain a suitable representation for code generation. To do so, the tool must analyze the definitions with an appropriate analysis, similar to the *monotonicity analysis* [20]. The inferred type annotations can drive which representation for function must be chosen. Employing these analysis abolishes the need to manually set up code generation and enables us to handle specifications that are not based on the existing libraries. Future work must clarify how to refine datatypes supported by such an analysis.

3.6 Related Work

There is huge amount of related work in the broad field of software testing [87]. Here we will focus the presentation of related work in the setting of functional programming languages and interactive theorem provers. Haskell's QuickCheck [33] was the first to explore specification-based testing in functional programming languages. Its success story has led to many descendants in interactive theorem provers and other (functional) programming languages. Besides Isabelle, PVS [95], Agda [43], ACL2 [44] and ACL2 Sedan [28] include a random testing tool like the original QuickCheck. Although Coq, HOL4, HOL-Light and Mizar have a considerable user base, counterexample generators are conspicuously missing in these systems. The integrated development environment Focal [7] also includes a Quickcheck-like random testing tool [25].

One of the main concerns of a testing tool like Quickcheck is to automate the construction of test data generators. Most Quickcheck-like systems are based on a specification language with inductive datatypes, some also support further constructions, such as records or subtypes. The automation of the test-data-generator construction usually focuses on inductive datatypes. As theorem provers are equipped with a well-developed meta-language, i.e., functions to construct and inspect types and terms, automated construction is technically simpler than in a programming language: Besides Isabelle's Quickcheck, PVS and ACL2 Sedan also provide an automatic construction for datatypes. The construction essentially follows along similar lines, with minor variations due to the different type definition mechanisms in the theorem provers.

Constructing test data generators in a programming language is technically more involved, and requires an introspection mechanism. This can be provided

by generic programming frameworks (cf. [58] for an overview), such as Template Haskell [106] and Generic Haskell [57]. Besides Haskell, other functional programming languages also provide Quickcheck-like testing tools, e.g., Standard ML [76], OCaml [108], Erlang [6] and Scheme. The tools are conceptionally very similar to the Haskell version, but the implementations differ as the underlying programming languages lack the concept of type classes. Due to this deficiency, Quickcheck for Standard ML requires the user to explicitly write out test data generators, essentially imitating the type class mechanism by creating dictionaries manually. To overcome this deficiency, another Quickcheck for ML [85] deploys a special reflection mechanism for types in a special ML dialect. The work [72] describes some infrastructure for automating the construction of test data generators in Standard ML. Similar work [24] is also done for the language OCaml. This work also extensively focuses on a uniform distribution of random values.

Our exhaustive testing is inspired by Haskell's SmallCheck [105] but is targeting ML with its strict evaluation. The implementation of Haskell's SmallCheck takes advantage of its laziness, simplifying the definition of generators, while Isabelle's tool takes the strictness of ML into account and uses continuations.

The Feat package [42] for Haskell allows us to enumerate values of algebraic types and separates the enumeration of values from its strategy. Hence, we can enumerate exhaustively, randomly or with various hybrid strategies by implementing the strategies employing only a single family of generators. Its distribution for the exhaustive enumeration differs from the one in SmallCheck, which was critically important while testing with large datatypes.

Gast [75] is an exhaustive testing tool for the programming language Clean. As Clean supports generic programming, Gast can automatically construct the test data generators for all datatypes when being invoked.

Beyond functional languages, Jhala and Majumdar [67, sect. 2] give an overview of bounded software model checking with concrete values for imperative programming languages, which is closely related to exhaustive testing.

Quickcheck has been successfully applied in a number of industrial applications [6, 34, 35, 39, 63, 64] and teaching [97].

In this work, we focused on the implementation aspects to check polymorphic conjectures to obtain test programs that check properties with finite domains very efficiently. The theoretical work of testing polymorphic properties [17] exploits parametricity of the functions under test. It presents an analysis to compute the monomorphic type that a polymorphic property should be tested on. Their analysis gives an upper bound for cardinality of the monomorphic type, i.e., the computed monomorphic type is sufficiently large to ensure that there exists a counterexample to the polymorphic property if and only if there exists a counterexample to the monomorphic instance.

Although the potential of this technique sounds promising, the analysis is only applicable for checking (dis)equality of truly polymorphic functions, i.e., polymorphic functions that do not use equality in their definitions. Hence, this is only applicable for very special conjectures and has not been implemented yet.

Isabelle's counterexample generator Nitpick employs a *monotonicity analysis* [20] to prune the search space for polymorphic conjectures with multiple type

variables. If a formula with n atomic types is monotonic, it suffices to check all models in which all atomic types have cardinality k, instead of k^n combinations of cardinalities. For example, to check associativity of the concatenation of maps,

$$(m_1 :: \alpha \Rightarrow \beta\ option) \oplus (m_2 \oplus m_3) = (m_1 \oplus m_2) \oplus m_3,$$

the counterexample generator must choose types with a fixed cardinality for the type variables α and β. In general, one would have to check all combinations of cardinalities for the types α and β. As the proposition is monotonic, one can limit checking to the cases where α and β have the same cardinalities. Nitpick employs an analysis to determine if the formula is monotonic. In contrast, Quickcheck chooses α and β with same cardinalities without a further analysis by default. Due to the lack of the monotonicity analysis, it is incomplete in theory. However in practice, most conjectures are monotonic, as already pointed out by Blanchette and Krauss [20]. We never observed missing a counterexample due to this incompleteness—besides on especially constructed examples to illustrate the weakness. In the future, one could integrate Nitpick's monotonicity analysis into Quickcheck to regain completeness in this respect, but this would put higher load on its current architecture.

Chapter 4

Conditional Conjectures

Counterexample generators that test with random values or exhaustively with small values, perform well on conjectures without premises. For example, for the invalid conjecture about lists

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

the counterexample generators provide the counterexample $\mathit{xs} = [\mathsf{a}_1]$ and $\mathit{ys} = [\mathsf{a}_2]$ (for $\mathsf{a}_1 \neq \mathsf{a}_2$) instantaneously. For conjectures of this kind, random and exhaustive testing are perfectly suited.

The main weakness of both random and exhaustive testing, already mentioned in the original QuickCheck for Haskell paper [33], is that they do not cope well with hard-to-satisfy premises. For example, when testing our previous conjecture about *insort* (cf. § 3.1),

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

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

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

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

The predicate *is-rbt* captures the invariant of red-black trees on the type of binary search trees (α,β) *tree*.[1] Again, binary trees generated naively rarely satisfy the premise, and we prefer to generate only trees satisfying the invariant.

In this chapter, we present two techniques to handle conditional conjectures: custom and smart generators.

[1] The type (α,β) *tree* should not be confused with type (α,β) *rbt* from § 3.2.5.

4.1 Custom Generators

The simplest solution to test conditional conjectures effectively is to let the user provide a custom generator. Assuming the user provides a generator for some type restricted by a predicate (cf. §3.2.5) that matches the condition, Quickcheck automatically *lifts* the conjecture to the restricted type. For example, the conjecture about *delete* is automatically lifted to the type (α,β) *rbt*, where $Rep_{rbt}\ t'$ maps a red-black tree t' of type (α,β) *rbt* to its representative binary tree on type (α,β) *tree*:

$$\textit{is-rbt}\ (Rep_{rbt}\ t') \implies \textit{is-rbt}\ (\textit{delete}\ k\ (Rep_{rbt}\ t'))$$

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

4.2 Smart Generators

Custom generators suffice to test conditional conjectures effectively. However to employ them, users have to spend some effort to set them up. In the case of the example with red-black trees, this might be an acceptable effort for users, as they spend considerable time proving operations on red-black trees and invent various conjectures for the operations with the premise *is-rbt t*. As users also define operations to construct red-black trees, setting up a custom generator fits naturally in the development process in this scenario.

In other scenarios, e.g., when stating the conjecture about sortedness, providing custom generators is more difficult. The function *insort* could serve as function for constructing sorted lists, but this requires to have already proved that *insort* preserves sortedness. When proving another conjecture with a premise *sorted xs*, users do not want to pursue a number of steps just to obtain a more effective test data generator.[2] Instead, users simply state the conjecture and hope that the counterexample generator does its best, as custom generators are too laborious for conjectures with premises that are unlikely to reoccur repeatedly in many other conjectures.

A more sophisticated solution to test conditional conjectures effectively, is smart test data generators that take the condition's definition into account. These test data generators construct values in a bottom-up fashion, simultaneously testing the condition and generating appropriate values. In contrast to custom generators, they do not require any user setup. We present two examples for these smart generators.

For further illustration, we focus on the valid conjecture about distinct lists:

$$\textit{distinct}\ xs \implies \textit{distinct}\ (\textit{tl}\ xs).$$

[2]To obtain the generator, users would have to define the *insort* function, prove that it preserves sortedness, define a new type of sorted lists, and set up a custom generator.

The counterexample generator that tests exhaustively employs the following *test program* (cf. §3.1 and §3.2.3) to check the validity of the conjecture:

$$\begin{aligned}&\mathit{exhaustive}_{\mathit{nat\ list}}\ (\lambda \mathit{xs}.\ \text{if}\ \neg\ \mathit{distinct}\ \mathit{xs}\ \text{then}\ \mathit{None}\\ &\qquad\qquad\qquad\qquad\text{else if}\ \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})\ \text{then}\ \mathit{None}\ \text{else}\ \mathit{Some}\ \mathit{xs})\ i\end{aligned}$$

This test program implements a simple *generate-and-test loop*. It uses the function $\mathit{exhaustive}_{\mathit{nat\ list}}$ to generate all lists of natural numbers up to a given bound and iteratively test the property at hand.

The smart generators interleave generation and checking in a way that avoids generating lists that are not distinct. From the definition of the *distinct* predicate,

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

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 prepending an element which is not in the list already. This insight is reflected in the smart test data generator *exhaustive-distinct* for lists of type α:

$$\begin{aligned}&\mathit{exhaustive\text{-}distinct}_{\alpha\ \mathit{list}}\ c\ i = (\text{if}\ i = 0\ \text{then}\ \mathit{None}\ \text{else}\ (c\ \mathit{Nil}\ \sqcup\\ &\quad(\mathit{exhaustive\text{-}distinct}_{\alpha\ \mathit{list}}\ (\lambda \mathit{xs}.\ \mathit{exhaustive}_{\alpha}\\ &\qquad(\lambda x.\ \text{if}\ x \notin \mathit{set}\ \mathit{xs}\ \text{then}\ c\ (x \cdot \mathit{xs})\ \text{else}\ \mathit{None})\ (i-1))\ (i-1))))\end{aligned}$$

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

$$\mathit{exhaustive\text{-}distinct}_{\mathit{nat\ list}}\ (\lambda \mathit{xs}.\ \text{if}\ \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})\ \text{then}\ \mathit{None}\ \text{else}\ \mathit{Some}\ \mathit{xs})\ i$$

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

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

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

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

The counterexample generators in the previous chapter test the conjecture with concrete values. Testing with concrete values has the clear advantage of being natively supported by the targeted functional programming language and hence is executed very fast, but it has the drawback that a large set of test inputs may

exhibit indistinguishable executions. For example, 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 they exhibit the same execution in the test program.

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

For conjectures without premises, we enumerate all possible concrete values. This is fairly effective, because usually there are only very few symmetric executions in that case. When premises occur in conjectures, the smart test data generators only produce values fulfilling the condition, and then test the conclusion. We generate values fulfilling the premises with a program that queries the premises' predicate. This generator enumerates values by considering all derivations for the predicate, similar to a query in Prolog, but in contrast to Prolog, the generator returns only ground solutions but does not use schematic variables. The query is integrated in a lightweight fashion into the test program by a *compilation*. Throughout this section, we use the term *compilation* to designate our translation of Horn clauses to programs written in Isabelle's functional programming language.

More specifically, Quickcheck synthesizes these generators by reformulating the definitions as a set of Horn clauses and computing its dataflow dependencies. The smart test data generator for a given predicate is produced by a compiler that analyzes the predicate's definition and synthesizes a purely functional program that serves as generator. For this purpose, the compiler reformulates the predicate's definition as logic programs by translating formulas in predicate logic with quantifiers and recursive functions to Horn clauses. The compiler analyzes the Horn clauses with a dataflow 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.

Using these generators reduces the number of tests, and as our evaluation (§6.2) 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 with distinct lists, 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.

Smart generators also address an issue with conditional conjectures that are expressed as a conjunction of conditions, such as

$$\begin{aligned} &\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys} \wedge \mathit{zip}\ \mathit{xs}\ \mathit{ys} = \mathit{zs} \Longrightarrow \mathit{map}\ \mathit{fst}\ \mathit{zs} = \mathit{xs} \wedge \mathit{map}\ \mathit{snd}\ \mathit{zs} = \mathit{ys} \\ &\mathit{sorted}\ \mathit{xs} \wedge i \leq j \wedge j < \mathit{length}\ \mathit{xs} \Longrightarrow \mathit{nth}\ \mathit{xs}\ i \leq \mathit{nth}\ \mathit{xs}\ j \end{aligned}$$

At first sight, it is not clear in which order the conjuncts should be checked. Smart generators attempt to reorder conjecture's premises based on the findings of its dataflow analysis.

4.2.1 Architecture

The counterexample generator performs these steps: As the original specification can be defined using various definitional mechanisms, the specification is prepro-

cessed by a few simple syntactic transformations (§4.2.2) to Horn clauses. The core component, which was previously described in [16], consists of a static dataflow analysis, i.e., the mode analysis (§4.2.4), and the code generator (§4.2.5). This core component only works on a syntactic subset of the Isabelle language, namely Horn clauses of the following form:

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

In a premise $Q_i\ \bar{u}_i$, Q_i must be a predicate defined by Horn clauses and the terms $\bar{u}_i$ 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 treats them as side conditions. 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 using nondeterministic executions and type-based generators.

4.2.2 Processing of Definitions to Horn Clauses

A definition in predicate logic is transformed to a system of Horn clauses, based on the observation that a definition of the form $P\ \bar{x} = \exists \bar{y}.\ Q_1\ u_1 \wedge \cdots \wedge Q_n\ u_n$ can be soundly under-approximated by a Horn clause $Q_1\ u_1 \Longrightarrow \cdots \Longrightarrow Q_n\ u_n \Longrightarrow P\ \bar{x}$. 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,

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

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

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

4.2.3 Function Flattening

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.

Basic terms

$\mathsf{flatten}\ c = \{(c, \emptyset)\}$ for any constant c

$\mathsf{flatten}\ x = \{(x, \emptyset)\}$ for any free variable x

$\mathsf{flatten}\ (\lambda x.\ t) = \{(\lambda x.\ t, \emptyset)\}$

Special function application

$$\mathsf{flatten}\ (\text{if } b \text{ then } x \text{ else } y) = \\ \big\{(r, \{b\} \cup \mathcal{P}).\ (r, \mathcal{P}) \in \mathsf{flatten}\ x\big\} \cup \big\{(r, \{\neg b\} \cup \mathcal{P}).\ (r, \mathcal{P}) \in \mathsf{flatten}\ y\big\}$$

$$\mathsf{flatten}\ (\text{case } t \text{ of } C_1\ \overline{y_1} \Rightarrow rhs_1 |\ \ldots |\ C_n\ \overline{y_n} \Rightarrow rhs_n) = \\ \textstyle\bigcup_{i=1}^{n} \big\{(r', \{r = C_i\ \overline{y_i}\} \cup \mathcal{P} \cup \mathcal{Q}).\ (r', \mathcal{P}) \in \mathsf{flatten}\ rhs_i,\ (r, \mathcal{Q}) \in \mathsf{flatten}\ t\big\}$$

$$\mathsf{flatten}\ (\text{let } x = t \text{ in } u\ x) = \\ \big\{(r', \mathcal{P} \cup \mathcal{Q}).\ (r', \mathcal{Q}) \in \mathsf{flatten}\ (u\ r), (r, \mathcal{P}) \in \mathsf{flatten}\ t\big\}$$

General function application

If for the function f a predicate f_P is defined:

$$\mathsf{flatten}\ (f\ u_1 \ldots u_n) = \\ \big\{(r, \{f_P\ res_1 \ldots res_n\ r\} \cup \textstyle\bigcup_{i=1}^{n} \mathcal{P}_i).\ (res_1, \mathcal{P}_1) \in \mathsf{flatten}\ u_1, \ldots \\ \ldots, (res_n, \mathcal{P}_n) \in \mathsf{flatten}\ u_n\big\}$$

otherwise:

$$\mathsf{flatten}\ (f\ u_1 \ldots u_n) = \\ \big\{(f\ res_1 \ldots res_n, \textstyle\bigcup_{i=1}^{n} \mathcal{P}_i).\ (res_1, \mathcal{P}_1) \in \mathsf{flatten}\ u_1, \ldots, (res_n, \mathcal{P}_n) \in \mathsf{flatten}\ u_n\big\}$$

Figure 4.1: Definition of function flattening

Example 2. The *length* of a list is defined by $length\ [] = 0$, and $length\ (x \cdot xs) = Suc\ (length\ xs)$. We derive a corresponding relation $length_P$ with two Horn clauses:

$$length_P\ []\ 0$$
$$length_P\ xs\ n \Longrightarrow length_P\ (x \cdot xs)\ (Suc\ n)$$

The premise $length\ xs = length\ ys$ is then transformed into

$$length_P\ xs\ n \wedge length_P\ ys\ n$$

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 dataflow.

Commonly, recursive functions in Isabelle are specified by equations, $f\ \bar{t} = rhs$ where $\bar{t}$ are constructor patterns. For every function f, we define a predicate f_P such that $f_P\ \bar{t}\ res$ holds if $f\ \bar{t} = res$. Right-hand sides of equations are flattened into a set of pairs with a result expression and set of premises by the function $\mathsf{flatten} :: term \Rightarrow (term \times premise\ set)\ set$ (see figure 4.1). For every equation $f\ \bar{t} = rhs$ and for every pair of result expression and premises $(r, \mathcal{P}) \in \mathsf{flatten}\ rhs$, we derive a Horn clause $P_1 \Longrightarrow P_2 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow f_P\ \bar{t}\ r$ with $\{P_1, \ldots, P_n\} \in \mathcal{P}$.

This well-known technique of flattening has been described previously in the literature, e.g., by Naish [89] and Rouveirol [104]. Our implementation also supports flattening of higher-order functions, which allows inversion of higher-order functions if the function argument is invertible.

4.2.4 Mode Analysis

For the execution of a predicate P, the predicate's arguments are classified as *input* or *output*, made explicit by means of *modes*.[3] Modes can be inferred using a static analysis on the Horn clauses. Our mode analysis is based on Mellish [84]. There are more sophisticated mode analysis approaches, e.g., by using abstract domains [107] or by translating to a boolean constraint system [94]. For our purpose, it suffices to 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.

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 $Q_1\ \bar{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \bar{u}_n \Longrightarrow P\ \bar{t}$ 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 $Q_1\ \bar{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \bar{u}_n \Longrightarrow P\ \bar{t}$, 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)** $v_j = v_{j-1} \cup FV(\bar{u}_j)$ obeys the conditions **(3)** $FV(\bar{u}_j\langle M_j\rangle) \subseteq v_{j-1}$ 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.

Example 3. Consider the predicate $append_P$, which is obtained by flattening the *append* function, with its two Horn clauses

$$append_P\ []\ ys\ ys$$
$$append_P\ xs\ ys\ zs \Longrightarrow append_P\ (x \cdot xs)\ ys\ (x \cdot zs)$$

The modes $i \Rightarrow i \Rightarrow o \Rightarrow bool$ and $o \Rightarrow o \Rightarrow i \Rightarrow bool$ for the predicate $append_P$ admit a consistent mode assignment for the two Horn clauses. For the first clause and the two modes, the four conditions are respected as $v_0 = \{ys\} = v_n$. For the second clause and $M = i \Rightarrow i \Rightarrow o \Rightarrow bool$, we choose $M_1 = i \Rightarrow i \Rightarrow o \Rightarrow bool$

[3] Modes were already introduced in §2.2.3. For a self-contained presentation, we recap parts from §2.2.3, but present the mode analysis in more detail.

and meet the four conditions, as $v_0 = \{x, xs, ys\}$, $FV(\bar{u}_1\langle M_1\rangle) = \{xs, ys\} \subseteq v_0$ and $v_1 = \{x, xs, ys, zs\} \supseteq FV(\bar{t})$ holds. Similarly for $M = o \Rightarrow o \Rightarrow i \Rightarrow bool$, we choose M_1 to be $o \Rightarrow o \Rightarrow i \Rightarrow bool$ and meet the four conditions, as $v_0 = \{x, zs\}$, $FV(\bar{u}_1\langle M_1\rangle) = \{zs\} \subseteq v_0$ and $v_1 = \{x, xs, ys, zs\} \supseteq FV(\bar{t})$ holds.

Generator Mode Analysis. To generate values that satisfy a predicate, we extend the mode analysis: If the mode analysis cannot detect a consistent mode assignment because the values of some variables are not *constrained* after the evaluation of the premises, we allow the use of *generators*. The values for these variables are constructed by an *unconstrained* enumeration. The overall resulting generator enumerates values either driven by the *computation* of a predicate or by *generation* based on its type. More specifically, the mode with all arguments classified as output describes a dataflow in the Horn clauses that generates values satisfying the predicate. For example, the generator mode analysis for the predicate *distinct* with mode $o \Rightarrow bool$ infers how to generate distinct lists.

Example 4. Given the predicate $op \notin$ in $x \notin set\ xs$ has one mode $i \Rightarrow i \Rightarrow bool$ only, the classical mode analysis fails to find a consistent mode assignment for the predicate *distinct* with mode $o \Rightarrow bool$ due to its second Horn clause

$$distinct\ xs \Longrightarrow x \notin set\ xs \Longrightarrow distinct\ (x \cdot xs).$$

To generate values for x and xs fulfilling $distinct\ (x \cdot xs)$, we combine computation and generation of values as follows: The values for variable xs are built using *distinct* with $M_1 = o \Rightarrow bool$; values for x are built by a generator and undergo the check of $x \notin set\ xs$ with mode $M_2 = i \Rightarrow i \Rightarrow bool$ for predicate $op \notin$.

This extension gives rise to a number of possible modes, because we omit the conditions **(3)** and **(4)** for the mode analysis. Instead, we use a heuristic to find an appropriate 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 $|FV(\bar{u}_j\langle M_j\rangle) - v_{j-1}|$ be minimal;
2. prefer functional predicates with their functional mode;
3. prefer 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 4 (continued). A priori, the mode analysis has two alternatives for the mode M_1 to the premise *distinct xs*: generating values for xs and then testing *distinct xs* with mode $i \Rightarrow bool$, or only generating values for xs using *distinct* 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 $i \Rightarrow bool$ thanks to criterion 1: for $v_0 = \{\}$, $\overline{u}_1 = xs$ and $M_1 = i \Rightarrow bool$, $FV(\overline{u}_1\langle M_1\rangle) - v_0 = \{xs\}$; 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 $M_1 = o \Rightarrow bool$.

Due to the second criterion, the mode analysis prefers functional predicates with their functional mode. This criterion attempts to reduce the number of possible intermediate solutions: The evaluation of predicates with functional modes cannot fail, returns exactly one value and can be implemented more efficiently.

Example 5. Consider a clause $R\ x\ y \Longrightarrow F\ x\ y \Longrightarrow P\ x\ y$ where 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. In contrast, the opposite order, i.e., computing y with $R\ x\ y$ and then checking $F\ x\ y$ could lead to generating many values for y, where at most one is fulfilled by $F\ x\ y$.

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 6. 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 $Q\ x$ to $R\ x\ y$, 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 commonly leads to a bottom-up generation of values.

Example 7. In a clause $P\ xs \Longrightarrow C\ xs \Longrightarrow P\ (x \cdot xs)$, where P and C admit both modes $i \Rightarrow bool$ and $o \Rightarrow bool$, the mode analysis could choose one of the two alternatives: The mode assignment $o \Rightarrow bool$ for $P\ xs$ and $i \Rightarrow bool$ for $C\ xs$ yields a dataflow that generates values fulfilling P and checks the condition C. The alternative assignment $i \Rightarrow bool$ for $P\ xs$ and $o \Rightarrow bool$ for $C\ xs$ leads to generate values for xs from the premise $C\ xs$ and checks $P\ xs$. The mode analysis prefers the first mode assignment to the second due to criterion 5. Here, generators should exploit the recursion in the predicate's definition: Given a recursive definition of a predicate, the corresponding generator attempts to generate values by recursively generating smaller values, for which the predicate holds. In our example, values fulfilling $P\ (x \cdot xs)$ are generated by prepending values for x to all values generated for $P\ xs$ and fulfilling $C\ xs$.

This generator mode analysis results in mode-annotated Horn clauses, where annotations mark which values are enumerated by exhaustive generators. In summary, it not only *discovers* an existing dataflow, but helps to *create* a dataflow by filling the gaps with the type-based test data generators.

4.2.5 Generator Compilation

In this section, we discuss the translation of the compiler from mode-annotated Horn clauses to functional programs. 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 dataflow given by the mode-annotated Horn clauses allows the compiler to generate a functional program with nondeterministic computations. The function P^M for the mode M with all arguments as output serves as test data generator for predicate P.

First, we discuss the execution mechanism based on nondeterministic computations. Then, we sketch the compilation scheme and its applications to the introductory examples.

Monads for Nondeterministic Computations

We use continuations to enumerate the (potentially infinite) set of values fulfilling the involved predicates.

Employing the *plus monad* operations describing nondeterministic 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.

The *continuation plus monad* supports four operations: *return*, *bind*, *mzero* and *mplus*. Their definitions and properties have been provided in §3.2.1.

If we employ these operations in Standard ML to compute solutions to query a predicate, the execution reflects 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 might cause infinite computations escaped from the control of the user. To avoid that the execution is stuck in such a computation, we employ a plus monad that limits the computation by a depth limit. As the evaluation answers negative queries with a negation-by-failure semantics, it must take special care of negation when evaluating predicates with a depth-limited computation. We implement different behaviors for queries in different contexts: for positive contexts, the implementation computes an underapproximation; for negative contexts, an overapproximation.

More specifically, the plain continuations without the depth limit have type $(\alpha \Rightarrow \mathit{result\ option}) \Rightarrow \mathit{result\ option}$. For a depth-limited computation in positive contexts, we provide an extended continuation plus monad with type $(\alpha \Rightarrow \mathit{result\ option}) \Rightarrow \mathit{int} \Rightarrow \mathit{result\ option}$. The bind^+ operation checks the depth limit and if reached, it behaves like a failing computation; otherwise it passes a decreased depth limit to its argument. This yields a sound underapproximation of the query. The other three plus monad operations pass around the additional argument and behave like the usual continuation monad operations:

$$\mathit{mzero}^+ = (\lambda c\ i.\ \mathit{None})$$
$$\mathit{return}^+\ x = (\lambda c\ i.\ c\ x)$$
$$\mathit{bind}^+\ m\ f = (\lambda c\ i.\ \text{if } i = 0 \text{ then } \mathit{None} \text{ else } (m\ (\lambda a.\ (f\ a)\ c\ i)\ (i - 1)))$$

$$mplus^+\ c_1\ c_2 = (\lambda c\ i.\ \text{case}\ c_1\ c\ i\ \text{of}\ None \Rightarrow c_2\ c\ i \mid Some\ x \Rightarrow Some\ x)$$

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, and continue the computation with this value.

$$\textbf{datatype}\ \alpha\ unknown = Unknown \mid Known\ \alpha$$

A negative computation can yield *None* or *Some x*, but it can also yield the unknown value. We encode the possible results in the *ext-option* type:

$$\textbf{datatype}\ \alpha\ ext\text{-}option = Unknown\text{-}value \mid Some\ \alpha \mid None$$

For the negative depth-limited computation, the type of the continuation plus monad is $(\alpha\ unknown \Rightarrow result\ ext\text{-}option) \Rightarrow int \Rightarrow result\ ext\text{-}option$.

The monad operations $mzero^-$, $return^-$ and $mplus^-$ do not consider the depth limit and are similar to their counterparts for positive contexts:

$$\begin{aligned}
&mzero^- = (\lambda c\ i.\ None)\\
&return^-\ v = (\lambda c\ i.\ c\ (Known\ v))\\
&mplus^-\ c_1\ c_2 = (\lambda c\ i.\ \text{case}\ c_1\ c\ i\ \text{of}\\
&\quad None \Rightarrow c_2\ c\ i \mid Some\ x \Rightarrow Some\ x \mid Unknown\text{-}value \Rightarrow Unknown\text{-}value)
\end{aligned}$$

The $bind^-$ operation implements the conjunction of two computations and handles the conjunction of a computation reaching the depth limit and failing computation special: If one computation reaches the depth limit and the continuation fails, then the overall computation fails; in other words *failure absorbs the unknown value* (which is consistent with a Kleene three-valued logic interpretation).

$$\begin{aligned}
bind^-\ m\ f = (\lambda c\ i.\ &\text{if}\ i = 0\ \text{then}\ c\ Unknown\\
&\text{else}\ m\ (\lambda a.\ \text{case}\ a\ \text{of}\ Unknown \Rightarrow c\ Unknown\\
&\qquad \mid Known\ a' \Rightarrow f\ a'\ c\ i)\ (i-1))
\end{aligned}$$

Because negative and positive occurrences of predicates are intermixed, actual enumerations combine the positive and negative monads: The bridge between them is performed by *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 *true*; 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 overall computation cannot proceed further.

The compilation scheme in the next subsection builds on the interface of plus monad structure and hence is employed for all three monads uniformly. For the rest of the presentation, we only employ the depth-limited monad for positive contexts and write $mplus^+$ and $bind^+$ infix as $\sqcup$ and $\gg\!=$.

Compilation of Mode-Annotated Clauses

The functional equation for P^M is the union of the output values generated by the characterizing clauses. Employing the dataflow 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 Standard ML, because they only employ the monad operations and pattern matching. A formal description of the compilation scheme is provided in [16]. For the extended compilation with generators, we only had to extend it such that the exhaustive type-based generators (cf. §3.2.3) are included if they are necessary. To give the reader some intuition of the compilation, we provide two examples of the compiled programs and show another example in §6.3.2.

Example 8. 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 $x \notin set\ xs$ with mode $i \Rightarrow i \Rightarrow bool$. This is then compiled to a test data generator $distinct^o$ for lists of type α:

$$\begin{aligned} distinct^o = {} & return^+\ [] \sqcup \\ & (distinct^o \gg\!= (\lambda xs.\ exhaustive_\alpha \gg\!= (\lambda x.\ \text{if } x \notin set\ xs \text{ then } return^+\ (x \cdot xs) \\ & \qquad\qquad \text{else } mzero^+))) \end{aligned}$$

Instantiating α to the natural numbers and unfolding the plus monad operators in the definition of $distinct^o$, we obtain the test data generator *exhaustive-distinct* from section 4.2.

Example 9. For the premise $length\ xs = length\ ys \wedge zip\ xs\ ys = zs$, we obtain the following mode-annotated clause:

- $length_P\ xs\ n$ with mode $o \Rightarrow o \Rightarrow bool$,
- $length_P\ ys\ n$ with mode $o \Rightarrow i \Rightarrow bool$,
- $zip_P\ xs\ ys\ zs$ 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 *zip xs ys*. The generator for this premise then is

$$\begin{aligned} & {length_P}^{oo} \gg\!= (\lambda(xs, n).\ {length_P}^{oi}\ n \\ & \quad \gg\!= (\lambda ys.\ return^+\ (zip\ xs\ ys) \gg\!= (\lambda zs.\ return^+\ (xs, ys, zs)))) \end{aligned}$$

Unfolding the definitions of the plus monad operators and reducing the syntactic clutter, we obtain

$$\begin{aligned} & \lambda\ c\ d.\ \text{if } d = 0 \text{ then } None \\ & \qquad \text{else } {length_P}^{oo}\ (d-1)\ (\lambda(xs, n).\ {length_P}^{oi}\ n\ (d-1)\ (\lambda ys.\ c\ (xs, ys, zip\ xs\ ys)) \end{aligned}$$

The arguments *c* and *d* make the continuation and the depth limit explicit, which are implicitly passed around by the monad operations.

4.2.6 Extensions

The mode analysis and the compilation is extended in two simple ways:

Simple support for arithmetic. For addition and subtraction on natural numbers and integers, we have special support, so that the mode analysis utilizes that these operations are invertible. For example, for a premise $a + b = c$, the analysis is aware that one can compute a given values for b and c by $a = c - b$. For subtraction on natural numbers, we must take special care of the case where the result value is zero: If $a - c = 0$ holds, we only know that the value a is in the interval of $0 < a \leq c$. For such intervals on natural numbers, we can enumerate all numbers within their bounds. This simple support for arithmetic operations suffices to obtain smart test data generators for various conjectures, e.g., conjectures about operations on indices in lists, because the conjectures contain only very simple arithmetical constraints. For intricate arithmetic constraints one would need to employ a constraint solver and dedicated decision procedures.

Specialization. Often, some crucial modes cannot be inferred because the mode analysis is too imprecise. For example, the function *lookup* returns the value of a key k from an association list.

$$\begin{array}{l} \mathit{lookup}\ []\ k = \mathit{None} \\ \mathit{lookup}\ ((k, v) \cdot \mathit{kvs})\ k' = \text{if } (k = k') \text{ then } \mathit{Some}\ v \text{ else } \mathit{lookup}\ \mathit{kvs}\ k' \end{array}$$

For a predicate with the premise $\mathit{lookup}\ \mathit{kvs}\ k = \mathit{Some}\ v'$, the evaluation must enumerate all possible values for k and v given kvs. By flattening the function (§4.2.3), we obtain the premise $\mathit{lookup}_P\ \mathit{kvs}\ k\ (\mathit{Some}\ v')$, where the predicate lookup_P has the Horn clauses

$$\begin{array}{l} \mathit{lookup}_P\ []\ k\ \mathit{None} \\ k = k' \Longrightarrow \mathit{lookup}_P\ ((k, v) \cdot \mathit{kvs})\ k'\ (\mathit{Some}\ v) \\ \mathit{lookup}_P\ \mathit{kvs}\ k'\ v' \Longrightarrow k \neq k' \Longrightarrow \mathit{lookup}_P\ ((k, v) \cdot \mathit{kvs})\ k'\ v' \end{array}$$

For enumerating all possible values for k and v given kvs, lookup_P must allow the mode $i \Rightarrow o \Rightarrow o \Rightarrow \mathit{bool}$. The first Horn clause $\mathit{lookup}_P\ []\ k\ \mathit{None}$ disallows this mode. However, for all results (k, v) due to this Horn clause, v is *None*. Hence, they do not match the premise $\mathit{lookup}_P\ \mathit{kvs}\ k\ (\mathit{Some}\ v)$—they lead only to failing computations for this premise. Here, the mode analysis is too imprecise to detect that if the third argument should only match the pattern $\mathit{Some}\ v'$, the first Horn clause of lookup_P provides no results.

If we define a specialized predicate $\mathit{lookup\text{-}Some}_P\ \mathit{kvs}\ k\ v$ defined by the formula $\mathit{lookup}\ \mathit{kvs}\ k = \mathit{Some}\ v$, and derive the specialized Horn clauses,

$$\begin{array}{l} k = k' \Longrightarrow \mathit{lookup\text{-}Some}_P\ ((k, v) \cdot \mathit{kvs})\ k'\ v \\ \mathit{lookup\text{-}Some}_P\ \mathit{kvs}\ k'\ v' \Longrightarrow k \neq k' \Longrightarrow \mathit{lookup\text{-}Some}_P\ ((k, v) \cdot \mathit{kvs})\ k'\ v', \end{array}$$

it allows us to enumerate all possible values for k and v given kvs, as the mode analysis now infers mode $i \Rightarrow o \Rightarrow o \Rightarrow \mathit{bool}$ for $\mathit{lookup\text{-}Some}_P$. In general, Quickcheck can derive new predicates with specialized Horn clauses, and refine some premises. As this specialization can give rise to a greater number of modes, it selectively enhances the employed mode analysis.

4.3 Related Work

We have already described some related work for testing functional programs in the related work section of the previous chapter. In this section, we keep our focus on white-box testing approaches for functional and logic programs.

Closely related to our work is the glass-box testing by Fischer and Kuchen [47] for functional logic programs in Curry. They take advantage of narrowing and natively supported nondeterministic executions in Curry. In their setting, test cases are generated with respect to some selected coverage criteria. To record the covered branches of the test case execution, the original functional programs are transformed to return the function's result and the branch decisions. Other related work based on symbolic evaluations is described in §5.5.

Another approach to finding values that fulfill the premises is to use a CLP(FD) constraint solver, as done by Carlier et al. [26, 27] for functional programs. Using constraint solving to find values for logic programs in Mercury is described in [38].

The work of Cheney and Momigliano [29] tests specifications in αProlog [30]. One of their key features is to *eliminate negations* in premises. If the negation elimination is applied successfully, negative occurrences of a predicate are turned into positive occurrences of a derived predicate. If predicates only occur positively in the Horn clauses, the test data generation is simplified because it does not need to handle negation by failure.

Related work to the compilation of inductive specifications to functional programs is described in [16].

Chapter 5

Narrowing-Based Testing

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

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

$$\exists n :: nat.\ \forall m :: nat.\ n = m$$

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

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

$$rev\ xs = xs \implies \exists ys.\ xs = ys{+\!\!+}rev\ ys.$$

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

Narrowing also deals very well with conditional conjectures. In our example with the *delete* operation on red-black trees (cf. chapter 4),

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

the premise *is-rbt t* ensures that the tree t has a black root node, and in fact, after a few refinements, narrowing will only test symbolic values satisfying this property, already pruning away about half of the overall test cases. Further refinements prune the search space even further, enabling us to find unique counterexamples in faulty implementations (§6.3.1).

5.1 Introduction to Narrowing

In this section, we give a short introduction to narrowing, as it forms the theoretical foundations for the evaluation mechanism of functional logic programming languages and our Quickcheck tool. We make use of this common notation from term rewriting (cf. [8]): $t|_p$ denotes the subterm of t at position p, $\sigma(t)$ denotes the application of an substitution σ to a term t, and $t[r]_p$ denotes replacing the subterm $t|_p$ with the term r in t. In our examples, we only need very specific positions: ϵ denotes the root position of a term, l and r denote the position of the first and second argument applied to a binary function symbol, i.e., $f\ a\ b|_l = a$ and $f\ a\ b|_r = b$. With this notation at hand, we define a term rewrite step.

Definition 1. For a term t, $t \to_{p,l\to r} t'$ is a *rewrite step* if

1. a position p is in t,
2. $l \to r$ is a rewrite rule (with fresh variables w.r.t. t),
3. and there exists a substitution σ with $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$.

Term rewriting allows us to describe the evaluation of functional programs in a formal manner. A functional program is described as term rewrite system, which is defined by a set of rewrite rules. The term rewrite system for the append function is given by the two rewrite rules

$$[] +\!\!+\ ys' \to ys' \quad (R_1)$$
$$(x \cdot xs) +\!\!+\ ys \to x \cdot (xs +\!\!+\ ys) \quad (R_2)$$

With this term rewrite system, the term $[\mathsf{a},\mathsf{b}]{+\!\!+}[\mathsf{c},\mathsf{d}]$ is rewritten to $[\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}]$ with the intermediate steps

$$[\mathsf{a},\mathsf{b}]{+\!\!+}[\mathsf{c},\mathsf{d}] \to_{\epsilon,R_2} \mathsf{a} \cdot ([\mathsf{b}]{+\!\!+}[\mathsf{c},\mathsf{d}]) \to_{r,R_2} \mathsf{a} \cdot (\mathsf{b} \cdot ([]{+\!\!+}[\mathsf{c},\mathsf{d}]) \to_{rr,R_1} [\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}].$$

An evaluation by narrowing combines the facets of functional and logic programming and is an interesting model for functional, logic and declarative programming. More precisely, narrowing allows us to apply rewrite rules to expressing containing logical uninstantiated variables. Hence, we can solve equations $t \approx t'$ by finding all possible substitutions σ such that $\sigma(t)$ and $\sigma(t')$ are reducible to the same ground constructor term.

In general, programmers benefit from narrowing most because it adds value to various function definitions. For example, splitting a list can be expressed using the append function and solving the equation $xs{+\!\!+}ys \approx zs$ for some given list zs. For $zs = [\mathsf{a},\mathsf{b}]$, we obtain the solutions $\{xs \mapsto [], ys \mapsto [\mathsf{a},\mathsf{b}]\}, \{xs \mapsto [\mathsf{a}], ys \mapsto [\mathsf{b}]\}, \{xs \mapsto [\mathsf{a},\mathsf{b}], ys \mapsto []\}$. Functions to compute the prefix or suffix of a list can be expressed similarly using the append function. Instead of spending time to write these related functions to append, programmers can simply employ the append function for those purposes.

More specifically for our objective, counterexample generation benefits from narrowing by computing a substitution in a conjecture t that is reducible to *false*. This substitution serves directly as a counterexample to the conjecture.

Formally, a narrowing step extends a rewrite step by an instantiation of variables in t.

Definition 2. For a term t, $t \leadsto_{p,l\to r,\sigma} t'$ is a narrowing step if

1. p is a non-variable position, and
2. $\sigma(t) \to_{p,l\to r} t'$, where σ is a unifier of $t|_p$ and l, and $t' = \sigma(t[r]_p)$.

For rewrite rules for append, the two possible narrowing steps for $xs{+}{+}ys$ are $xs{+}{+}ys \leadsto_{\epsilon,R_1,\{xs\mapsto[],ys'\mapsto ys\}} ys$ and $xs{+}{+}ys \leadsto_{\epsilon,R_2,\{xs\mapsto x'\cdot xs',ys'\mapsto ys\}} x' \cdot (xs'{+}{+}ys)$ where x', xs' and ys' are the fresh variables of the rule R_2.

A narrowing strategy selects the rewrite rule to be applied and its position in the term for its reduction. A strategy can be defined with different (possibly competing) criteria in mind, e.g., the strategy should be easy to implement or it should avoid unnecessary rewrite steps. There are a number of narrowing strategies, e.g., basic narrowing [65], innermost narrowing [49, 59], lazy narrowing [50, 71, 101] and needed narrowing [4]. Needed narrowing is Curry's evaluation mechanism [3] and it is roughly simulated by our Quickcheck tool.

5.2 Existing Narrowing Implementations

Evaluating a term by narrowing can be implemented in at least three different ways:

1. Target a language that natively supports narrowing, such as the functional-logical language Curry, instead of ML.
2. Simulate narrowing by generating a functional program that includes its own refinement algorithm [105].
3. Simulate narrowing by embedding the narrowing-based execution with a library of combinators [48, 77] in a functional language.

We tried out the first two approaches: For our experiments, we extended the code generator to produce source code for Curry with type-class support, which can be executed by the Münster Curry compiler [82]. However, we found that the Curry execution is prohibitively slow, and the second approach using Haskell and a refinement algorithm is considerably faster than the Curry execution. The third approach looks promising but would require a more involved translation.

5.3 Abstract Description of the Narrowing Implementation

In this section, we give an abstract description of the implemented narrowing evaluation mechanism.

At its core, the mechanism evaluates boolean expressions where free variables are replaced by partially instantiated terms. These terms are *constructor terms*, i.e., they are built from datatype constructors and distinct variables, e.g., *Suc n*, *Zero* and *Cons* x_1 (*Cons* x_2 x_3). Exploiting evaluations in Haskell, an expression with partial terms is evaluated to head normal form as far as possible: The execution either returns the ground reduct if it is reduced despite variables in the initial

$\wedge$	T	F	U	X
T	T	F	U	X
F	F	F	F	F
U	U	F	U	X
X	X	F	X	X

$\vee$	T	F	U	X
T	T	T	T	T
F	T	F	U	X
U	T	U	U	U
X	T	X	U	X

Table 5.1: Conjunction and disjunction truth tables

term, or it indicates which variable is critical for the evaluation. For the evaluation of a boolean expression, it yields ground values *true* or *false*, if the expression is true or false for all substitutions of the free variables, resp., or it indicates the critical variable. For example, the execution determines that $Zero \neq Suc\ n$ is true for all natural numbers n, but the value of $Suc\ Zero \neq Suc\ n$ depends on the value of n.

On top of this evaluation for partial terms, there is a refinement algorithm that refutes formulas in prenex normal form.[1] It uses a *refinement tree* that records the results of the evaluation with partial terms and keeps track of refinements. The tree is used to determine the formula's truth value and successive evaluations with partial terms. Figure 5.1 shows the refinement tree during the refutation of the conjecture $\exists n :: nat.\ \forall m :: nat.\ m = n$.

Leafs of the tree carry the evaluation's result: initially *unevaluated* (X), and after the evaluation, the definite results *true* (T) or *false* (F). If the evaluation required a further refinement beyond the maximal bound, the leafs are annotated with *unknown* (U). Inner nodes carry a variable and are classified as universal or existential. Each branch assigns a single constructor with fresh variables as arguments to its parent's variable. A path from the root to a leaf represents an assignment of partial terms by composing the substitutions along the path. For example, the path to the node annotated with T in figure 5.1d assigns *Zero* to n and m.

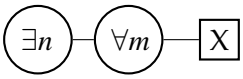
The truth value of a tree is defined recursively: The leafs' values are given by their annotations; the value of a universal node is the conjunction of the values of its subtrees; dually for existential nodes, it is the disjunction of its subtrees. Conjunction and disjunction are defined by truth tables in table 5.1. On the boolean domain, the operators behave as their boolean counterparts. For conjunction, falsity has priority over unevaluated, and unevaluated has priority over truth. Disjunction is defined dually.

Starting with an initial tree with no refinements, the refinement algorithm performs the following three steps:

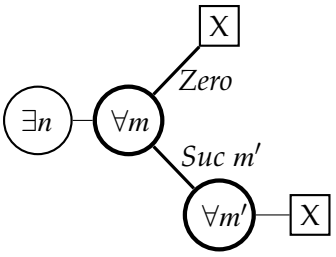
1. It finds by depth-first search a leaf that makes the tree's truth value *unevaluated*, and evaluates the property with the partial terms associated with this leaf.

2. If the evaluation yields a boolean truth value, the algorithm updates the leaf's annotation with the boolean value. If the evaluation calls for a refine-

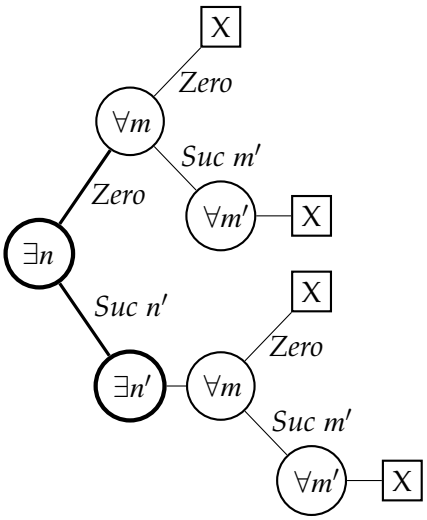
[1] A formula is in *prenex normal form* if it has a prefix of universal and existential quantifiers followed by a quantifier-free part.



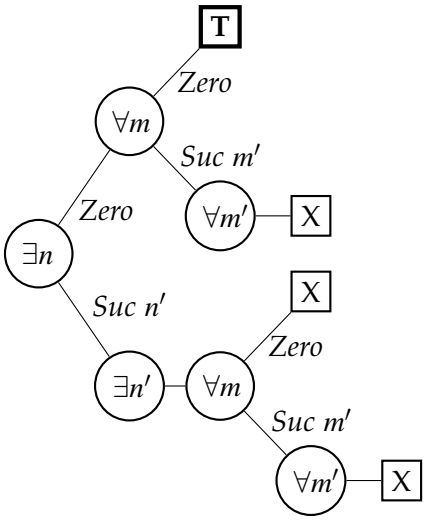
(a) The initial tree



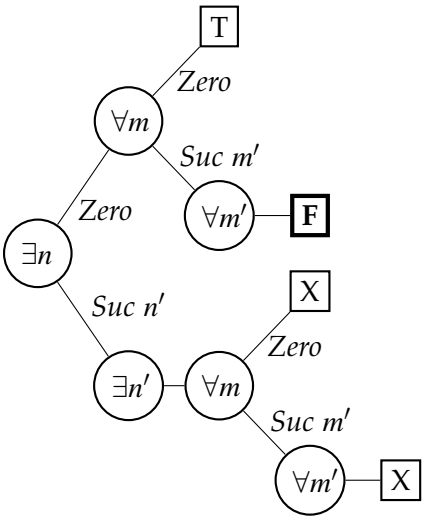
(b) After the first refinement



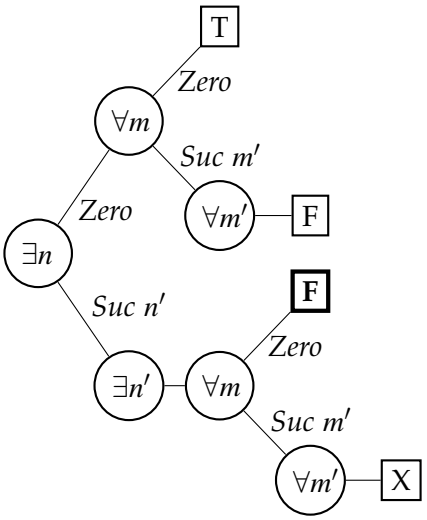
(c) After the second refinement



(d) After the first ground evaluation



(e) After the second ground evaluation



(f) After the third ground evaluation

Figure 5.1: Refinement tree for the evaluation of $\exists n :: nat.\ \forall m :: nat.\ m = n$

ment, the algorithm alters the refinement tree reflecting a case distinction on the critical variable.

3. If the new tree's truth value is false, the algorithm has found a counterexample to the conjecture. If the new tree's truth value remains *unevaluated*, the algorithm continues with the first step. If the tree's truth value is unknown as the evaluation requires too many refinement steps, the algorithm aborts. In rare cases, the tree's truth value might be true, indicating that the conjecture is a theorem.

The illustrated evaluation in figure 5.1 starts with an initial tree that represents the quantifier part of the formula above and one leaf annotated with X (figure 5.1a). The first evaluation of $m = n$ with symbolic values m and n yields to refine m. The top-most constructor of m can either be *Zero* or *Suc* (figure 5.1b). The next evaluation with $m \mapsto Zero$ requires a refinement of n, resulting in the state of figure 5.1c. Now, the evaluation with $n \mapsto Zero, m \mapsto Zero$ yields true (figure 5.1d), and for $n \mapsto Zero, m \mapsto Suc\ m'$ with some fresh variable m' yields false (figure 5.1e). As the truth value of the upper branch $n \mapsto Zero$ is false, we continue with the lower branch $n \mapsto Suc\ n'$. The last evaluation for $n \mapsto Suc\ n', m \mapsto Zero$ yields false, and thus shows the invalidity of the formula (figure 5.1f). We note that the refutation never evaluated $n \mapsto Suc\ n', m \mapsto Suc\ m'$.

Our refinement algorithm is designed for counterexample generation in contrast to proving. This choice plays a role in the chosen semantics of the two operators conjunction and disjunction on the values unknown and unevaluated. As we are interested in finding counterexamples, we define $U \vee X = U$ and $U \wedge X = X$. These choices are justified as follows. Our goal is to find if the formula could possibly evaluate to false. For a formula A with the truth value U, and a formula B that has not been evaluated (truth value X) the formula $A \vee B$ can be found to be true if B is true. It remains unknown otherwise. For finding $A \vee B$ to be false, the evaluation of B is pointless. We avoid it by choosing $U \vee X = U$. On the other hand, $A \wedge B$ becomes false if the truth value of B turns out to be false through its evaluation. For $A \wedge B$, we do not omit the evaluation of B. We choose $U \wedge X = X$. If we were interested in proving properties, we would invert the two choices, i.e., define $U \wedge X = U$, and $U \vee X = X$. In actual developments, it is rather unlikely that proofs are found by narrowing, as it is limited to equational rewriting with the constants' definitions, and structural case distinctions for the occurring variables. In other words, narrowing is limited to proofs with do not require any induction step or any derived property of the functions.

5.4 Implementation

Technically, the implementation consists of two parts. One part of the implementation generates the Haskell program under test employing Isabelle's code generator. Similar to the random and exhaustive testing, the generators for narrowing are defined for each type of the variables in the conjecture. As the types are defined in Isabelle, Quickcheck can inspect the type's definition and automatically define

these generators. The generators, the conjecture and all definitions necessary for the conjecture's evaluation are then transformed into Haskell source code.

The other part of the implementation provides the narrowing-based evaluation with its refinement algorithm. This part is independent of the conjecture under test and was directly implemented in Haskell. The Quickcheck tool combines the code generated by Isabelle and the Haskell source code, compiles it with the Haskell compiler, executes the compiled program, and reads back the program's result. Despite being implemented in Isabelle and Haskell, we focus on the presentation of the combined source code in Haskell's syntax in this section. As Haskell's syntax is similar to Isabelle/HOL's syntax, we introduce Haskell by pointing out the main differences from Isabelle.

In Haskell, types are composed from type variables denoted with lowercase letters, and type constructors are capitalized. Haskell provides among others the boolean type `Bool`, the product type `(a, b)` and the polymorphic list type `[a]` with elements of type `a`. Haskell's lazy datatypes are declared with `data`, type abbreviations with `type`. Signatures, such as `f :: [a] -> [b]`, declare the type of functions.

In expressions, constructors are capitalized; all other values start with a lowercase letter. For example, `True` and `False` are the two boolean constructors, and `not` negates a boolean value. Haskell's notation `\ x -> f x` stands for $\lambda x.\ f\ x$. We use an underscore in an abstraction if the bound variable is not used, as in `\_ -> True`. Furthermore, `[]` and `(:)` are the two list constructors *Nil* and *Cons*.

Haskell provides syntax to define local functions, as in `f x = y where y = x` where `y` is defined locally in the function `f`. We use the `IO` monad with the `return` function and infix notation `(>>=)` for the bind function and employ Haskell's do notation, which makes `do y <- f x; g y` a short cut for `f x >>= (\y -> g y)`.

We use common Prelude functions without further explanations (cf. [69] for their definitions). To avoid CamelCase and underscores in this presentation, we relax Haskell's syntax and allow identifiers to contain hyphens.

5.4.1 Basic Data Structures

Partial terms are represented by the two datatypes `Type` and `Term`:

```
type Position = [Int]
data Type = T [[Type]]
data Term = Var Position Type | Ctr Int [Term]
```

Given a datatype, the `Type` datatype reflects the type's structure as a sum of products. For example, Haskell's boolean type with two nullary constructors `True` and `False` is isomorphic to a sum type of two nullary products (two unit types). Hence, its type value is `boolT = T [[], []]`. Analogously, the Haskell type `(Bool, Bool)` is built from the single binary constructor `Pair` and hence has the type value `T [[boolT, boolT]]`. We can also express recursive datatypes with infinite values: The value for the type of natural numbers with constructors `Zero` and `Suc` is `natT = T [[], [natT]]`. In 5.4.3, we describe the construction of type values for all datatypes in more detail.

Partial terms can either be a variable with some type and a specific position, or a constructor applied to a list of argument terms. Given a mapping from integers to constructors for every datatype, constructors in partial terms are identified by an index. In §5.3, variables were referenced by names, and we invented fresh names during the refinement step. In the actual implementation, we use the positions in the term as names. Positions are simply encoded as a list of integers that describe the path in the tree-shaped term. For example, given the `Nat` type has constructors `Zero` with index 0 and `Suc` with index 1, the abstract partial term *Suc* n' (n' has position [0, 0]) obtained after refinement of variable n (position [0]) has the Haskell value `Ctr 1 [Var [0, 0] natT]`.

For a fixed conjecture, the basic evaluation function `eval` takes a list of partial terms as input and returns an `Answer` value: When the conjecture's evaluation with the partial terms results a boolean value `b`, it returns `Known b`. Otherwise if position `p` must be refined, it returns `Refine p`. As the evaluation uses exceptions, we must employ the IO monad:

```
data Answer = Known Bool | Refine Position
eval :: [Term] -> IO Answer
```

We present the implementation of the evaluation mechanism later in §5.4.3. For now, we consider the function `eval` as a black box.

The central data structure of the refinement algorithm is the refinement tree, which is defined by three datatypes. The tree has three different nodes: leafs that carry truth values, variable nodes with only one branch, and constructor nodes with one branch for each constructor.

```
data Quantifier = Existential | Universal
data Truth = Evaluated Bool | Unevaluated | Unknown
data Tree = Leaf Truth |
  Variable Quantifier Truth Position Type Tree |
  Constructor Quantifier Truth Position [Tree]
```

The variable nodes correspond to inner non-branching nodes in §5.3. The constructor nodes correspond to the inner branching nodes that are obtained by refinement steps. The variable and constructor nodes originated from an universally or existentially quantified variable. This origin is stored as part of the node with the value of type `Quantifier`. The variable nodes carry the information about their variable's type structure and position. Although the truth values in the leafs suffice to determine the tree's truth value, the inner nodes also carry the truth value of their subtree to quickly find the leaf that makes the tree's truth value *unevaluated*. The truth values of inner nodes are kept consistent whenever the truth value of some leaf is changed. As a result, the tree's truth value can always be retrieved from the truth value in the root node:

```
value-of :: Tree -> Truth
value-of (Node v) = v
value-of (Variable _ v _ _ _) = v
value-of (Constructor _ v _ _) = v
```

The trees in figure 5.1a and figure 5.1b depict the values `a` and `b`:

```
a = Variable Existential Unevaluated [0] natT
  (Variable Universal Unevaluated [1] natT (Leaf Unevaluated))
b = Variable Existential Unevaluated [0] natT
  (Constructor Universal Unevaluated [1] natT
    [Leaf Unevaluated, Leaf Unevaluated])
```

5.4.2 Refinement Algorithm

The refinement algorithm iteratively modifies the refinement tree. It finds the next leaf in the tree (which is addressed by its path) and evaluates the conjecture with the partial term corresponding to the selected path (functions `find`, `terms-of` and `eval`). Depending on the answer of the basic evaluation, the tree is refined. Answers of the form `Known b` are recorded in the refinement tree. The position for an answer `Refine p` is used to refine the tree. As the algorithm checks the position's length, it stops the refinement at a fixed depth. As long as there are some leafs to be visited, the algorithm continues the refinement with the updated tree:

```
refute :: Tree -> IO Tree
refute t =
  do
    path <- return (find t);
    a <- eval (terms-of [] path);
    t' <- case a of
        Known b -> return (update path (Evaluated b) t))
      | Refine p ->
          if length p < maximal-depth then
            return (refine-tree path p t)
          else return (update path Unknown t));
    case value-of t' of
      Unevaluated -> refute t'
    | _ -> return t'
```

Besides addressing leafs in the tree, the `path` also carries all information to construct the partial terms, i.e., the positions, the node's kind and the type of a variable or the constructor's index.

```
data Edge = V Position Type | C Position Int
type Path = [Edge]
```

The functions `find` and `update` are easy to implement (listing 5.1). When we update a truth value, we recompute the cached truth value of the nodes along the path to ensure their consistency. Functions `ball` and `bexists`, which compute the new truth value from the given subtrees, are omitted from listing 5.1. They basically encode the truth tables of table 5.1. The function `refine-tree` finds the node with the critical position on the selected path and replaces the variable node by a constructor node with a forest of fresh variable nodes. This replacement is expressed with the `refine` function.

```
refine (Variable quant truth pos (T typss) t) =
  Constructor quant truth pos [foldr
    (\(i,typ) t -> Variable quant truth (pos++[i]) typ t) t
      (zip [0..] typs) | typs <- typss]
```

The functions `replace` and `refine-tree` are then straightforward (listing 5.1).

Listing 5.1: Auxiliary Functions for the Refinement Algorithm

```
position-of :: Edge -> Position
position-of (V p _) = p
position-of (C p _) = p

term-of :: Position -> Path -> Term
term-of p (C [] i : es) = Ctr i (terms-of p es)
term-of p [V [] ty] = Var p ty

terms-of :: Position -> Path -> [Term]
terms-of p es = terms-of' 0 es
  where
    terms-of' i [] = []
    terms-of' i (e : es) = (t : terms-of' (i + 1) rs)
      where
        (ts,rs) = partition (\e -> head (position-of e) == i) (e : es)
        t = term-of (p ++ [i]) (map (map-pos tail) ts)
        map-pos f (V p ty) = V (f p) ty
        map-pos f (C p ts) = C (f p) ts

find :: Tree -> Path
find (Leaf Unevaluated) = []
find (Variable _ _ p ty t) = V p ty : (find t)
find (Constructor _ _ p ts) = C p i : find (ts !! i)
  where
    Just i = findIndex (\t -> value-of t == Unevaluated) ts

update :: Path -> Truth -> Tree -> Tree
update [] v (Leaf _) = Leaf v
update (V _ _ : es) v (Variable q r p ty t) =
    Variable q (value-of t') p ty t'
  where
    t' = update es v t
update (C _ i : es) v (Constructor q r p ts) = Constructor q r' p ts'
  where
    (xs, y : ys) = splitAt i ts
    y' = update es v y
    ts' = xs ++ (y' : ys)
    r' = val ts'
    val = case q of { Universal -> ball; Existential -> bexists}

replace :: (Tree -> Tree) -> Path -> Tree -> Tree
replace f [] t = (f t)
replace f (V _ _ : es) (Variable q v p ty t) =
    Variable q v p ty (replace f es t)
replace f (C _ i : es) (Constructor q v p ts) = Constructor q v p ts'
   where
     (xs, y : ys) = splitAt i ts
     ts' = (xs ++ (replace f es y : ys)
```

```
refine-tree :: [Edge] -> Position -> Tree -> Tree
refine-tree es p t = replace refine (path-of-position p es) t
  where
    path-of-position p es = takeWhile (\e -> position-of e /= p) es
    refine (Variable q r p (T tss) t) =
      Constructor q r p
        [foldr (\(i,ty) t -> Variable q r (p++[i]) ty t) t
           (zip [0..] ts) | ts <- tss]
```

5.4.3 Basic Evaluation Mechanism

To evaluate the conjecture with partial terms, term representations are turned into Haskell terms with exception values at the variable positions using Haskell's native `error` function [70]. Due to Haskell's lazy evaluation, the conjecture can be evaluated with partially-defined values. Hence, the exception values at variable positions are not raised as long as their evaluation is not needed, i.e., evaluation does not pattern-match against the value. E.g., `[] = (error 'a' : error 'b')` evaluates to `False` despite of the exception values on the right-hand side. Thanks to this feature in Haskell, our symbolic evaluations can be mapped to Haskell evaluations. The mechanism ensures that when an exception is raised, the exception value returns the position of the variable in the input term.

For example, the abstract partial term $Suc\ n'$ where n' is the variable at position $[0,0]$ is represented as Haskell term

```
Suc (error (marker : map toEnum [0, 0]))
```

where `marker` distinguishes the refinement exception values from other ones, and Prelude functions `toEnum` and `fromEnum` convert between positions and strings. For the conversion from the partial term datatype to Haskell values, we use a list of *constructor functions*. For each constructor, the constructor function takes partial terms for the constructor's arguments as input and returns the Haskell value of the constructor applied to the arguments that correspond to the given partial terms:

```
conv :: [[Term] -> a] -> Term -> a;
conv cs (Var p uu) = error (marker : map toEnum p);
conv cs (Ctr i xs) = (cs !! i) xs;
```

For example, `conv csnat` converts the partial term `Ctr 1 [Var [0, 0] natT]` to `Suc (error (marker : map toEnum [0, 0]))`, where `csnat` are the constructor functions for the `Nat` type:

```
csnat = [\[] -> Zero, \[t] -> Suc (conv csnat t)]
```

Employing this conversion function, we convert values of the partial term datatype and apply them as arguments to the boolean expression that is the quantifier-free part of the property under test. This boolean value is evaluated with the `answer` function. The function employs Haskell's native functions `try` and `evaluate`. The function `evaluate :: a -> IO a` forces its argument to be evaluated to weak head normal form. We can catch exception values with `try`. The expression `try x`

returns `Right r` if no exception was raised and the evaluation of `x` resulted in value `r`. If an exception was raised, the expression `try x` returns `Left e` where `e` is the exception value. The `answer` function evaluates the boolean value, catches the refinement exception values and converts them into refinement answers that carry the position of the critical variable:

```
answer :: Bool -> IO Answer;
answer a =
  do res <- try (evaluate a)
     case res of
       Right b -> Known b
       Left (ErrorCall (marker : p)) -> Refine (map fromEnum p)
       Left e -> throw e
```

The dedicated type class `Narrowing` provides the constructor functions for the conversion and the description of the type's structure for every datatype:

```
data N a = N Type [[Term] -> a]
class Narrowing a where narrowing :: N a;
```

For example, the narrowing instance for the natural numbers could be defined by the values `natT` and `csnat` from our previous examples. Borrowing the terminology from the other testing approaches, we call values of type `N a` the (narrowing) generator for type `a`. We define generators for all datatypes with three basic combinators. Given a datatype constructor of type `a`, the `cons` combinator builds the associated conversion and the type description, wrapping it into `N a`. The second combinator `apply` applies a generators to another. The third combinator `sum` creates the union of two generators.

```
cons :: a -> N a
cons a = N (T [[]]) [(\ _ -> a)]

apply :: N (a -> b) -> N a -> N b
apply (N (T p s) cfs)) (N t cs) =
  N (T (map (\ ts -> t : ts) ps))
    map (\ cf (x : xs) -> cf xs (conv cs x)) cfs

sum :: N a -> N a -> N a
sum (N (T ta) ca) (N (T tb) cb) = N (T (ta ++ tb)) (ca ++ cb)
```

For a given datatype, generators are composed following the datatype's structure. For example, the generator for type `Nat` is built from `cons Zero :: N Nat` and `cons Suc :: N (nat -> nat)`. To the latter expression, we must apply the generator for type `Nat`. The entire generator, called `narrowing-nat`, is defined as the sum of those two expressions.

```
narrowing-nat :: N nat
narrowing-nat = sum (cons Zero) (apply (cons Suc) narrowing-nat)
```

Similarly for lists, the generator is expressed as sum of constructor `Nil` and the constructor `Cons` with two values applied. As we require a generator for the list's element type, the type for `a` must belong to the `Narrowing` class.

```
narrowing-list :: Narrowing a => N [a]
narrowing-list = sum (cons [])
  (apply (apply (cons (\ a b -> a : b)) narrowing) narrowing-list)
```

Following the scheme of these examples, Quickcheck automatically derives instances for all datatypes.

Finally, values of type `Property` encode formulas in prenex normal form. The `Property` constructor wraps the quantifier-free part of the formula. Quantifiers of the formula are expressed with the functions `exists` and `forall`.

```
data Property =
  UniversalP Type (Term -> Property)
| ExistentialP Type (Term -> Property)
| Property Bool;

exists :: Narrowing a => (a -> Property) -> Property
exists f = ExistentialP ty (\ t -> f (conv cs t))
  where
    N ty cs = narrowing;

forall :: Narrowing a => (a -> Property) -> Property
forall f = UniversalP ty (\ t -> f (conv cs t))
  where
    N ty cs = narrowing;
```

For example, the conjecture $\exists n :: nat.\ \forall m :: nat.\ m = n$ corresponds to this expression:

```
exists (\ n -> forall (\ m -> Property (m = n))
```

Given a property `P`, functions `tree-of` and `eval-of` extract the initial refinement tree and the evaluation function, respectively (cf. listing 5.2). The `refute` function actually takes the aforementioned `eval` function as argument. Now, checking a property `p` is expressed by

```
check p = refute (\ts -> answer (eval-of p ts)) (tree-of 0 p)
```

Before we present the last aspect of the implementation (§5.4.4), we must draw some attention to a delicate issue in this tool. In the beginning of this section, we pointed out that parts of the implementation are formulated in Isabelle/HOL and turned into Haskell code by Isabelle's code generator. More specifically, the narrowing instances of all datatypes are defined in Isabelle. In case of recursive datatypes, the description of the type's structure is an infinite tree, not representable by an inductive datatype, but only by a coinductive datatype. However at the time of its development, Isabelle/HOL only supported the definition and

code generation of inductive datatypes. To define coinductive datatypes, one must derive the type, the constructors and its properties manually with the basic definition mechanisms. As this is laborious, the narrowing instances are axiomatized in Isabelle without providing a proper definition, but only with axioms that allow code generation in Haskell. The consistency of the logical system is not in danger, as the axioms are used only in Quickcheck and are not leaked to the user. The newly introduced support of codatatypes in Isabelle [113] would allow a proper and simple definition of the narrowing generators and make the axioms obsolete.

Listing 5.2: Auxiliary Functions for the Evaluation Mechanism

```
eval-of :: Property -> [Term] -> Bool
eval-of (Property b) = (\[] -> b)
eval-of (UniversalP _ f) = (\(t : ts) -> eval-of (f t) ts)
eval-of (ExistentialP _ f) = (\(t : ts) -> eval-of (f t) ts)

tree-of :: Int -> Property -> Tree
tree-of n (Property _) = Leaf Unevaluated
tree-of n (UniversalP ty f)  = Variable Universal Unevaluated [n] ty
    (tree-of (n + 1) (f undefined))
tree-of n (ExistentialP ty f) = Variable Existential Unevaluated [n] ty
    (tree-of (n + 1) (f undefined))
```

5.4.4 Presentation of Results

After the execution of the refutation algorithm, we must extract *the variable assignment* of the counterexample from the resulting refinement tree. If the property only uses universally quantified variables, we simply extract the partial terms of the variables from the refinement tree. In the presence of existential quantifiers, the counterexamples are not just a single assignment of partial terms, but rather the relevant subtree of the refinement tree. As for the other strategies, we present the counterexample as an assignment of the universally quantified variables, but the variable assignment might depend on the values of the existential quantified variables. These dependencies can entail case distinctions, which are presented by case expressions in the counterexample. To our example $\exists n :: nat.\ \forall m :: nat.\ m = n$, Quickcheck returns the counterexample

$$m = \text{case } n \text{ of } Zero \Rightarrow Suc\ \_ \mid Suc\ \_ \Rightarrow Zero.$$

5.5 Related Work

There are a number of tools that implement or employ narrowing with the goal of counterexample generation. EasyCheck [31] and its successors [46] are testing tools for Curry. As Curry supports narrowing directly, its testing tool can easily employ narrowing for testing Curry programs. Lindblad's work [77] implements a narrowing-based counterexample generator that builds on a very basic compiler for narrowing. The implemented system shows the theoretical benefits for counterexample generation by narrowing, but as the compiler is very rudimentary, the practical benefits are only very moderate.

The program analysis *Reach* [90] computes values that causes the evaluation of a functional program to reach explicitly marked target expressions using a narrowing-based evaluation strategy.

The tool LazySmallCheck [105] uses Haskell's evaluation mechanism, which makes the implementation lightweight and the evaluation fast. Our approach builds on the same evaluation mechanism. However to support existential quantifiers, the necessary refinement algorithm is considerably more involved than LazySmallCheck's refinement algorithm. Recently, LazySmallCheck has been successfully applied in a larger application [102], which tests the correctness of compiler implementations and optimizations.

We chose to implement narrowing by the simulating it in Haskell instead of using a direct translation to Curry as in our experiments executing Curry programs were slower than executing it by simulation (as done by LazySmallCheck). In the meantime, there have been further steps to improve the performance of Curry programs: Various techniques to embed nondeterministic computations into functional programs [2, 48] and the implementation of the Curry compiler KiCS2 [22, 23] have shown very promising results to improve the run time of functional logic programs.

Independently of the work in functional languages, Darga and Boyapati [36, 103] have discovered model checking techniques that are very similar to narrowing for testing data structures and type systems written in object-oriented programming languages.

5.6 Discussion

Combining narrowing and data refinement allows us to evaluate the conjecture with abstractions that are dynamically refined during the evaluation. For example, let us assume that we express the arithmetic operations on real numbers by operations on intervals, i.e., an upper and lower bound approximating the real number. With a suitable datatype refinement, partially instantiated values represent intervals with fuzzy bounds. As a result, a narrowing-based execution evaluates conjectures starting with a large (fuzzy) interval and refinement steps tighten those bounds if needed for the conjecture's evaluation.

To refute conjectures with functions, we already employ a simple data refinement. Functions are approximated by finite functions defined by a simple datatype

$$\textsf{datatype}\ (\alpha,\beta)\ \mathit{ffun} = \mathit{Constant}\ \beta \mid \mathit{Update}\ \alpha\ \beta\ (\alpha,\beta)\ \mathit{ffun}$$

For example, the conjecture over functions f and g

$$\forall\, f\ g\ xs.\ \mathit{map}\ g\ (\mathit{map}\ f\ xs) = \mathit{map}\ (f\ \circ\ g)\ xs$$

is rewritten to a conjecture over the dedicated datatype (α,β) *ffun*,

$$\forall\, f\ g\ xs.\ \mathit{map}\ (\mathit{eval}_{\mathit{ffun}}\ g)\ (\mathit{map}\ (\mathit{eval}_{\mathit{ffun}}\ f)\ xs) = \mathit{map}\ ((\mathit{eval}_{\mathit{ffun}}\ f)\ \circ\ (\mathit{eval}_{\mathit{ffun}}\ g))\ xs,$$

with

$$\textsf{fun}\ \mathit{eval}_{\mathit{ffun}} :: (\alpha,\beta)\ \mathit{ffun} \Rightarrow \alpha \Rightarrow \beta$$

$$\textsf{where}$$

$$\begin{aligned} & eval_{ffun}\ (Constant\ c)\ x = c \\ | & eval_{ffun}\ (Update\ x'\ y\ f)\ x = (\text{if } (x = x') \text{ then } y \text{ else } eval_{ffun}\ f\ x) \end{aligned}$$

The $eval_{ffun}$ function requires the values on the domain to allow an equality check. If this is not possible, e.g., for types such as $(nat \Rightarrow nat) \Rightarrow nat$, we employ a reduced datatype with the *Constant* constructor only.

The functions produced by these simple data refinements were sufficient to refute most invalid conjectures about functions in Isabelle. However, this data refinement cannot express simple functions such as

$$(\lambda x.\ \text{case } x \text{ of } Inl\ l \Rightarrow True \mid Inr\ r \Rightarrow False) :: nat + nat \Rightarrow bool$$

To cover a large set of functions, one could use a tree structure as representation and evaluation functions that turn these trees into functions built by nested case-expressions. However for the current applications, we expect small gains.

Although we can handle existential and universal quantifiers, there are still even very simple conjectures that the narrowing-based evaluation cannot refute, essentially because it is limited to finitely many refinements. For example, the conjecture $\exists x.\ x = Suc\ x$ is invalid. However, narrowing refines the values for the variable x repeatedly, but it cannot ultimately refute the conjecture. A simple disunification check could easily determine that there no value for which $x = Suc\ x$ holds. Undoubtedly, refuting conjectures with existential quantifiers can be very difficult. It could require to derive some sophisticated proof that employs intermediate lemmas and induction. For such conjectures, a narrowing-based evaluation alone might not be powerful enough.

The technique of narrowing and smart generators (§4.2) share the motivation to handle hard-to-satisfy premises. Both techniques reduce the search space of test values by rather different ways. In general, it is very difficult to compare the two techniques. For a given premise, smart generators are constructed after inspection of the premise. The dataflow analysis reorders the original program's dataflow for generating values. Hence typically, smart generators construct values in a bottom-up fashion. Narrowing constructs partial values and refines them, leading to construct values in a top-down fashion. Smart generators are restricted to fully instantiated values. In contrast to narrowing, possible symmetries in the conjecture's conclusion are not exploited by the smart generators. It is interesting to investigate how to combine ideas from these two approaches: Smart generators could construct partial values, or narrowing could employ the mode analysis to reorder premises before evaluating them symbolically.

Once future Curry implementations are still immature; should newer versions perform better, we could consider integrating code generation to Curry in Isabelle. Although Curry's syntax closely follows Haskell's syntax, Curry still lacks the support of type classes. Hence, Isabelle's code generator would have to resolve type classes by a dictionary construction. The evaluation with Curry allows us to execute programs with distinguished features, such as existential quantification and unification constraints, and it offers a new method for executing inductive predicates.

Chapter 6

Empirical Results and Applications

In the previous chapters, we have presented various techniques in Quickcheck to refute conjectures. In this chapter, these techniques are compared on a large set of benchmarks and a number of representative case studies of Isabelle formalizations. In the second part, we describe further applications of Quickcheck.

6.1 Evaluation on Theorem Mutations

To obtain a large set of non-theorems in Isabelle, we derive formulas by *mutating* existing theorems, as in [14, 21]. The formulas are constructed by replacing constants and swapping arguments in the existing theorems. Table 6.1 shows the results of running the counterexample generators on 400 mutated theorems of 13 theories with a liberal time limit of 30 seconds. The chosen set of theories focuses on executable ones, and leaves out those that are obviously not executable. For example, theories with axiomatic definitions or with coinductive datatypes are not executable with Isabelle's code generation. Conjectures in these theories can be checked only by Nitpick. The first nine are basic theories in Isabelle from three different domains: arithmetics (Divides, GCD, MacLaurin), set theory (Fun, Relation, Set, Wellfounded), and simple inductive datatypes (List, Map). The last four theories selected from the Archive of Formal Proofs [74] and cover these domains: graph theory (Max-Card-Matching), formal language theory (Regular-Set) and data structures (Huffman, List-Index).

The four columns show the absolute number of genuine counterexamples of the different approaches: random testing, exhaustive testing, narrowing-based testing, and Nitpick. We omit Refute from this evaluation as it has been superseded by Nitpick. In a cell with values A/B, A is the number of mutants that exhibit a genuine counterexample and B the number of mutants that are executable by the corresponding counterexample generator. As Nitpick is not restricted to a clearly specified fragment, it is in principle able to check all 400 mutants. Quickcheck can use finite types or integers to instantiate polymorphic conjectures (cf. §3.3.3). For theories with polymorphic conjectures, we show both modes separately in the table, indicated with [fin] and [int]. Using finite types for

polymorphic conjectures makes almost all conjectures in the set theory domain amenable to Quickcheck, closing the previously existing gap between Quickcheck and Nitpick in this domain. The narrowing-based testing can execute more conjectures than concrete testing with random and exhaustive testing. We gain most on the Regular-Sets theory, increasing from 304 to 368.

We also compared the tools against each other, and measured the number of counterexamples that can be found uniquely by one tool compared with another. Table 6.2 shows the comparison of random against exhaustive (Ran./Exh.), exhaustive against narrowing (Exh./Nar.), narrowing against Nitpick (Nar./Nit.) and exhaustive against Nitpick (Exh./Nit.). Exhaustive testing slightly outperforms random testing. Narrowing often finds a few more counterexamples than exhaustive testing, but this is mainly due to the larger set of executable formulas. The exceptions, i.e., Fun [fin] and Wellfounded [fin], reveal a weakness with narrowing-based testing: They contain conjectures with many quantifiers over some large finite domains. Exhaustive testing can evaluate ground formulas faster than narrowing, and thus also explores the large finite domains faster. In a few cases, exhaustive testing finds the counterexample where narrowing reaches the time limit. Narrowing and Nitpick complement each other to some extent, as witnessed most prominently by Isabelle's GCD theory. In absolute numbers, narrowing and Nitpick find 228 and 216 counterexamples, respectively; hence differing only by 12. However, they succeed on different conjectures—narrowing finds 23 counterexamples where Nitpick fails, Nitpick finds 11 where narrowing fails—meaning that employing them in combination yields 239 counterexamples.

Quickcheck's major strength—already acknowledged by its name—is its high reactivity. Although the counterexample generators could search for thirty seconds, almost all counterexamples are found by any counterexample generator within the first four seconds. Figure 6.1 shows the total number of refuted conjectures on all theories on a time line. On the large collection of mutated theorems, the counterexample generator mainly respond with different reactivity due to their individual system architectures, but not due to their individual techniques.

Random and exhaustive testing require no external system calls, and immediately respond with counterexamples. Within half a second, both strategies are close to their asymptotic bound. Nitpick requires a fixed start-up time of half a second to start its back-end tool. Then, the number of refuted counterexamples slowly increases while Nitpick steadily increases the scope of its finite approximations, but after two seconds, Nitpick is also close to its bound. The system architecture of Narrowing requires to compile a Haskell program. The compilation for Haskell takes about one second and all refutations are found within the next two seconds.

Nevertheless, the evaluation on automatically generated mutants has to be taken with a grain of salt: Mutating theorems at arbitrary term positions can break very natural abstractions. Some mutants are terms that users cannot even input. For example, in mutants of the arithmetical theories, some arbitrary part of the internal representation of numerals is replaced by a variable, and Quickcheck and Nitpick deal with these obscure terms very differently. Sometimes these conjectures appear executable to Quickcheck at first, but then produce strange errors

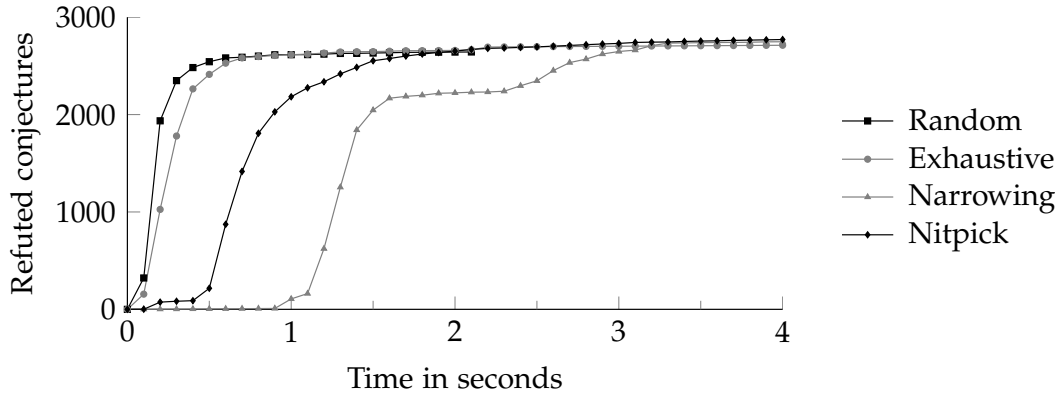


Figure 6.1: Responsiveness of the counterexample generators

during the execution, hindering Quickcheck from refuting those conjectures. Some heuristics in the mutation generation tries to reduce these obscure examples, but still some fraction of the mutants which can be genuinely solved by only one of the tools is due to these candidates.

If all constants of the theory are executable, Quickcheck performs equally well or sometimes even better than Nitpick. As Quickcheck is more responsive than Nitpick, users normally prefer Quickcheck in that case. To illustrate the differences in strength between testing with Quickcheck and model finding with Nitpick, we show two interesting examples of our evaluation. On the one hand, consider one of the monotonicity lemmas for integer division:

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

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

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

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

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

Table 6.1: Results for running counterexample generators on mutated theorems with a time limit of 30 seconds

Theory	Relative comparison			
	Ran./Exh.	Exh./Nar.	Nar./Nit.	Exh./Nit.
Arithmetics				
Divides [fin]	0/13	0/9	8/46	5/52
Divides [int]	0/15	0/9	19/30	16/36
GCD	0/0	0/25	23/11	13/26
MacLaurin [fin]	0/0	0/1	26/0	26/1
MacLaurin [int]	0/0	0/1	37/0	37/1
Set Theory				
Fun [fin]	0/1	18/4	1/35	0/20
Fun [int]	9/7	0/17	0/74	0/91
Relation [fin]	0/2	3/0	8/7	8/4
Relation [int]	0/15	1/6	8/95	8/100
Set [fin]	0/0	0/3	3/14	0/14
Set [int]	0/1	0/14	2/42	0/54
Wellfounded [fin]	1/4	4/4	5/22	5/22
Wellfounded [int]	0/2	2/6	2/200	2/204
Datatypes				
List [fin]	1/1	1/31	11/29	6/54
List [int]	0/2	0/23	10/39	6/58
Map [fin]	0/0	0/0	0/1	0/1
Map [int]	1/3	0/12	0/98	0/110
AFP Theories				
Huffman [fin]	0/4	3/1	0/7	0/5
Huffman [int]	5/6	0/10	2/9	0/17
List-Index [fin]	0/0	0/7	0/8	0/15
List-Index [int]	0/3	0/7	0/8	0/15
Max-Card-Matching [fin]	0/60	0/0	16/18	16/18
Max-Card-Matching [int]	0/0	0/0	0/210	0/210
Regular-Sets	2/0	0/58	85/17	39/29

Table 6.2: Relative comparison of the counterexample generators on mutated theorems

6.2 Evaluation on Conditional Conjectures

To evaluate exhaustive testing, the smart generators (§4.2) and narrowing-based testing (§5), we compared their performance against each other on some conditional conjectures. We compare their performance validating conjectures up to a given size. As random testing checks the conjecture in an incomplete manner, it does not make sense to compare its run time against the other approaches in this setting.

Table 6.3 shows the number of test cases up to a given size, and the number of test cases (for that size) for which the premises *distinct* and *sorted* hold. In other words, we measured the density of the search space if restricted by some premise, compared with the unrestricted search space. For example, testing the proposition $\mathit{distinct}\ \mathit{xs} \Longrightarrow \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})$, 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 of the possible improvement avoiding vacuous tests.

Table 6.4 shows the run time[1] to validate properties with values up to a given size on some representative conjectures from Isabelle's library with the premise *distinct* (D_1, D_2, D_3) and *sorted* (S_1, S_2, S_3):

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

For the premise *distinct xs* and *sorted xs*, employing the smart generators with the depth limit i covers the same set of values as employing the exhaustive generators and narrowing with the size i. In general, the distribution of enumerating values with smart generators up to some depth can differ significantly from enumerating values up to some size. For *distinct* and *sorted*, the depth limit and size coincide and one can compare the run time meaningfully.

The numbers of D_1 indicate the improvement using the smart test generators for *distinct*. In the case of D_2, a more representative conjecture of the Isabelle's theory of lists, we observe a similar behavior. In the case of D_3, the exhaustive testing does not enumerate all pairs of lists for *xs* and *ys*, but only generates lists *ys* if the generated list *xs* is distinct. This simple optimization (cf. §3.1) already reduces the number of useless tests dramatically. With this optimization, only 0.025 percent of all tests are rejected by the premise. As a result, using the smart generator does not add any further significant improvement in the run time behavior. Hence the smart generators perform practically the same to the exhaustive testing. Symbolic

[1] 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.

execution with narrowing performs worst due to its overhead in the execution in all three cases.

On the very sparse premise, *sorted xs*, the improvements with smart test data generators are even more apparent. For example, in the case of 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.

	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 6.3: Number of test cases for given sizes and premise

		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 6.4: 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

6.3 Case Studies

In the evaluation on theorem mutations, we were interested in many conjectures that were simple to refute. The presented case studies have a different flavor: Here, we are interested in the refutation of a few conjectures of some skewed formalization. The formalizations and their intricate conjectures are difficult to refute and allow us to show the benefits of the new techniques.

6.3.1 Functional Data Structures

We evaluated the different testing approaches on faulty implementations of typical functional data structures. We injected faults by adding typos into the correct implementations of the delete operation of AVL trees, red-black trees, and 2-3 trees. By adding typos, we create 10 different (mostly 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 $\mathit{ordered}\ t \Longrightarrow \mathit{ordered}\ (\mathit{delete}\ k\ t)$. We check two similar properties for AVL trees, and three similar properties for red-black trees. With the 10 versions, this yields 20 tests each for 2-3 and AVL trees, and 30 tests for red-black trees, on which we apply various counterexample generators. In this setting, we compare how the techniques deal with *conditional conjectures*. Random testing is applied with 2,000 and 20,000 iterations for each size (abbreviated $\mathrm{R_{2K}}$, $\mathrm{R_{20K}}$). Furthermore, we used exhaustive testing (Exh.), custom generators (Cu.G., §4.1), smart generators (Sm.G., §4.2), narrowing (Nar.) and Nitpick (Nit.).

Table 6.5 summarizes the results. Overall, narrowing, smart, and custom generators beat exhaustive testing, which itself performs better than random testing and Nitpick. Nitpick struggles with large functional programs and is limited to shallow errors in the smaller implementations of AVL and red-black trees. Increasing the number of iterations for random testing helps, but in our experience, it does not find any error that was not also found by testing exhaustively. For the 2-3 trees, the smart generators and narrowing find errors in 5 more cases than exhaustive testing. In principle, exhaustive testing should find the errors eventually. Thus, in these more intricate cases, we increased the time for the naive exhaustive testing to finally discover the fault. Even after one hour of testing, exhaustive testing was not able to detect a single one. This shows that using the test data generators and narrowing-based testing in this setting is clearly superior to

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

Table 6.5: Number of counterexamples on faulty implementations of functional data structures (time limit: 30 s for AVL and red-black trees; 120 s for 2-3 trees)

naive exhaustive testing. The smart generators and narrowing find 12 errors in 20 conjectures. In the eight cases where they did not find anything within the time limit, even testing more thoroughly for an hour did not reveal any further errors. There are two possibilities: Either Quickcheck is not able to reveal the invalidity of the property, or the property still holds, as the randomly injected faults do not necessarily affect the invariant.

6.3.2 Hotel Key Card System

As a further case study, we checked a hotel key card system by Nipkow [92]. The faulty system contains a man-in-the-middle attack that is only uncovered by a trace of length 6.

It models a digital key card system, similar to existing ones in many hotels. 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. In the decentralized key card system, the locks cannot communicate with each other or the reception. Nevertheless, only guests that check 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 protocol: 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 [92] gives a formalization of this hotel key card system in Isabelle, which itself was inspired by a model by Jackson [66]. 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).

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 model the events *Checkin g r* (k, k'), *Enter g r* (k, k'), and *Exit g r* with the *event* datatype.

datatype *event* =
Checkin guest room $(key \times key)$ | *Enter guest room* $(key \times key)$ | *Exit guest room*

A trace, represented as a list of events, describes the temporal order of events taking place. Given a trace *evs*, a room *r* and a guest *g*, the functions *currentkey evs r* and *roomkey evs r* denote the key that is currently recorded at the reception for the room *r* and the last key that the room was entered with, respectively. The functions

issued evs, *cards evs g*, *isin evs r*, *owner evs r* denote the set of already issued keys, the set of cards of a guest g, the guests in a room r, and the owner of a room r. The set of possible and valid traces in a hotel is given by the functional description of the predicate *hotel*.

$$\begin{aligned}
&\mathit{hotel}\ [] = \mathit{True}\\
&\mathit{hotel}\ (e \cdot \mathit{evs}) = (\mathit{hotel}\ \mathit{evs}\ \wedge\ (\text{case } e \text{ of}\\
&\quad \mathit{Checkin}\ g\ r\ (k,k') \Rightarrow k = \mathit{currentkey}\ \mathit{evs}\ r\ \wedge\ k' \notin \mathit{issued}\ \mathit{evs}\ |\\
&\quad \mathit{Enter}\ g\ r\ (k,k') \Rightarrow (k,k') \in \mathit{cards}\ \mathit{evs}\ g\ \wedge\ (\mathit{roomkey}\ \mathit{evs}\ r \in \{k,k'\})\ |\\
&\quad \mathit{Exit}\ g\ r \Rightarrow g \in \mathit{isin}\ \mathit{evs}\ r))
\end{aligned}$$

This reflects the explanation above: An empty trace is valid; the card that the new guest receives when checking in contains the key k of the previous guest and a fresh key k'; a guest can only enter a room if one of the two keys k or k' on a card he owns matches the current room key; and to exit a room, the guest must have been in the room before.

A guest might feel *safe* in room r if he was the last person to check in and the room was empty when he entered the room after his check-in.

$$\begin{aligned}
&\mathit{safe}\ \mathit{evs}\ r = (\exists \mathit{evs}_1\ \mathit{evs}_2\ \mathit{evs}_3\ g\ c\ c'.\\
&\quad \mathit{evs} = \mathit{evs}_3 +\!\!+ (\mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2 +\!\!+ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1)\ \wedge\\
&\quad \mathit{noCheckin}\ (\mathit{evs}_3 +\!\!+ \mathit{Enter}\ g\ r\ c \cdot \mathit{evs}_2)\ r\ \wedge\\
&\quad \mathit{isin}\ (\mathit{evs}_2 +\!\!+ \mathit{Checkin}\ g\ r\ c' \cdot \mathit{evs}_1)\ r = \{\})
\end{aligned}$$

where $\mathit{noCheckin}\ \mathit{evs}\ r = \neg(\exists g\ c.\ \mathit{Checkin}\ g\ r\ c \in \mathit{evs})$.

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 random and exhaustive testing face two problems: Firstly, the naive black-box testing generates many traces for which the *hotel* predicate evaluates to false. Secondly, the *safe* predicate cannot be executed (without refinement) because it contains unbounded existential quantifiers (over an infinite type) for evs_1, evs_2, evs_3.

The smart generators tackle the two problems, generating test data that fulfills the premise, and eliminating the existential quantifiers by its dataflow analysis. For this example, the smart generators find the man-in-the-middle attack within a few seconds.

In this paragraph, we describe the function hotel^o with mode $o \Rightarrow \mathit{bool}$ that serves as smart test data generator for values of the predicate *hotel*. The necessary notions are introduced in chapter 4.

$$\begin{aligned}
&\mathit{hotel}^o = (\mathit{return}^+\ []) \sqcup\\
&\quad (\mathit{hotel}^o \gg\!= (\lambda \mathit{evs}.\ {\mathit{hotel}_{\mathit{aux}}}^{oi}\ \mathit{evs} \gg\!= (\lambda e.\ \mathit{return}^+\ (e \cdot \mathit{evs}))))\\
&{\mathit{hotel}_{\mathit{aux}}}^{oi}\ \mathit{evs} = (\mathit{return}^+\ \mathit{evs} \gg\!= (\lambda \mathit{evs}.\ {\mathit{currentkey}_P}^{ioo}\ \mathit{evs} \gg\!= (\lambda(r,k_1).\ \mathit{exhaustive}_{\mathit{key}}\\
&\quad \gg\!= (\lambda k_2.\ \mathit{not}\ ({\mathit{issued}}^{ii}\ \mathit{evs}\ k_2) \gg\!= (\lambda().\ \mathit{exhaustive}_{\mathit{guest}}\\
&\quad \gg\!= (\lambda g.\ \mathit{return}^+\ (\mathit{Checkin}\ g\ r\ (k_1,k_2)))))))) \sqcup\\
&\quad (\mathit{return}^+\ \mathit{evs} \gg\!= (\lambda \mathit{evs}.\ {\mathit{cards}}^{ioo}\ \mathit{evs} \gg\!= (\lambda(g,(k_1,k_2)).\ {\mathit{hotel}_{\mathit{aux2}}}^{ioii}\ \mathit{evs}\ k_1\ k_2
\end{aligned}$$

$$\gg\!= (\lambda r.\ return^+\ (Enter\ g\ r\ (k_1, k_2))))))\ \sqcup$$
$$(return^+\ evs \gg\!= (\lambda evs.\ isin^{ioo}\ evs\ \gg\!= (\lambda(r, g).\ return^+\ (Exit\ g\ r))))$$

$$hotel_{aux2}{}^{ioii}\ evs\ k_1\ k_2 =$$
$$(return^+\ (evs, (k_1, k_2)) \gg\!= (\lambda(evs, (k_1, \_)).\ roomkey_P{}^{ioi}\ evs\ k_1 \gg\!= (\lambda r.\ return^+\ r)))\ \sqcup$$
$$(return^+\ (evs, (k_1, k_2)) \gg\!= (\lambda(evs, (\_, k_2)).\ roomkey_P{}^{ioi}\ evs\ k_2 \gg\!= (\lambda r.\ return^+\ r)))$$

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 enumerate all guests for *Checkin* events.

Applying the smart generators to the safety property results in the following counterexample trace:

$$Enter\ g_1\ r_0\ (k_1, k_2) \cdot Enter\ g_1\ r_0\ (k_0, k_1) \cdot Checkin\ g_0\ r_0\ (k_2, k_3)$$
$$\cdot\ Checkin\ g_1\ r_0\ (k_1, k_2) \cdot Checkin\ g_0\ r_0\ (k_0, k_1)$$

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 as it does not open the door, so he uses the card from his last stay (k_0, k_1) which unlocks the door.
5. At night, Eve enters the room with card (k_1, k_2).

A subtle error in the key card system causes this jeopardy and can be resolved if Joe would have followed a reasonable safety policy and used only his recent card. Adding this safety policy, Nipkow proved the safety of the key card system.

While the smart generators excel at this conjecture, the other testing approaches perform poorly or need some manual refinements. After refining the formalization and removing the existential quantifiers, the naive random and exhaustive testing fail to find the counterexample within ten minutes of testing, as the search space remains too large. Narrowing-based testing can handle the existential quantifiers in principle, but in practice it performs badly with the deeply nested existential quantifiers in the specification. This renders it impossible to find the counterexample with narrowing. After eliminating the existential quantifiers manually, we also obtain a counterexample with narrowing within a few seconds.

On this trace-based version of the hotel key card system, Nitpick fails to find the counterexample with a time limit of ten minutes. However, Nitpick finds the counterexample on a *state-based* reformulation of the hotel key card system (cf. [21], §6.2). This indicates that Quickcheck and Nitpick excel on formalizations with different specification styles: Nitpick on relational descriptions, Quickcheck on realistic functional programs and trace-based descriptions.

6.3.3 Needham-Schroeder Security Protocol

We show how Quickcheck is used to find a man-in-the-middle attack to the faulty Needham-Schroeder security protocol, based on the formalization of Bella and Paulson [12, 100]. The Needham-Schroeder protocol [91] based on public key encryption consists of three steps:[2]

1. $A \rightarrow B: \{N_A, A\}_{K_{PB}}$
2. $B \rightarrow A: \{N_A, N_B\}_{K_{PA}}$
3. $A \rightarrow B: \{N_B\}_{K_{PB}}$

A and B denote the two agents Alice and Bob, K_{PA} and K_{PB} are the public keys of Alice and Bob. The agents use asymmetric public key encryption for sending secret messages. The agents initially know their own private key and the public keys of all agents. We assume that the public keys were distributed via some public key infrastructure beforehand. Assuming that the private keys of the agents are never compromised, only Alice can decrypt a message that was encrypted with her public key, i.e., only Alice can obtain the plain message X from an encrypted message $\{X\}_{K_{PA}}$. Alice initiates a communication with Bob sending an encrypted message for Bob with a nonce N_A and her name. Bob invents a nonce N_B, and sends a encrypted message with Alice's nonce N_A and a fresh nonce N_B for Alice. Sending back N_A, he proves his authenticity to Alice, as Bob is the only one that could have decrypted the initial message. Finally, Alice authenticate herself by sending back Bob's nonce N_B to Bob. At the end of the protocol, only Alice and Bob know the two nonces N_A and N_B, but they are not known to eavesdroppers. The two nonces N_A and N_B can be used as a secret session key for subsequent communication between those two agents. However, the protocol is vulnerable to a well-known man-in-the-middle attack, initially found by Lowe [80].

We present a simplified formalization in Isabelle, based on Paulson's formalization [100]. On the original formalization, none of the testing approaches can find the attack due to various definitions that cause an explosion of the search space. We removed and simplified some definitions, e.g., some definitions that serve as common basis for several security protocols, but are unnecessary for the specific Needham-Schroeder protocol. We then employed Quickcheck on this simplified formalization to gain some insight about the performance of the testing approaches.

In the formalization, there are three agents, Alice, Bob and the attacking spy:

datatype *agent* = *Alice* | *Bob* | *Spy*

A basic message is an agent's identifier, a key or some unguessable nonce. Messages can also be composed or encrypted with a key:

datatype *key* = *pubK agent* | *priK agent*
datatype *msg* = *Agent agent* | *Nonce nat* | *MPair msg msg* | *Crypt key msg*

[2] We use here the common security protocol notation.

We use $\{|m_1, m_2|\} = MPair\ m_1\ m_2$ for pairing two messages m_1 and m_2.

The communication of agents adheres to the assumptions of the Dolev-Yao model [41]. The only event in the network is sending messages from one agent to another:

datatype *event* = *Says agent agent msg*

We assume that messages cannot be lost in the network, but agents are free to choose whether they react to a message or not. Furthermore, agents cannot distinguish the sender, and the spy can see all messages of the network. In Isabelle, we encode the trace of events in the Needham-Schroeder protocol using an inductive predicate:

inductive *needham* :: *event list* ⇒ *bool*
where
  needham []
| *needham evs* ⟹ *Nonce* N_A ∉ *used evs*
  ⟹ *needham* (*Says A B* (*Crypt* (*pubK B*) {|*Nonce* N_A, *Agent A*|}) · *evs*)
| *needham evs* ⟹ *Nonce* N_B ∉ *used evs*
  ⟹ *Says A′ B* (*Crypt* (*pubK B*) {|*Nonce* N_A, *Agent A*|}) ∈ *set evs*
  ⟹ *needham* (*Says B A* (*Crypt* (*pubK A*) {|*Nonce* N_A, *Nonce* N_B|}) · *evs*)
| *needham evs*
  ⟹ *Says A B* (*Crypt* (*pubK B*) {|*Nonce* N_A, *Agent A*|}) ∈ *set evs*
  ⟹ *Says B′ A* (*Crypt* (*pubK A*) {|*Nonce* N_A, *Nonce* N_B|}) ∈ *set evs*
  ⟹ *needham* (*Says A B* (*Crypt* (*pubK B*) (*Nonce* N_B)) · *evs*)

The first rule provides a starting point for deriving valid protocol runs by defining the empty trace as a valid run. The other three rules encode the three steps of the Needham-Schroeder protocol. They are mainly reflecting our description above, but make some assumptions about the messages sent over the network more explicit. Note that the *needham* predicate describes arbitrarily many protocol interactions between any agents. When checking the conjecture

needham evs ⟹ *Says A B* (*Crypt* (*pubK B*) (*Nonce* N_B)) ∉ *set evs*

with Quickcheck, it returns a counterexample that provides a valid run of the protocol in which Alice witnesses Bob that they share the common secret nonces. Random, exhaustive, and narrowing-based testing cannot find such a run of the protocol within one hour. Smart testing performs great by enumerating only valid traces and finds the valid protocol trace within a tenth of a second.

Determining that the protocol is productive with Quickcheck is pleasant, but the real scenario for a counterexample generator is to detect the man-in-the-middle attack. In the model, the attacker has no ability to send messages to interfere with Alice's and Bob's communication. Following [100], the general rule for the spy

needham evs ⟹ *M* ∈ *synth* (*analz* (*spies evs*))
  ⟹ *needham* (*Says Spy B M* · *evs*)

expresses the spy's ability to send *fake* messages to other participants. Given a set of messages M, the functions *synth* and *analz* describe the messages one can synthesize and analyze from M, respectively. The function *spies* describes the spy's knowledge after the trace of events *evs* (i.e., the set of all messages sent over the network in a trace of events *evs*). Its combination *synth* (*analz* (*spies evs*)) describes the (infinite) set of messages the spy can synthesize after analyzing all messages that the spy has seen in the trace of events *evs*. By adding this rule to the definition of the inductive predicate *needham*, we model the spy's ability to interfere in the protocol.

The *secrecy theorem* states that if the second message of the protocol with nonces N_A and N_B is sent from the honest agent B to the honest agent A, then the spy cannot determine the nonce N_B with his ability to analyze the messages that were sent over the network:

$$\begin{aligned}&\mathit{needham\ evs} \ \wedge\ A \neq \mathit{Spy} \ \wedge\ B \neq \mathit{Spy} \ \wedge\ A \neq B \ \wedge\\ &\mathit{Says}\ B\ A\ (\mathit{Crypt}\ (\mathit{pubK}\ A)\ \{|\mathit{Nonce}\ N_A, \mathit{Nonce}\ N_B|\}) \in \mathit{set\ evs}\\ &\Longrightarrow \mathit{Nonce}\ N_B \notin \mathit{analz}\ (\mathit{spies\ evs})\end{aligned}$$

Unsurprisingly, random, exhaustive, and narrowing-based testing cannot find the man-in-the-middle attack, but even the smart testing does not succeed within one hour. The general rule for the spy is difficult to handle because it allows the spy to send infinitely many different messages with a large degree of branching. Thus the search space of the *needham* predicate explodes and makes it infeasible to find the attack. Quickcheck is not directly applicable.

At this point, we reformulate the general rule to direct the search for test cases to some extent based on the following insight: Although the spy can send many different messages, only very few messages can confuse other participants in the protocol. Hence we restrict the spy to only fake messages that the other participants would react to: The faked messages must match the format of first or second message of the Needham-Schoeder protocol. This is done by replacing the general rule by these two rules:

$$\begin{aligned}&\mathit{needham\ evs} \Longrightarrow\\ &\mathit{needham}\ (\mathit{Says\ Spy}\ B\ (\mathit{Crypt}\ (\mathit{pubK}\ B)\ \{|\mathit{Nonce}\ N_A, \mathit{Agent}\ A|\}) \cdot \mathit{evs})\\ &\mathit{needham\ evs} \Longrightarrow\\ &\mathit{needham}\ (\mathit{Says\ Spy}\ A\ (\mathit{Crypt}\ (\mathit{pubK}\ A)\ \{|\mathit{Nonce}\ N_A, \mathit{Nonce}\ N_B|\}) \cdot \mathit{evs})\end{aligned}$$

After replacing the rule, the testing with smart generators finds Lowe's attack as a counterexample within thirty seconds:[3]

$$\begin{aligned}&\mathit{evs} = [\mathit{Says\ Alice\ Spy}\ (\mathit{Crypt}\ (\mathit{pubK\ Spy})\ (\mathit{Nonce}\ 1)),\\ &\mathit{Says\ Bob\ Alice}\ (\mathit{Crypt}\ (\mathit{pubK\ Alice})\ \{|\mathit{Nonce}\ 0, \mathit{Nonce}\ 1|\}),\\ &\mathit{Says\ Spy\ Bob}\ (\mathit{Crypt}\ (\mathit{pubK\ Bob})\ \{|\mathit{Nonce}\ 0, \mathit{Agent\ Alice}|\}),\\ &\mathit{Says\ Alice\ Spy}\ (\mathit{Crypt}\ (\mathit{pubK\ Spy})\ \{|\mathit{Nonce}\ 0, \mathit{Agent\ Alice}|\})]\\ &A = \mathit{Alice}, B = \mathit{Bob}, N_A = 0, N_B = 1\end{aligned}$$

[3]To read the trace in its chronological order, it must be read from back to front.

To test the specification, we must do some manual work to provide implementations of the inductive sets *analz* and *parts*. We show the necessary refinement for *analz*. The steps for *parts* are analogous. The set *analz* is defined inductively, where *invKey K* stands for the key to decrypt a message encrypted with key *K*:

inductive-set *analz* :: *msg set* ⇒ *msg set*
where
  $X \in H \Longrightarrow X \in analz\ H$
| $\{|X, Y|\} \in analz\ H \Longrightarrow X \in analz\ H$
| $\{|X, Y|\} \in analz\ H \Longrightarrow Y \in analz\ H$
| $Crypt\ K\ X \in analz\ H \Longrightarrow Key\ (invKey\ K) \in analz\ H \Longrightarrow X \in analz\ H$

To enumerate this set, we compute the least fix-point by tabulation. This is guaranteed to terminate, as the set operators preserve finiteness and the set of messages sent over the network is finite. For this example, we just provide a simple but inefficient implementation of the fix-point equation:

$analz\ H = (\text{let}$
  $step = (\lambda m.\ \text{case}\ m\ \text{of}$
    $\{|X, Y|\} \Rightarrow \{X, Y\}\ |$
    $Crypt\ K\ X \Rightarrow \text{if}\ Key\ (invKey\ K) \in H\ \text{then}\ X\ \text{else}\ \{\}\ |$
    $\_ \Rightarrow \{\})$
  $H' = H \cup \bigcup(step`H)$
  in if $H' = H$ then H else $analz\ H')$

Harvesting the library for computing transitive closures [110], one could provide a more efficient implementation by reusing general work-list algorithms, but the performance improvements in this case study would be minor because the computed sets remain small.

Although we fell short of reaching the goal of finding the counterexample in the original formalization, the restricted formulation shows the benefit of the smart testing approach compared with previously existing ones. It gives a dramatic improvement compared with random or exhaustive testing and also outperforms narrowing testing.

However, it was not our main goal to develop a tool that checks security protocols for errors. There are other tools [5, 19, 81] already existing with much more developed techniques. Nevertheless, it is nice to see that our general testing tools can also handle checking security protocols. Finding the man-in-the-middle attack on the original specification in Isabelle remains a challenge. It motivates future work to either integrate helpful analysis for the search and the enumeration of test cases, or to exploit external tools that already incorporate such analyses and transformations.

6.4 Applications

Although counterexample generators cannot prove conjectures, they certainly can bring some evidence for the validity of conjectures: If the counterexample generator does not find a counterexample after checking for a large number of variable

assignments, the conjecture is probably valid—even if we have not found a proof for it. We show two applications of Quickcheck, in which we take advantage of this circumstance.

6.4.1 Synthesis of Conjectures

The tool *IsaCoSy* [68], a program for inductive theory formation, synthesizes conjectures and tries to prove them automatically. To make this process tractable, a sophisticated constraint mechanism generates conjectures and employs Quickcheck to reduce the number of conjectures that are then passed to the automatic inductive prover *IsaPlanner* [40]. The authors noted "that many of the conjectures in this theory which pass counter-example checking, but are not proved by IsaPlanner, appear to be theorems. A random selection of 20 out of the 46 unproved conjectures were proved by hand, and no non-theorems were found, which supports our confidence in Isabelle's counter-example checker for simple equational theories" [68].

By our collaboration, we improved the performance of the synthesis of conjectures. The main bottleneck of the synthesis was the large number of code generator invocations while employing Quickcheck heavily. As generating the test program with the code generator takes about 100 ms (for the typical conjectures) and most non-theorems can be refuted with very few test cases, the generation of a single test program commonly took more time than the actual testing.

To improve its performance, Quickcheck provides a special compilation, in which Quickcheck compiles test programs for multiple conjectures with one invocation of the code generator. When generating test programs for multiple conjectures with one code generator invocation, the overall run time for generating the test programs conjoined is lower than generating the test programs individually. Furthermore, we are not interested in the counterexample, but only if a counterexample exists or not. This allows us to simplify the test programs and reduces the size of the generated code. Overall, our experience showed that the run time reduced by at least one order of magnitude when using the special-purpose compilation in this application.

As the synthesized conjectures are simple equational theorems, the random testing approach suffices to refute the non-theorems. In the future, synthesized theorems with premises for larger development could benefit from the more sophisticated testing approaches.

6.4.2 Detection of Superfluous Assumptions

In a theory that serves as library for further developments, general theorems are more useful than specialized ones. There are many ways in which a theorem can be more specialized than necessary. We focus on the existence of *superfluous assumptions*.

For example, given sets A and B, a function f is injective on $A \cap B$, if f is injective on A or B. However, Isabelle's library only provided the overspecialized theorem

$$\textit{inj-on } f\ A\ \wedge\ \textit{inj-on } f\ B \Longrightarrow \textit{inj-on } f\ (A \cap B)$$

with superfluous assumptions *inj-on f A* and *inj-on f B*, whereas either one of those would suffice. To apply the theorem, users would need to clutter their proof with unnecessary steps to discharge the superfluous assumption.

Another example is found in the theory about lists, which provided the following property of *tl* and *replicate*:

$$n \neq 0 \Longrightarrow \textit{tl}\ (\textit{replicate}\ n\ x) = \textit{replicate}\ (n-1)\ x$$

At first sight, the assumption seems to be necessary: If $n = 0$, the list *replicate n x* is *Nil*, and it does not have a tail. However, *tl Nil* is defined as *Nil*, and hence both sides on the equation are equal, even for the case $n = 0$. Again, the assumption is this property is superfluous.

Isabelle's library also contained a theorem about uniqueness of remainder for integer division, where *divmod-int-rel a b* (q, r) holds if and only if a divided by b is q with remainder r:

$$\textit{divmod-int-rel}\ a\ b\ (q, r)\ \wedge\ \textit{divmod-int-rel}\ a\ b\ (q', r')\ \wedge\ b \neq 0 \Longrightarrow r = r'$$

The first two premises are essential for stating uniqueness, while the third premise seems to be a side condition for the theorem to hold. However in this case, the definition of *divmod-int-rel* ensures the property's validity even for $b = 0$. These three examples already suggest that theorems with superfluous assumptions occur in many theories.

There are various reasons why theory developments might provide a more specific theorem than the most general one:

- *Proving a specialized theorem is simpler than proving a more general one.* Consider this overspecialized theorem about lists and mappings:

  $$\begin{aligned}&\textit{length ys} = \textit{length xs}\ \wedge\ \textit{length zs} = \textit{length xs}\ \wedge\ x \notin \textit{set xs}\ \wedge\\ &\textit{map-of}\ (\textit{zip xs ys})(x \mapsto y) = \textit{map-of}\ (\textit{zip xs zs})(x \mapsto z)\\ &\Longrightarrow \textit{map-of}\ (\textit{zip xs ys}) = \textit{map-of}\ (\textit{zip xs zs})\end{aligned}$$

  The stated fact also holds for lists of unequal lengths, i.e., *length ys* $=$ *length xs* and *length zs* $=$ *length xs* are superfluous assumptions. However, proving this fact requires more effort than the one with the assumptions that the lists are equal length. In the latter case specialized induction and simplification rules can be applied, while for the general theorem more case distinctions are required.

- *The theorems were stated with only one concrete application in mind.* Commonly during a proof development, users notice that theorems about basic functions are missing in the library. Users then prove those theorems as they require them for their own proofs. At that point, they might not be the most general statement.

  Eventually if they are of general interest, these theorems are integrated back to the library theories. Even though the theory development of the libraries is done with great care and attention, an overspecialized theorem occasionally slips into the library theories.

- *They evolve by generalizing definitions.* The library theories in Isabelle are subject to frequent changes by various developers. Definitions are often generalized, and theorems adapted to those generalizations. Modifications of library theories are checked by ensuring that existing developments remain intact. However, this does not guarantee that existing theorems are generalized after the definitions were changed.

- *Users are not aware of the corner cases in some definitions.* In our example above about *tl* and *replicate*, some user might have been misled by the relationship of *hd* and *replicate*, that holds only if $n \neq 0$:

  $$n \neq 0 \Longrightarrow hd\ (replicate\ n\ x) = x$$

  Such confusions about corner cases of definitions frequently occur with functions that are intuitively partial function, but made total in Isabelle/HOL choosing some reasonable value. Examples of such functions are the already seen *tl* for lists, and the division operation for arithmetic domains, i.e., in contrast to common mathematics, x divided by zero is defined to be zero in Isabelle/HOL.[4]

Irrespective of the reasons they arise, it is usually preferable to remove the needless assumptions. We provide a tool that allows users to check if theorems of a theory have superfluous assumptions. It removes assumptions and checks if Quickcheck can find a counterexamples to those modified conjectures. If it finds a counterexample to such a conjecture, the assumptions that were removed are essential for the validity of the original theorem. If it does not find a counterexample, it is likely that the conjecture also holds without the removed assumptions. The tool returns the largest sets of assumptions that can be removed. A largest set of superfluous assumption is not uniquely determined, e.g., it might be possible that two disjoint sets of assumptions can be removed. For example, a conjecture with three assumptions A_1, A_2, and A_3 might be valid with the single assumption A_1, or A_3. Removing A_2 and A_3, or removing A_1 and A_2 is an option, but removing all three assumptions would be invalid.

A naive solution to find all maximal sets of superfluous assumptions is to check for counterexamples for all possible subsets of assumptions. However, this would lead to 2^n many checks for a theorem with n assumptions. Assuming that superfluous assumption are rather rare, we pursue another strategy. Given a theorem with n assumptions, we first check all conjectures where one assumption has been removed. This yields a set of superfluous assumptions for this theorems, which we denote by S_1. We then proceed to check for larger sets of superfluous assumptions. Then we check all conjectures where two assumptions have been removed. The sets of cardinality 2 are constructed by taking two elements of S_1. Assuming that we found conjectures with 2 superfluous assumptions, we proceed building sets of 3, taking the sets of cardinality 2 and adding an element from S_1. We continue this process iteratively for larger sets, until all assumptions are removed, or we find counterexamples for all sets of a given cardinality.

[4] It is debatable whether superfluous assumptions in these theorems should be removed.

Of course, the tool is approximative and it can report *false positives*, i.e., non-theorems for which Quickcheck cannot find a counterexample after an essential assumption has been removed. This is mainly due to two reasons:

- Checking the conjecture times out before encountering a counterexample.
- The representation of values inherently limits Quickcheck to find a counterexample to an invalid conjecture.

Table 6.6 shows the results of this tool on a few theories, that serve as libraries for other developments. The columns show the total number of theorems, the number of theorems with assumptions, the number of theorems where the tool found superfluous assumptions. By manual investigation, we found them to be essential assumptions (false positives reported by the tool) or we were able to make the theorems more precise. This is indicated by the last two columns (false pos. and fixed).

Our experience suggests that once the unnecessary assumptions of a theorem are identified, it is usually easy to modify the existing proof to a similar one which does not rely on the premise: After removing the assumption, the proof remains either unchanged as it never relied on the assumption, or one adds a case distinction on this assumption in the proof, where the new case is usually trivial to prove. For some cases (i.e., two cases in Divides, six cases in GCD, one case in RComplete and three cases in Map), we did not improve the theorems as they were either only for internal use of another proof tool or modifying the proof such that it did rely on the assumption was too difficult to be done within a few minutes. For example, the very specific theorem about maps obtained from zipped lists and updating maps,

$$\begin{aligned}&\mathit{map\text{-}of}\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys})(x \mapsto y) = \mathit{map\text{-}of}\ (\mathit{zip}\ \mathit{xs}\ \mathit{zs})(x \mapsto z)\ \wedge\ x \notin \mathit{set}\ \mathit{xs}\ \wedge\\ &\mathit{length}\ \mathit{ys} = \mathit{length}\ \mathit{xs}\ \wedge\ \mathit{length}\ \mathit{zs} = \mathit{length}\ \mathit{xs}\\ &\Longrightarrow \mathit{map\text{-}of}\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys}) = \mathit{map\text{-}of}\ (\mathit{zip}\ \mathit{xs}\ \mathit{zs}),\end{aligned}$$

is valid without the last two assumptions, but the existing proof uses the assumptions heavily and a proof without those assumptions requires a much longer proof compared with the existing one. For such theorems, we did not consider the improvement worth the effort.

The benefits were already palpable, as we could simplify a few proofs in further developments and sustain theorems free from superfluous assumptions after major modifications in the system [61].

The presented tool was built to show a simple useful application of Quickcheck. For further research, there are many directions one could take from here:

- Currently, it only finds counterexamples using Quickcheck, but it could also employ Nitpick if Quickcheck fails to find a counterexample quickly. This could help reduce the number of false positives in a few cases.
- Once a premise of a theorem has been removed, a further step would be to automatically find proofs that relied on the overspecialized theorem and detect if they can be simplified, as some parts of the proof were only required to prove the assumption of the overspecialized theorem.

Theory	total	w.assms	sf.assms	false pos.	fixed
Arithmetics					
Divides	313	142	15	0	13
GCD	255	106	7	1	0
RealDef	253	56	4	0	4
RComplete	103	47	4	0	3
Set Theory					
Fun	155	91	1	0	1
Relation	193	59	4	4	0
Set	471	176	1	0	1
Wellfounded	114	68	3	3	0
Datatypes					
List	844	308	8	1	7
Map	133	59	6	1	3
AFP Libraries					
List-Index	35	16	2	0	2
Regular-Sets	44	15	0	0	0
Matrix	126	63	3	2	1

Table 6.6: Detected superfluous assumptions in Isabelle's libraries

- Automatic detection of such "smells" is just the first step in the process of improving the quality of theories. A next step is to automatically refactor the proof document, as known from modern integrated development environments for programming languages.

- Unnecessary premises is just one of many possible over-specializations. One could also extend the tool to detect other over-specializations. For example, users frequently state theorems with overly strict arithmetic bounds, e.g., a theorem with a premise $i < j$ might also be valid for $i \leq j$.

- An alternative approach to detect unnecessary premises is to automatically inspect the proof term of theorems [13], and this way find assumptions that were not used in the proof. Analyzing the proof term is limited to find superfluous assumptions based on a syntactic criterion, but the proof term analysis ensures that the existing proof is still appropriate. However, it cannot detect superfluous assumptions if the existing proof requires the assumption, but there exists a proof that does not require the assumption. In contrast, employing counterexample generators detects superfluous assumptions based on semantic observations: It can potentially provide false positives, does not yield a fixed proof, but can detect superfluous assumptions beyond the syntactic criterion of the proof term analysis. It would be interesting to see how the results of these two approaches differ on Isabelle's current theories.

Chapter 7

Conclusion

7.1 Results

Earlier versions of Quickcheck, which test with random values only, were already very useful. Although exhaustive testing is limited to explore the property with very small values, we have seen that counterexamples are often found before one reaches larger values. In our benchmarks and case studies, exhaustive testing slightly outperformed random testing. For the programs and properties that one encounters in Isabelle, we have come to the conclusion that exhaustive testing is better suited than random testing. In general, it certainly depends on various factors of the program, e.g., its size and its complexity, and the property to check. Isabelle's users benefit from having both strategies at their disposal: they can choose the best for their actual development.

When random and exhaustive testing fail to find an existing counterexample within reasonable time, other methods are unlikely to help. The reasons are commonly that the problem's nature entails a huge, asymmetric search space or that the naive methods do not take an important facet into account, e.g., the existence of a premise or many symmetric evaluations. We could address the latter with two further testing approaches, testing with specialized generators (smart testing) and symbolic evaluation. Especially in case of the smart testing approach, getting from the principal idea to the actual implementation was not just an engineering effort, but required some research and many experiments to obtain a competitive testing technique.

Many further improvements in this thesis are related to making larger parts of Isabelle's theories executable. This was certainly beneficial for all testing approaches.

7.2 Future Work

In the introduction, we have mentioned that counterexample generation in Isabelle is split into two camps, led by the tools Nitpick and Quickcheck. Both counterexample generators have gathered a set of techniques for refuting conjectures in the past years. A next logical step is to combine their techniques.

Taking a high level view of Nitpick and Quickcheck, they differ in two main aspects, the evaluation order and the representation of values. Nitpick is more flexible in these two aspects than Quickcheck. It has no fixed evaluation order, as the evaluation order is dynamically determined by the underlying boolean satisfiability solver. Furthermore, it applies various encodings for boolean values (either as two-valued type or approximated with a three-valued type), sets, predicates and functions.

Quickcheck has a fixed evaluation order dominated by the evaluation order of the employed programming language. Furthermore, Quickcheck globally fixes the representation for every type, which is mainly determined by user's setup of the code generator. To make Quickcheck in these two aspects more like Nitpick, we could provide different data and program refinements for the code generator that allow more flexible representations of values, similar to the different encodings in Nitpick. However datatype and program refinement is technically much more involved than Nitpick's translation. Isabelle's code generator requires to extend theories by defining new constants and proving theorems for the translation. Another difficulty is that Nitpick's translation performs global optimizations, but the code generation's optimizations must be local ones to combine them with the current code generation process.

Equipping the code generator with other execution principles would allow different evaluation orders in Quickcheck. One such example is providing automatic compilation techniques to tabulate sets efficiently. However, a compilation that naively enumerates all values of a set probably quickly exhibits a large set of values that exhausts all available physical memory. Tabled Prolog systems, such as XSB [111], and Datalog systems show that the enumeration of sets is manageable if one employs further techniques, such as the magic set transformation [10]. By integrating this technique, one could enumerate sets up to a suitable size for counterexample generation. Combining this with functional executions and lazy enumerations might allow us to check further specifications. Instead of targeting a functional programming language, an alternative is to translate to languages with different evaluation mechanisms. Suitable candidates are the functional logic programming language Curry [3] and logic programming languages XSB and λProlog [88].

Further good targets are satisfiability-modulo-theories (SMT) solvers [11, 37]. SMT solvers drive the evaluation by the underlying boolean satisfiability solver and integrate further mechanisms elegantly. For example, functional programs could be evaluated with the built-in equational reasoner and sets could be enumerated with the built-in Datalog engine.

An alternative is to provide a dedicated model finder for higher-order logic, similar to first-order model finders SEM [122] and Mace4 [83]. This model finder would eliminate the need for a translation to other paradigms. To compete against the existing counterexample generators, it must be fairly scalable and efficient. We could implement such a model finder by combining evaluation mechanisms with different embeddings, i.e, the evaluation would combine ground execution, symbolic evaluation, such as normalization by evaluation [1] and narrowing, and Isabelle's provers, such as the simplifier and the tableau reasoner.

Bibliography

[1] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2008.

[2] Abdulla Alqaddoumi, Sergio Antoy, Sebastian Fischer, and Fabian Reck. The pull-tab transformation. In *Preproceedings of the Third International Workshop on Graph Computation Models (GCM 2010)*, 2010.

[3] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53:74–85, 2010.

[4] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *Journal of the ACM*, 47:776–822, 2000.

[5] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang (Erlang 2006)*, pages 2–10. ACM, 2006.

[7] Philippe Ayrault, Matthieu Carlier, David Delahaye, Catherine Dubois, Damien Doligez, Lionel Habib, Thérèse Hardin, Mathieu Jaume, Charles Morisset, François Pessaux, Renaud Rioboo, and Pierre Weis. Trusted software within Focal. In *Computer & Electronics Security Applications Rendez-vous (CESAR 2008)*, pages 162–179, 2008.

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[9] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *the 5th International Conference on Mathematical Knowledge Management (MKM 2006)*, volume 4108 of *Lecture Notes in Artificial Intelligence*, pages 31–43. Springer, 2006.

[10] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS 1986)*, pages 1–15. ACM, 1986.

[11] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[12] Giampaolo Bella. *Inductive Verification of Cryptographic Protocols*. PhD thesis, University of Cambridge, 2000.

[13] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2000)*, Lecture Notes in Computer Science, pages 38–52. Springer, 2000.

[14] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

[15] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL – lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36, 1999.

[16] 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 *Lecture Notes in Computer Science*, pages 131–146. Springer, 2009.

[17] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In Andrew D. Gordon, editor, *19th European Symposium on Programming, Programming Languages and Systems (ESOP 2010)*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer, 2010.

[18] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004.

[19] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 82–96. IEEE Computer Society, 2001.

[20] Jasmin Christian Blanchette and Alexander Krauss. Monotonicity inference for higher-order formulas. *Journal of Automated Reasoning*, 47(4):369–398, December 2011.

[21] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[22] Bernd Braßel, Michael Hanus, Björn Peemöller, and Florian Reck. KiCS2: a new compiler from Curry to Haskell. In *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, volume 6816 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2011.

[23] Bernd Braßel, Sebastian Fischer, Michael Hanus, and Fabian Reck. Transforming functional logic programs into monadic functional programs. In *Proceedings of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, volume 6559 of *Lecture Notes in Computer Science*. Springer, 2011.

[24] Benjamin Canou and Alexis Darrasse. Fast and sound random generation for automated testing and benchmarking in objective Caml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML (ML 2009)*, pages 61–70. ACM, 2009.

[25] Matthieu Carlier and Catherine Dubois. Functional testing in the focal environment. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs (TAP 2008)*, volume 4966 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.

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

[27] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. A first step in the design of a formally verified constraint-based testing tool: FocalTest. In Achim D. Brucker and Jacques Julliand, editors, *Tests and Proofs (TAP 2012)*, volume 7305 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2012.

[28] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In David Hardin and Julien Schmaltz, editors, *Proceedings of the 10th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 70 of *Electronic Proceedings in Theoretical Computer Science*, pages 4–19, 2011.

[29] James Cheney and Alberto Momigliano. Mechanized metatheory model-checking. In *Principles and Practice of Declarative Programming (PPDP 2007)*, pages 75–86. ACM, 2007.

[30] James Cheney and Christian Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *20th International Conference on Logic Programming (ICLP 2004)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283, 2004.

[31] Jan Christiansen and Sebastian Fischer. EasyCheck – test data for free. In Jacques Garrigue and Manuel Hermenegildo, editors, *9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[32] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[33] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

[34] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In Graham Hutton and Andrew P. Tolmach, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 149–160. ACM, 2009.

[35] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: guessing formal specifications using testing. In *Tests and Proofs (TAP 2010)*, Lecture Notes in Computer Science, pages 6–21. Springer, 2010.

[36] Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 363–382. ACM, 2006.

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[38] François Degrave, Tom Schrijvers, and Wim Vanhoof. Automatic generation of test inputs for Mercury. In Michael Hanus, editor, *Logic-Based Program Synthesis and Transformation*, volume 5438 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2009.

[39] John Derrick, Neil Walkinshaw, Thomas Arts, Clara Benac Earle, Francesco Cesarini, Lars-Ake Fredlund, Victor Gulias, John Hughes, and Simon

Thompson. Property-based testing – the ProTest project. In Frank de Boer, Marcello Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 250–271. Springer, 2010.

[40] Lucas Dixon and Jacques D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In Franz Baader, editor, *Proceedings of the 19th International Conference on Automated Deduction (CADE-19)*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283, 2003.

[41] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[42] Jonas Duregard, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell (Haskell 2012)*, pages 61–72, 2012.

[43] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In David A. Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2003.

[44] Carl Eastlund. Doublecheck your theorems. In *8th Int. Workshop On The ACL2 Theorem Prover and Its Applications*, 2009.

[45] Sebastian Fischer. Reinventing Haskell backtracking. In *GI Jahrestagung 2009*, pages 2875–2888, 2009.

[46] Sebastian Fischer. *On Functional Logic Programming and its Application to Testing*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2010.

[47] Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Principles and Practice of Declarative Programming (PPDP 2007)*, pages 63–74. ACM, 2007.

[48] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. In Graham Hutton and Andrew P. Tolmach, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 11–22. ACM, 2009.

[49] Laurent Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *IEEE International Symposium on Logic Programming (SLP 1985)*, pages 172–184, 1985.

[50] Elio Giovannetti, Giorgio Levi, Corrado Moiso, and Catuscia Palamidessi. Kernel-LEAF: a logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139–185, 1991.

[51] Michael Gordon and Tom Melham. *Introduction to HOL: A theorem proving environment for Higher Order Logic*. Cambridge University Press, 1993.

[52] Florian Haftmann. Data refinement (raffinement) in Isabelle/HOL. This is a draft of an envisaged publication still to be elaborated which, applying the usual rules of academic confidentiality, can be inspected at http://www4.in.tum.de/~haftmann/pdf/data_refinement_haftmann.pdf.

[53] Florian Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.

[54] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In *10th International Symposium on Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010.

[55] John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[56] Ralf Hinze. Deriving backtracking monad transformers. In Martin Odersky and Philip Wadler, editors, *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 186–197. ACM, 2000.

[57] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003.

[58] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In *ICS, Utrecht University*, pages 72–149. Springer, 2006.

[59] Steffen Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Computer Science*. Springer, 1989.

[60] Paul Hudak, Philip Wadler, Arvind Brian, Boutel Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27, 1992.

[61] Brian Huffman. personal communication, 2012.

[62] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Isabelle Users Workshop*, 2012.

[63] John Hughes, Ulf Norell, and Jérôme Sautret. Using temporal relations to specify and test an instant messaging server. In *Proceedings of the 5th Workshop on Automation of Software Test (AST 2010)*, pages 95–102. ACM, 2010.

[64] John M. Hughes and Hans Bolinder. Testing a database for race conditions with QuickCheck. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang (Erlang 2011)*, pages 72–77. ACM, 2011.

[65] Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert A. Kowalski, editors, *Proceedings of the 5th International Conference on Automated Deduction (CADE-5)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

[66] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[67] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

[68] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, October 2011.

[69] Simon L. Peyton Jones. Haskell 98: Standard prelude. *Journal of Functional Programming*, 13(1):103–124, 2003.

[70] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1999)*, pages 25–36. ACM, 1999.

[71] Juan José Moreno-Navarro, Herbert Kuchen, and Rita Loogen. Lazy narrowing in a graph machine. In Hélène Kirchner and Wolfgang Wechler, editors, *Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317. Springer, 1990.

[72] Vesa A.J. Karvonen. Generics for the working ML'er. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML (ML 2007)*, pages 71–82. ACM, 2007.

[73] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[74] Gerwin Klein, Tobias Nipkow, and Larry Paulson. The Archive of Formal Proofs. http://afp.sf.net/.

[75] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: generic automated software testing. In *Revised Selected Papers of the 14th international Workshop on Implementation of functional languages (IFL 2002)*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2003.

[76] Christopher League. QCheck/SML. http://contrapunctus.net/league/haques/qcheck/.

[77] Fredrik Lindblad. Property directed generation of first-order test data. In Marco Morazán, editor, *The Eigth Symposium on Trends in Functional Programming (TFP 2007)*, pages 105–123. Intellect, 2008.

[78] Andreas Lochbihler. Jinja with threads. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/JinjaThreads.shtml, 2007. Formal proof development.

[79] Andreas Lochbihler. Formalising FinFuns – generating code for functions as data from Isabelle/HOL. In Stefan Berghofer, Tobias Nipkow, Chrsitian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference of Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 310–326. Springer, 2009.

[80] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[81] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Journal of Computer Security*, pages 53–84. Society Press, 1998.

[82] Wolfgang Lux. Implementing encapsulated search for a lazy functional logic language. In Aart Middeldorp and Taisuke Sato, editors, *Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 1999.

[83] William McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.

[84] Christopher S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence, 1981.

[85] Louis Morgan. Random Testing of ML Programs. Master's thesis, School of Informatics, University of Edinburgh, 2010.

[86] Michal Muzalewski. An outline of PC Mizar. Technical report, Fondation Philippe le Hodey, Brussels, 1993.

[87] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In *5th International Logic Programming Conference (ICLP 1988)*, pages 810–827. MIT Press, 1988.

[89] Lee Naish. Adding equations to NU-Prolog. In Jan Maluszynski and Martin Wirsing, editors, *3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP 1991)*, volume 528 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 1991.

[90] Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 133–142, Washington, DC, USA, 2007. IEEE Computer Society.

[91] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21 (12):993–999, December 1978.

[92] Tobias Nipkow. Verifying a hotel key card system. In *3rd International Colloquium on Theoretical Aspects of Computing (ICTAC 2006)*, volume 4281 of *Lecture Notes in Computer Science*. Springer, 2006. Invited paper.

[93] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[94] David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of Mercury. In *Principles and Practice of Declarative Programming (PPDP 2002)*, pages 109–120. ACM, 2002.

[95] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods (AFM 2006)*, 2006.

[96] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[97] Rex Page. Property-based testing and verification: a catalog of classroom examples. In Andy Gill and Jurriaan Hage, editors, *Proceedings of the 23rd Symposium on Implementation and Application of Functional Languages (IFL 2011)*, volume 7257 of *Lecture Notes in Computer Science*. Springer, 2012.

[98] Lawrence C. Paulson. *Isabelle – A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[99] Lawrence C. Paulson. *ML for the working programmer (2. ed.)*. Cambridge University Press, 1996.

[100] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, January 1998.

[101] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *International Symposium on Logic Programming*, pages 138–151, 1985.

[102] Jason S. Reich, Matthew Naylor, and Colin Runciman. Lazy generation of canonical test programs. In *Proceedings of the 23rd Symposium on Implementation and Application of Functional Languages*. To appear.

[103] Michael Roberson, Melanie Harries, Paul T. Darga, and Chandrasekhar Boyapati. Efficient software model checking of soundness of type systems. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA 2008)*, pages 493–504. ACM, 2008.

[104] Céline Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14(2):219–232, 1994.

[105] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the 1th ACM SIGPLAN Symposium on Haskell (Haskell 2008)*, pages 37–48, 2008.

[106] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, December 2002.

[107] Jan-Georg Smaus, Patricia M. Hill, and Andy King. Mode analysis domains for typed logic programs. In *Sel. papers from the 9th Int. Workshop on Logic Programming Synthesis and Transformation*, pages 82–101. Springer, 2000.

[108] Roma Sokolov. Ocaml-QuickCheck. https://github.com/camlunity/ocaml-quickcheck.

[109] Michael Spivey. Combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, July 2000.

[110] Christian Sternagel and René Thiemann. Executable transitive closures of finite relations. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/Transitive-Closure.shtml, 2011. Formal proof development.

[111] Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming, Special Issue*, 12 (1-2):157–187, 2012.

[112] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[113] Dmitriy Traytel, Andrei Popescu, and Jasmin Christian Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*, pages 596–605. IEEE, 2012.

[114] Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 1985.

[115] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[116] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 1989)*, pages 60–76. ACM, 1989.

[117] 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.

[118] Tjark Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, April 2008.

[119] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.

[120] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 1997)*, volume 1275 of *Lecture Notes of Computer Sciences*, pages 307–322, 1997.

[121] Ashley Yakeley. MonadPlus reform proposal. http://www.haskell.org/haskellwiki/MonadPlus_reform_proposal, 2006. [Online; accessed 24-July-2012].

[122] Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995)*, pages 298–303. Morgan Kaufmann, 1995.